CHAPTER

4 Map

This chapter goes into depth on the **map** pattern, introduced in Section 3.3.2. Both serial and parallel versions of this pattern are given in Figure 4.1. The map pattern compresses the time it takes to execute a loop, but it only applies when all instances of the loop body are independent.

Map applies a function to every element of a collection of data in parallel. More generally, map executes a set of function invocations, each of which accesses separate data elements. We will call the functions used in map **elemental functions**. The elemental functions used in a map should have no side effects to allow all the instances of the map to be able to execute in any order. This independence offers maximum concurrency with no need to synchronize between separate elements of the map, except upon completion. There is, however, no assumption or requirement that the instances of the map actually *will* be run simultaneously, or in any particular order. This provides the maximum flexibility for scheduling operations.

The map pattern is simple, being just a set of identical computations on different data, without any communication. It is so simple it is sometimes called **embarrassing parallelism**. However, while conceptually simple, map is the foundation of many important applications, and its importance should not be underestimated. It is important to recognize when map can be used since it is often one of the most efficient patterns. For example, if you do not have one problem to solve but many, your parallel solution may be as simple as solving several unrelated problems at once. This trivial solution can be seen as an instance of the map pattern.

Map is often combined with other patterns to make new patterns. For example, the **gather** pattern is really a combination of a serial random read pattern and the map pattern. Chapter 5 discusses a set of patterns that often combine with the map pattern, the **collectives**, including **reduction** and **scan**. Chapter 6 discusses various patterns for data reorganization, which also often combine with map. Often map is used for the basic parallel computation, and then it is combined with other patterns to represent any needed communication or coordination. Chapter 7 also discusses the **stencil** and **recurrence** patterns, which can be seen as generalizations of map to more complex input and output dependencies. Some additional generalizations, such as **workpile**, are also briefly discussed in this chapter.

Patterns have both semantic and implementation aspects. The semantics of the map pattern are simple, but achieving a scalable implementation often requires a surprising amount of care for best performance. For example, invoking a separate thread for each function invocation in a map is not a good idea if the amount of work each instance does is small. Threads provide **mandatory parallelism**, which is unnecessary in the case of a map, and potentially too heavyweight—a tasking model, which is lightweight but specifies only **optional parallelism**, is more suitable. It is also important to parallelize the overhead of synchronization at the beginning and end of the map and to deal with the fact that the functions invoked in each instance of the map may, in the general case, vary in the amount of

