CHAPTER

10 Forward Seismic Simulation

Reflection seismology is the imaging of the Earth's subsurface structure using sound waves. It is widely used in the oil and gas industry to determine where to find hydrocarbons and most efficiently extract them. *Reverse time migration* (RTM) is one approach to imaging that yields high quality images but requires much computation, and hence is desirable to parallelize.

The example here is not the entire RTM algorithm, but the key part that dominates the computational burden. The obvious parallelization with the **stencil** pattern (Section 4.6.1) suffers from low **arithmetic intensity**, even when using **tiling** for cache (Section 7.3). However, the arithmetic intensity can be raised much further by recognizing that the problem involves not only a stencil in space but also a **recurrence** (Section 7.5) in space–time, since the stencil is iterated. This is a common pattern in solvers for partial differential equations. Using recursive subdivision on this space–time recurrence results in an efficient **cache-oblivious** algorithm with good arithmetic intensity.

10.1 BACKGROUND

A reflection seismology survey consists of collecting data in the field and processing it to generate a subsurface image. The collection step involves sending acoustic waves from *sources* into the Earth and recording the echoes at *receivers*. Examples of land sources are explosives or trucks with massive thumper/vibrator machinery. Water sources are typically devices called *air guns* that pop out compressed air. Receivers are microphones staked into the ground or towed behind a boat. Often a line or grid of receivers record the echoes from a single shot.

Reflection seismology works because different kinds of rock have different velocities and acoustical impedances. Where sound crosses between rocks of varying impedance, some of the sound is reflected from the boundary. The waveform recorded by the receiver is a superposition of reflections from the shot, along many paths in the subsurface, including paths with multiple reflections. For further information, see the program Seismic Duck (`http://sourceforge.net/projects/seismic-duck/`), which provides an animated introduction to reflection seismology.

Reverse time migration is a way of generating a subsurface image from the source/receiver data. RTM starts by modeling acoustic waves both forward and backward in time. Because of symmetries in the underlying physics, the two directions are essentially the same computation and differ only in boundary conditions:

- The forward model includes sources running forward in time as a boundary condition.
- The reverse model includes receivers running backward in time as boundary condition.

Structured Parallel Programming. DOI: 10.1016/B978-0-12-415993-8.00010-4

Each model computes a time-varying acoustic signal at each point in the subsurface volume. RTM generates an image such that the value of each point in the image is the time correlation of the signals at that point from the forward and reverse models. The intuition behind doing this is that a rock transition reflects the signal from the source to the receiver. At a reflection point the signals from the forward model and reverse model will match and generate a high correlation value. There may be points where coincidental correlations happen. Summing images for many different shot and receiver locations dilutes the effect of these coincidental matches.

Wavefield models vary in sophistication and computational burden. Our example will use a simple model, the acoustic wave equation, which accounts only for pressure effects. More complex models, such as the elastic wave equation, account for shear effects, at the cost of significantly more memory and arithmetic operations. The physics for the acoustic wave equation involves two fields over points (x, y, z) in the subsurface volume:

- $A_t(x, y, z)$ is the pressure at point (x, y, z) at time t.
- $V(x, y, z)$ is the velocity of sound at point (x, y, z).

Field V is independent of time, but precisely knowing it requires knowing the kind of rock at point (x, y, z). For example, the value of V is much higher for salt than sandstone. This might seem like a hopelessly circular problem—you need to know the rock structure before you can image it! Fortunately, an iterative process works. Start with a rough initial guess for V and generate an approximate RTM image. Use the resulting image to refine the estimate of V, and rerun RTM. This process can be repeated until the image seems reasonable to a geophysicist.

10.2 STENCIL COMPUTATION

Our example code focuses on computing the forward and reverse models of the wavefield, because those are the computationally intensive steps in RTM. The acoustic wave equation is:

$$\frac{\partial^2 A}{\partial t^2} = V^2 \nabla^2 A, \tag{10.1}$$

where the Laplacian $\nabla^2 A$ is defined as:

$$\nabla^2 A = \frac{\partial^2 A}{\partial x^2} + \frac{\partial^2 A}{\partial y^2} + \frac{\partial^2 A}{\partial z^2}. \tag{10.2}$$

Our example code discretizes A and V into three-dimensional (3D) grids and approximates the Laplacian $\nabla^2 A$ with a stencil. Here is the discretized version of the equations:

$$A_{t+1} = 2A_t - A_{t-1} + V^2 (C * A_t), \tag{10.3}$$

where t is a time index and $C * A_t$ denotes the convolution of a stencil C with the value of field A at time t. As a minor optimization, the code stores a field V^2 instead of V in an array. The code stores field A in a four-dimensional array, with $A_{t,x,y,z}$ stored in `A[t&1][z][y][x]`. The expression `t&1` computes `t` modulo two. Storing two snapshots of A in time suffices because at each time step, the value A_{t+1}

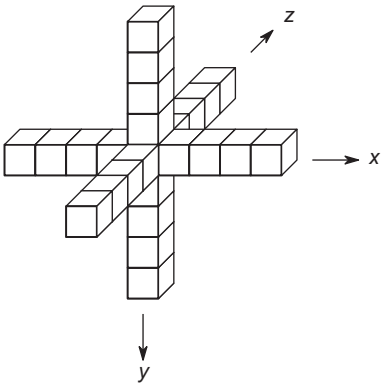

FIGURE 10.1

Structure of stencil for estimating Laplacian.

depends only upon A_t and A_{t-1}. Thus, the location of $A_{t-1}(x,y,z)$ can be overwritten with the value of $A_{t+1}(x,y,z)$.

Figure 10.1 shows the structure of the stencil C. Each pair of opposite arms estimates the second partial derivative in that direction. For example, the arms pointing in the $+x$ and $-x$ directions estimate $\frac{\partial^2 A}{\partial x^2}$. The six arms differ only in direction and thus there are only five independent coefficients, which the code stores in an array `C`. Listing 10.1 shows code for using the stencil serially.

10.3 IMPACT OF CACHES ON ARITHMETIC INTENSITY

One way to parallelize the code in Listing 10.1 is to use the geometric decomposition pattern (Section 6.6). There is a large amount of available parallelism, because for a given value of `t`, the inner three loops can all be parallel. Grids in RTM tend to be as big as memory will allow. For exposition, assume that the spatial grid is $500 \times 500 \times 500$, for a total of 125×10^6 grid points. For single-precision values, that works out to 1.0 GByte for `A` and 0.5 GByte for `V`. That is potential 125×10^6-way parallelism! In fact, just changing the `z` loop to a `cilk_for` enables 500-way thread parallelism, and marking the `x` loop with `#pragma simd` enables 500-way vector parallelism. Here is what the changed code looks like:

```
        ...
        // Apply stencil over [x0,x1) × [y0,y1) × [z0,z1)
        cilk_for (int z=z0; z < z1; ++z)
           for (int y=y0; y < y1; ++y)
#pragma simd
              for (int x=x0; x < x1; ++x) {
                 ...
              }
```

```
1  // Time−domain simulation of wavefield for t ∈ [t0,t1) over volume [x0,x1) × [y0,y1) × [z0,z1)
2  void serial_stencil( int t0, int t1,
3     int x0, int x1,
4     int y0, int y1,
5     int z0, int z1)
6  {
7     // Compute array strides for y and z directions
8     int sy = Nx;
9     int sz = Nx*Ny;
10    // March through time interval [t0,t1)
11    for( int t=t0; t<t1; ++t) {
12       // Apply stencil over [x0,x0) × [y0,y1) × [z0,z1)
13       for (int z=z0; z<z1; ++z)
14          for (int y=y0; y<y1; ++y)
15             for (int x=x0; x<x1; ++x) {
16                int s = z * sz + y * sy + x;
17                // a points to A_t(x,y,z)
18                float *a = &A[t&1][s];
19                // a_flip points to A_{t−1}(x,y,z)
20                float *a_flip = &A[(t+1)&1][s];
21                // Estimate ∇²A_t(x,y,z)
22                float laplacian = C[0] * a[0]
23                   + C[1] * ((a[1] + a[−1]) +
24                           (a[sy] + a[−sy]) +
25                           (a[sz] + a[−sz]))
26                   + C[2] * ((a[2] + a[−2]) +
27                           (a[2*sy] + a[−2*sy]) +
28                           (a[2*sz] + a[−2*sz]))
29                   + C[3] * ((a[3] + a[−3]) +
30                           (a[3*sy] + a[−3*sy]) +
31                           (a[3*sz] + a[−3*sz]))
32                   + C[4] * ((a[4] + a[−4]) +
33                           (a[4*sy] + a[−4*sy]) +
34                           (a[4*sz] + a[−4*sz]));
35                // Compute A_{t+1}(x,y,z)
36                a_flip[0] = 2*a[0] − a_flip[0] + Vsquared[s] * laplacian;
37             }
38    }
39 }
```