Structured Parallel Programming. DOI: 10.1016/B978-0-12-415993-8.00004-9

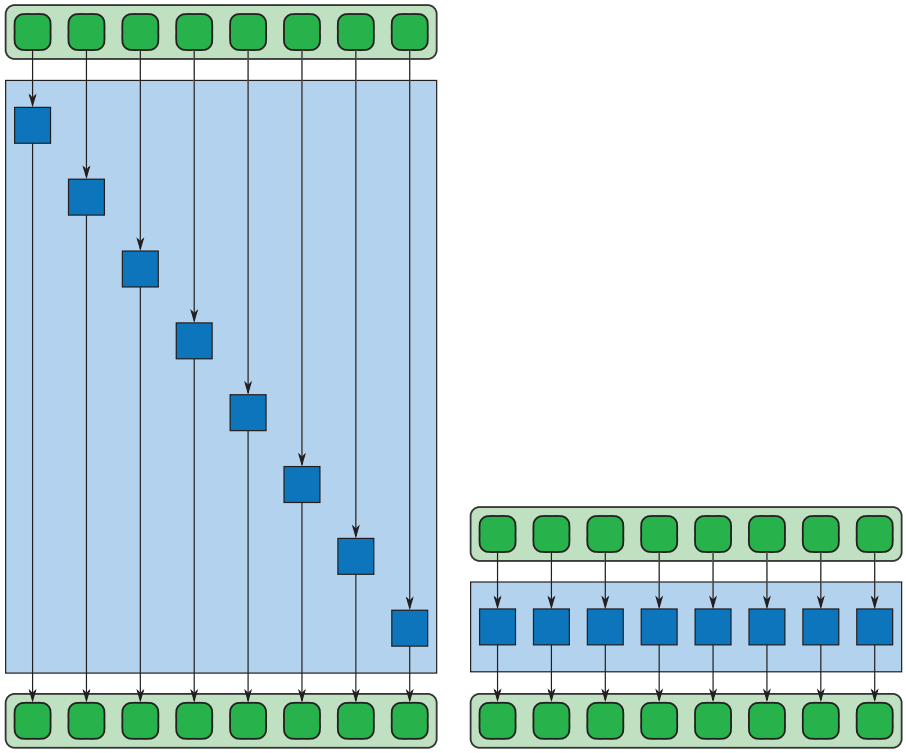

FIGURE 4.1

Serial and parallel execution of a map pattern. In the map pattern, an elemental function is applied to all elements of a collection, producing a new collection with the same shape as the input. In serial execution, a map is often implemented as a loop (with each instance of the loop body consisting of one invocation of the elemental function), but because there are no dependencies between loop iterations all instances can execute at the same time, given enough processors.

work they consume. Fortunately, the parallel programming models we use in this book include good implementations of the map pattern that take care of these details. For completeness we will discuss such implementation issues, but if you use a good programming model it will probably be unnecessary for you to worry about this yourself.

The map pattern is commonly used as the basis of **vectorization** as well as **parallelization**. To support full **nesting** of patterns, it is important to be able to support serial control flow and data access patterns inside a vectorized map. Again, these implementation details are (probably) not something you have to worry about yourself, if you are using a programming model that supports them, but the emulation of control flow in a vectorized map does have some performance implications.

In the following chapters, we will discuss some additional optimizations that are possible when combinations of map with other parallel patterns are considered. One particular combination of patterns, Map-Reduce, is at the heart of the success of Internet giant Google [DG04, Kon11]. Map-Reduce

is used for thousands of computations on large distributed systems daily. This is a testament to the power of parallelism to process enormous quantities of data efficiently. Even after discussing only a small number of patterns it will be possible to understand the basis of such a system.

This is the first chapter where we will show code samples. We give examples demonstrating the map pattern in various programming models, including TBB, Cilk Plus, OpenMP, ArBB, and OpenCL. Unlike the cases in the later chapters, these code samples are not meant to showcase efficient implementations. Instead, these samples have been intentionally simplified to show the map pattern in its purest form, so they can be used as templates for building up more complex programs. Also, it should be noted that code samples in this book do not necessarily show all the wrapper code needed for a full application. However, full application source code, including all of the necessary wrapper code for each of the examples in this book, is available online. We also do not provide full documentation for each of the programming models used in this book, since such documentation is available online. However, for a summary of the features of the primary programming models used in this book and pointers to further reading, please see the appendices.

4.1 MAP

In the comment pattern a function, which we will call an **elemental function**, is replicated and applied to different data. We can think of either applying the elemental function to each element of a collection of data or applying it to a set of indices that are then used to access different data for each of many instances of the function. The second approach is often used in some systems because map operations frequently access several sources of data and using a common set of indices is a convenient way to do it. We will call each parallel invocation of the elemental function on a different set of data, or portion of the index space, an **instance** of the elemental function.

The map pattern is closely associated with the **SIMD** model if there is no control flow in the function or the **SPMD** model if there is control flow in the function. The map pattern is also used with the **SIMT** model, which is just the SPMD model simulated on tiled SIMD hardware. These names were explained in Section 2.4.3. They do not refer to the map pattern itself, but to specific mechanisms for implementing it.

Instead of a single function, the map pattern may also be expressed indirectly as a sequence of vector operations. We will show in Section 4.4 that this form is semantically equivalent to the standard form using an elemental function. Since grouping operations is usually more efficient, implementations may try to convert sequences of vector operations into elemental functions automatically.

The map pattern corresponds to a parallelization of the serial iteration pattern in the special case that all iterations are independent. Therefore, the map pattern is often expressed in programming models using a "parallel for" construct. This is also equivalent to the elemental function form. In the case of the "parallel for," the loop body is the elemental function and the index variable in the parallel for construct generates an index space. Such "parallel for" constructs, being just alternative syntaxes for the map pattern, are significantly more restricted than regular `for` loops.

The map pattern assumes elemental functions have certain properties. In particular, elemental functions should not have side effects. Instances of elemental functions may read from other data in memory, as long as that data is not being modified in parallel. When we address data reorganization

patterns in Chapter 6, we discuss the **gather** pattern. A gather is just a set of random reads inside a map. It is also possible to combine random write with a map, giving the **scatter** pattern, although this can cause **non-determinism** and **race conditions.** Therefore, we assume in a "pure" map that random reads from memory are permitted but not random writes. Instead of writing to random locations in memory, each element of a "pure" map can output a fixed number of results.