LISTING 10.1

Serial code for simulating wavefield.

Though the changes usually speed up the code, they usually fall far short of the hardware capabilities. The problem is *arithmetic intensity*, as defined in Section 2.4.2. Consider the floating-point arithmetic and memory operations required for one point update:

- 29 memory operations:
  - 25 reads from `a` for the stencil.
  - 1 read for `a_flip[0]`.
  - 1 write to `A_flip[0]`.
  - 1 read from `Vsquared`.

  The duplicate read of `a[0]` does not count, since either the compiler or cache will turn it into a single read.
- 33 floating-point operations:
  - 25 additions
  - 1 subtraction
  - 7 multiplications

Hence, the arithmetic intensity is $33/29 \approx 1.14$. This intensity is about an order of magnitude lower than what would keep arithmetic units busy on typical current hardware.

A cache hierarchy helps raise the intensity by eliminating some reads from main memory. This effect happens along several axes. As the inner `x` loop advances, the stencil reads some locations that were read by the previous iteration. Figure 10.2 visualizes this. Each `x` iteration moves the stencil along the x axis, leaving the ghost of its former locations in cache. Any parts overlapping the ghost are already in cache. Only the parts with exposed faces on the right side contribute to memory loads. So the cache has automatically removed 8 loads, raising the arithmetic intensity to $33/(29-8) \approx 1.57$.

In fact, a similar thing happens on the y-arms. If the cache is big enough, then most of the loads for the y-arm will be loads from cache because they overlap ghosts of former loads, except for the load of `a[4*sy]`. The elimination of most of the loads along the x-arm and y-arms raises the arithmetic intensity to $33/(29-8-7) \approx 2.36$, about twice the uncached version.

Is caching of y-arm loads a reasonable assumption for Listing 10.1? To analyze this, consider what one iteration of the `y` loop would load into cache:

- A row of `a` corresponding to each cube with a dark face in Figure 10.2. That is a total of 17 rows. The rows corresponding to the z-arms do not really need to be kept in cache, but the hardware does not know that.

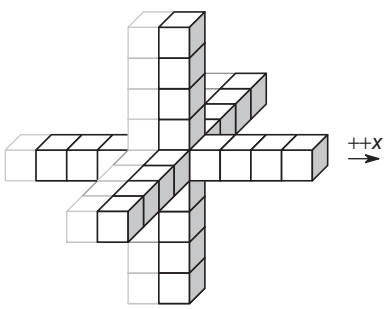

FIGURE 10.2

Reuse of memory when stencil steps along the x axis.

- A row of `a_flip`.
- A row of `Vsquared`.

That is 19 rows of 500 elements each, or $19 \times 500 \times 4\text{bytes} = 38$ KBytes. Allowing for imperfect associativity and imperfect approximations of least recently used (LRU), this will certainly fit in cache, though perhaps not in the innermost cache.

WARNING

If the x and y grid dimensions are exactly powers of two, there is a risk that the y-arm and z-arm will all map to the same associative set, possibly exceeding the size of the associative set of the cache and thus force premature evictions, which can cause slowdown.

What about caching of the z-arms? If the z-arm loads could be cached long enough to be reused, then each array element has to be loaded only once, and the arithmetic intensity increases significantly to $33/4 = 8.25$. That requires about 198 KByte of ideal cache for our problem dimensions, for a single thread. That is well within the size of caches on modern CPUs, though not the innermost and fastest cache. However, if the cache is shared by multiple threads, there might not be enough cache to go around. As coded, the problem is reaching the limits of the free benefit from caches.

10.4 RAISING ARITHMETIC INTENSITY WITH SPACE–TIME TILING

But there is another way to code the problem that raises the arithmetic intensity much higher. A fourth dimension in the problem is a recurrence in time. Suppose the code runs for 10 timesteps. Imagine a cache that is big enough to hold all of the arrays across all timesteps. Then the arithmetic intensity grows by a factor of 10! Alas, that requires a 1.5×10^9 byte cache, several orders of magnitude beyond current offerings and surely not the innermost cache.

Fortunately, there is a trick that gets most of the benefit without a behemoth cache. Chop the problem into cache-sized subproblems. Each subproblem will simulate the wavefield over a chunk of space–time. This chunking is possible because for any value $A_t(x, y, z)$, its influence on the rest of the simulation propagates at a top speed of four grid units per time step, because the longest stencil arm is four grid points. This upper limit on propagation of information acts as a "speed of light" for the simulation.

By way of analogy, consider the impact of the real speed of light on a simulation of the Earth. The Earth could be simulated for an 8-minute chunk of time, starting with the state of space within 8 light-minutes of Earth, without knowing the state of the rest of the universe. For example, the state of the sun, even if it blew up, would be irrelevant for such a simulation time-frame because the sun is 8.3 light minutes away. The principle is similar for our simulation—the upper bound on the speed of information propagation bounds what state information is necessary to know to model a region over a limited time interval. The limit on information propagation in the seismic simulation has a similar effect, though there is a slight geometric difference. The propagation of information in the code follows

```
1  for( int t=t0; t<t1; ++t )
2     for( int i=i0; i<i1; ++i ) {
3        float* b = &B[t&1][i];
4        float* b_flip = &B[t+1 & 1][i];
5        b_flip[0] = f(b[−1],b[0],b[1],b_flip[0]).
6     }
```

LISTING 10.2

Code for one-dimensional iterated stencil. The code marches a one-dimensional field B through time, using a flip-flop indexing scheme similar to the one in Listing 10.1.

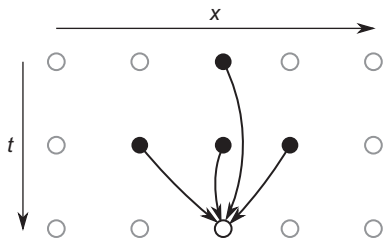

FIGURE 10.3

Dependence of value in space–time for one-dimensional problem in Listing 10.2. Computing the value of $B_{t+1,x}$ requires knowing only four other values in space–time: $B_{t,x-1}$, $B_{t,x}$, $B_{t,x+1}$, and $B_{t-1,x}$.

"taxicab geometry"; thus, the set of points within a given distance forms an octohedron instead of a sphere.

Space-time geometry is a bit tricky to visualize for four dimensions, so we explain the details for a one-dimensional (1D) problem first, using a three-point stencil, and then extend it to multiple dimensions. Listing 10.2 shows the code for the 1D problem, which abstracts the stencil as a function `f`. Figure 10.3 shows the dependencies for each point in space–time. To update a point does *not* require the value of all points for the previous timestep. It requires only the values of its neighbors in the previous timestep. Thus, a subgrid cells can be updated multiple steps into the future without updating the entire grid, as shown in Figure 10.4.

Choosing a trapezoid small enough to fit in cache raises arithmetic intensity, because:

- The arithmetic operations will be proportional to its area.
- The cache misses will be proportional to the length of its top or bottom, whichever is longer.

So **tiling** space-time with trapezoids can raise the arithmetic intensity significantly.