The map pattern is **deterministic** if side effects and hence interdependencies between elemental function instances are avoided. In the correct use of map, the outcome of the map does not depend on the order in which the various instances of the elemental function are executed.

As noted, the parameters to the instances of an elemental function can be either data itself or an index that is then used to access data using a random memory read. In the case that data itself is the input to a map, there are two kinds of arguments: data that is different for each instance of the map, and data that is the same. At least some of the data fed into each instance of a map should be different, of course; otherwise there would be no point in running multiple instances. However, it is frequently useful to "broadcast" to all instances of the map some common data that is the same in all instances. We will call data that is different for each instance of the map **varying** data, while data that is broadcast to each instance and is the same for every instance we will call **uniform** data.

4.2 SCALED VECTOR ADDITION (SAXPY)

We begin our discussion of map with a simple example: scaled vector addition of single-precision floating point data. This operation, called SAXPY, is an important primitive in many linear algebra operations.

We emphasize that, since SAXPY does very little work relative to the amount of data it produces and consumes, its scalability is limited. However, this example is useful for introducing map as well as the concepts of **uniform** and **varying** parameters, and our code samples show how these concepts are expressed in different parallel programming models.

4.2.1 Description of the Problem

The SAXPY operation scales a vector **x** by scalar value a and adds it to vector **y**, elementwise. Both vectors have length n. This operation shows up frequently in linear algebra operations, such as for row cancellation in Gaussian elimination. The name SAXPY is from the industry standard **BLAS** (Basic Linear Algebra Subprograms) library for the single-precision version of this operation. Double precision is DAXPY, complex is CAXPY, and double complex is ZAXPY.

The mathematical definition of SAXPY is:

$$\mathbf{y} \leftarrow a\mathbf{x} + \mathbf{y},$$

where vector **x** is used as an input and vector **y** is used for both input and output; that is, the old value of **y** is destroyed. Overwriting an input like this is not a fundamental mathematical requirement but is how it is defined in BLAS because it is the common use case. For our purposes, it lets us show how to use the same variable for both input and output in different programming models.

Alternatively, SAXPY operation can be described as a function acting over individual elements, and applying this function to every element of the input data. Suppose the ith element of $\mathbf{x}$ is x_i and the ith element of $\mathbf{y}$ is y_i. Then we can define

$$f(t,p,q) = tp + q,$$
$$\forall i: \; y_i \leftarrow f(a,x_i,y_i).$$

Function f is an example of an **elemental function** (Section 1.5.3). The variables t, p, and q are used here in the definition of the elemental function to emphasize that these are formal arguments, to be bound to individual elements of the input and output collections. The map pattern invokes the elemental function as many times as there are elements in its input. We call each such invocation an *instance* of the map.

As discussed in the introduction, elemental functions have two kinds of arguments. There are arguments like a that are the same in every invocation of the function, and those like x_i and y_i that are different for every invocation. Parameters like a will be called **uniform** parameters, since they are the same (uniform) in every invocation of the function. Those like x_i and y_i will be called **varying** parameters.

Because the **arithmetic intensity** of SAXPY is low, it probably will not scale very well. This operation, called SAXPY in the single-precision case, is a Level 1 BLAS routine. Level 2 BLAS perform matrix–vector operations, and Level 3 BLAS perform matrix–matrix operations. Higher level BLAS routines offer more opportunity for parallelism and hence scale better. There is simply not enough work in each unit of parallelism for most Level 1 BLAS routines relative to the cost of managing the parallelism and accessing memory. For more complex operations that do more work, however, the map pattern can scale very well, since there is no communication and synchronization only occurs at the end of the map.

Although the SAXPY example is simple, it does give us an opportunity to talk about several key concepts, and the examples can be used as a template for more complex applications of the map pattern.

We will demonstrate how to code the SAXPY example in both TBB and Cilk Plus, as well as OpenMP, ArBB, and OpenCL. In some cases, we will give multiple versions if there is more than one way to code the algorithm. The TBB version is explicitly **tiled** for better performance. We provide two Cilk Plus versions: one written using a parallel `cilk_for` and another using Cilk array notation.