Explicitly choosing the **tile** size based on the cache size could be done but requires knowing the cache size. Furthermore, there are typically multiple levels of cache. So a better approach is to adopt a **cache-oblivious** strategy of recursive tiling. Trapezoids decompose into trapezoids, and some of the subtrapezoids can be evaluated in parallel, similar to the way that diamonds were decomposed into

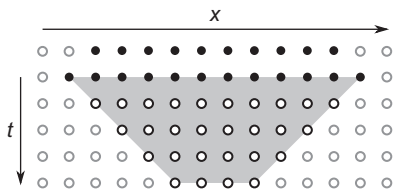

FIGURE 10.4

Trapezoidal space–time region for one-dimensional stencil. All points in the trapezoid can be computed from its top points, without knowing the value of any other points.

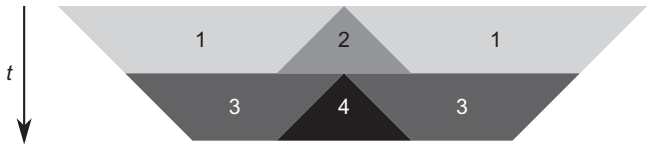

FIGURE 10.5

Decomposing a space–time trapezoid.

diamonds in Section 8.12. Figure 10.5 shows such a decomposition. Horizontal cuts are cuts in time; slanted cuts are cuts in space. The first level of recursion did a time cut. The numerical labels indicate order of evaluation. Trapezoids with the same label can be evaluated in parallel. Note that the triangles are trapezoids with zero-length tops, so they can also be decomposed. Further recursive decomposition of the triangles and trapezoids creates recursive parallelism.

10.5 CILK PLUS CODE

The trapezoid trick of the previous section works in multiple dimensions. The only change is that the trapezoids are now four dimensional. Space cuts can be in any of the three spatial dimensions. It is difficult to visualize, but not difficult to extrapolate from the 1D case. A space–time trapezoid in the 1D case can be represented by four integers: `x0`, `x1`, `dx0`, `dx1`, where:

- `[x0,x1)` is the half-open interval representing the top of the trapezoid.
- `dx0` and `dx1` are the corresponding slopes of the sides with respect to `t`.

In the 1D example where the stencil arms extend only one unit, the slopes are usually 1 or −1, and 0 at a boundary of the grid. For the 3D example, the arms extend four units, so the slopes are usually 4 or −4, and 0 at boundaries of the grid.

Listing 10.3 shows the base case code. It is almost identical to Listing 10.1, except that after each time step it adjusts the boundaries of the spatial hyperplane by the corresponding slope values. It also adds a `#pragma simd` that gives the compiler permission to vectorize the inner loop.

```
1  void base_trapezoid(int t0, int t1,
2     int x0, int dx0, int x1, int dx1,
3     int y0, int dy0, int y1, int dy1,
4     int z0, int dz0, int z1, int dz1 )
5  {
6     // Compute array strides for y and z directions
7     int sy = Nx;
8     int sz = Nx*Ny;
9     // March through time [t0,t1)
10    for (int t=t0; t < t1; ++t) {
11       // Apply stencil over [x0,x0) × [y0,y1) × [z0,z1)
12       for (int z=z0; z < z1; ++z)
13          for (int y=y0; y < y1; ++y)
14 #pragma simd
15             for (int x=x0; x < x1; ++x) {
16                // Update one point. The code here is the same as the
17                // body of the x loop in Listing 10.1.
18                ...
19             }
20       // Slide down along the trapezoid
21       x0 += dx0; x1 += dx1;
22       y0 += dy0; y1 += dy1;
23       z0 += dz0; z1 += dz1;
24    }
25 }
```

LISTING 10.3

Base case for applying stencil to space–time trapezoid.

Listing 10.4 shows the recursive code. It biases cutting towards using the longest spatial axis that permits a cut into K trapezoids. A value of `K = 2` works well in practice. The three space cuts are essentially the same logic, only reoriented along a different spatial axis. Time cuts are used as a last resort, since they do not directly introduce parallelism, though they may open opportunities for more space cuts of the resulting subtrapezoids. See Section 8.8 for a similar situation for the choice of cuts in recursive matrix multiplication.

The strategy shown requires two synchronizations for each split along an axis, one after each `cilk_for` loop. Thus, splitting each dimension once adds six synchronizations to the span. It turns out that splitting all three dimensions at once enables using only four synchronizations [TCK+11], thus enabling greater parallelism. The logic is substantially more complicated to code, so it seems like a technique best left to "stencil compilers" such as Pochoir [TCK+11].

The gain from the space–time restructuring can help even sequential code. On one of the author's machines, just the cache-oblivious algorithm without `#pragma simd` yielded a 1.6× speed improvement, using just a *single* thread!

```
1  const int ds = 4; // Slant of a space cut (in grid units per time step)
2
3  #define TRAPEZOID(u0,u1) \
4     u0+i*w,                  /* left side */ \
5     ds,                      /* left slope */ \
6     i<K-1 ? u0+(i+1)*w : u1, /* right side */ \
7     -ds                      /* right slope */
8
9  #define TRIANGLE(u0,du0,u1,du1) \
10    i<K ? u0+i*w : u1,      /* left side */ \
11    i==0 ? du0 : -ds,       /* left slope */ \
12    i<K ? u0+i*w : u1,      /* right side */ \
13    i<K? ds : du1           /* right slope */
14
15 void recursive_trapezoid(int t0, int t1,
16          int x0, int dx0, int x1, int dx1,
17          int y0, int dy0, int y1, int dy1,
18          int z0, int dz0, int z1, int dz1 )
19 {
20    int dt = t1-t0;
21    if( dt>1 ) {
22       int dx = x1-x0, dy = y1-y0, dz v z1-z0;
23       if (dx >= dx_threshold && dx >= dy && dx >= dz && dx >= 2*ds*dt*K) {
24          int w = dx / K;
25          cilk_for (int i=0; i<K; ++i)
26             recursive_trapezoid(t0, t1,
27                TRAPEZOID(x0,x1),
28                y0, dy0, y1, dy1,
29                z0, dz0, z1, dz1);
30          cilk_for (int i=K; i>=0; --i)
31             recursive_trapezoid(t0, t1,
32                TRIANGLE(x0,dx0,x1,dx1),
33                y0, dy0, y1, dy1,
34                z0, dz0, z1, dz1);
35          return;
36       }
37       if (dy >= dyz_threshold && dy >= dz && dy >= 2*ds*dt*K) {
38          int w = dy / K;
39          cilk_for (int i=0; i<K; ++i)
40             recursive_trapezoid(t0, t1,
41                x0, dx0, x1, dx1,
42                TRAPEZOID(y0,y1),
43                z0, dz0, z1, dz1);
44          cilk_for (int i=K; i>=0; --i)
45             recursive_trapezoid(t0, t1,
46                x0, dx0, x1, dx1,
47                TRIANGLE(y0,dy0,y1,dy1),
```

```
48                 z0, dz0, z1, dz1);
49           return;
50        }
51        if (dz >= dyz_threshold && dz >= 2*ds*dt*K) {
52           int w = dz / K;
53           cilk_for (int i=0; i<K; ++i)
54              recursive_trapezoid(t0, t1,
55                 x0, dx0, x1, dx1,
56                 y0, dy0, y1, dy1,
57                 TRAPEZOID(z0,z1));
58           cilk_for (int i=K; i>=0; --i)
59              recursive_trapezoid(t0, t1,
60                 x0, dx0, x1, dx1,
61                 y0, dy0, y1, dy1,
62                 TRIANGLE(z0,dz0,z1,dz1));
63           return;
64        }
65        if (dt > dt_threshold) {
66           int halfdt = dt / 2;
67           recursive_trapezoid(t0, t0 + halfdt,
68              x0, dx0, x1, dx1,
69              y0, dy0, y1, dy1,
70              z0, dz0, z1, dz1);
71           recursive_trapezoid(t0 + halfdt, t1,
72              x0 + dx0*halfdt, dx0, x1 + dx1*halfdt, dx1,
73              y0 + dy0*halfdt, dy0, y1 + dy1*halfdt, dy1,
74              z0 + dz0*halfdt, dz0, z1 + dz1*halfdt, dz1);
75           return;
76        }
77     }
78     base_trapezoid(t0, t1,
79        x0, dx0, x1, dx1,
80        y0, dy0, y1, dy1,
81        z0, dz0, z1, dz1);
82  }
```

LISTING 10.4

Parallel cache-oblivious trapezoid decomposition in Cilk Plus. The code applies a stencil to recursively divide a space–time trapezoid.

10.6 ARBB IMPLEMENTATION

The discussion so far has focused on aggressive hand restructuring of the serial code. ArBB offers a competitive approach based on the premise that if you specify the computation at a sufficiently high level then the ArBB runtime can take care of the optimizations.

```
void rtm_stencil(arbb::dense<arbb::f32> Acoef,
   arbb::f32 in, arbb::f32& out, arbb::f32 vsq)
{
   using namespace arbb;
   f32 laplacian = coef[0] * in;

   for (int n = 1; n <= 4; ++n) {
      laplacian += Acoef[n] *
         (neighbor(in, -n, 0, 0) + neighbor(in, n, 0, 0) +
          neighbor(in, 0, -n, 0) + neighbor(in, 0, n, 0) +
          neighbor(in, 0, 0, -n) + neighbor(in, 0, 0, n));
   }
   out = 2 * in - out + vsq * laplacian;
}

void arbb_stencil(arbb::usize t0, arbb::usize t1, arbb::dense<arbb::f32> coef,
   arbb::dense<arbb::f32, 3> Ac, arbb::dense<arbb::f32, 3>& An, arbb::dense<arbb
       ::f32, 3> vsq)
{
   using namespace arbb;

   _for (usize t = t0, t < t1, ++t) {
      _if (t % 2 == 0) {
         map(rtm_stencil)(coef, Ac, An, vsq);
      }
      _else {
         map(rtm_stencil)(coef, An, Ac, vsq);
      } _end_if;
   } _end_for;
}
```

LISTING 10.5

ArBB code for simulating a wavefield.

The ArBB implementation of the stencil given in Listing 10.5 is simple because ArBB supports stencils as a built-in pattern, internally implementing many of the stencil optimizations discussed in Section 7.3, in particular strip mining. However, it does not currently implement the recursive space–time trapezoid decomposition, discussed in the previous section.

Syntactically, in the **elemental functions** used for `map` operations in ArBB, not only can the current element be used as an input but also offset inputs can be accessed using the `neighbor` function, which takes an offset. The offset must be a normal C++ type, not an ArBB type, which means it can be computed at ArBB function definition time, but not at runtime. As far as ArBB is concerned, neighbor offsets are constants, and the stencil shape is a constant.

Using offsets means that at the boundaries of a collection, some offsets will be outside the bounds of the input collection. In ArBB, such accesses always return 0. If a different treatment of boundaries is

needed, the current position can be found using a `position` call and the stencil computation modified accordingly. There is no requirement that the stencil computation be linear or uniform, as it can be a function of the position as well. However, the compiler for ArBB is smart enough that if conditional statements are given that are linear functions of the position, which is the case when we only want to modify the stencil within some fixed distance of the boundary, then these conditional statements will only be evaluated near the boundaries of the input collection and not everywhere in the interior.

10.7 SUMMARY

Stencils are an effective approach to solving the wave equation in the time domain. Memory bandwidth, not hardware threads, can easily become the limiting resource if the stencil is parallelized in the obvious way, because the arithmetic intensity will be low. Caches ameliorate the iterated stencil problem somewhat, though greater gains can be obtained by tiling. The iterated stencil problem is a stencil pattern in space and recurrence pattern in time, so the tiles have a trapezoidal shape in space–time. The slopes of the sides relate to the stencil dimensions. Tiling recursively enables a cache-oblivious algorithm, which optim- izes for all possible levels and sizes of cache while being oblivious to which really exist. Because the resulting code uses cache more efficiently, it often runs faster, even when run with a single thread, than the original serial code runs.