Both TBB and Cilk Plus support the map pattern by a "parallel for" construct. Additionally, in Cilk Plus you can express the same operation using expressions over array sections. However, to start off with, we will show a serial implementation in order to provide a baseline.

4.2.2 Serial Implementation

As a basis for comparison, a serial implementation of the SAXPY operation is shown in Listing 4.1. The main algorithm is expressed as a loop that visits each element of the input and output arrays in turn and performs an operation on each element. Note that all the loop iterations are independent, which is what makes this algorithm a candidate for parallel implementation with the map pattern.

4.2.3 TBB

Listing 4.2 gives a TBB implementation of SAXPY. This implementation uses a **lambda function**, a feature introduced by the C++11 standard and widely available in new C++ compilers. We use lambda

```
void saxpy_serial(
    size_t n,          // the number of elements in the vectors
    float a,           // scale factor
    const float x[],   // the first input vector
    float y[]          // the output vector and second input vector
) {
    for (size_t i = 0; i < n; ++i)
        y[i] = a * x[i] + y[i];
}
```

LISTING 4.1

Serial implementation of SAXPY in C.

```
void saxpy_tbb(
    int n,       // the number of elements in the vectors
    float a,     // scale factor
    float x[],   // the first input vector
    float y[]    // the output vector and second input vector
) {
    tbb::parallel_for(
        tbb::blocked_range<int>(0, n),
        [&](tbb::blocked_range<int> r) {
            for (size_t i = r.begin(); i != r.end(); ++i)
                y[i] = a * x[i] + y[i];
        }
    );
}
```

LISTING 4.2

Tiled implementation of SAXPY in TBB. Tiling not only leads to better **spatial locality** but also exposes opportunities for vectorization by the host compiler.

functions for brevity throughout the book, though they are not required for using TBB. Appendix D.2 discusses lambda functions and how to write the equivalent code by hand if you need to use an old C++ compiler.

The TBB code exploits **tiling**. The `parallel_for` breaks the half-open range $[0,n)$ into subranges and processes each subrange `r` with a separate task. Hence, each subrange `r` acts as a tile, which is processed by the serial `for` loop in the code. Here the range and subrange are implemented as `blocked_range` objects. Appendix C.3 says more about the mechanics of `parallel_for`.

TBB uses thread parallelism but does not, by itself, vectorize the code. It depends on the underlying C++ compiler to do that. On the other hand, tiling does expose opportunities for vectorization, so if the basic serial algorithm can be vectorized then typically the TBB code can be, too. Generally, the

performance of the serial code inside TBB tasks will depend on the performance of the code generated by the C++ compiler with which it is used.

4.2.4 Cilk Plus

A basic Cilk Plus implementation of the SAXPY operation is given in Listing 4.3. The "parallel for" syntax approach is used here, as with TBB, although the syntax is closer to a regular `for` loop. In fact, an ordinary `for` loop can often be converted to a `cilk_for` construct if all iterations of the loop body are independent—that is, if it is a map. As with TBB, the `cilk_for` is not explicitly vectorized but the compiler may attempt to auto-vectorize. There are restrictions on the form of a `cilk_for` loop. See Appendix B.5 for details.

4.2.5 Cilk Plus with Array Notation

It is also possible in Cilk Plus to explicitly specify vector operations using Cilk Plus array notation, as in Listing 4.4. Here `x[0:n]` and `y[0:n]` refer to `n` consecutive elements of each array, starting with `x[0]` and `y[0]`. A variant syntax allows specification of a stride between elements, using `x[start:length:stride]`. Sections of the same length can be combined with operators. Note that there is no `cilk_for` in Listing 4.4.

```
1  void saxpy_cilk(
2      int n,       // the number of elements in the vectors
3      float a,     // scale factor
4      float x[],   // the first input vector
5      float y[]    // the output vector and second input vector
6  ) {
7      cilk_for (int i = 0; i < n; ++i)
8          y[i] = a * x[i] + y[i];
9  }
```

LISTING 4.3

SAXPY in Cilk Plus using `cilk_for`.

```
1  void saxpy_array_notation(
2      int n,       // the number of elements in the vectors
3      float a,     // scale factor
4      float x[],   // the input vector
5      float y[]    // the output vector and offset
6  ) {
7      y[0:n] = a * x[0:n] + y[0:n];
8  }
```

LISTING 4.4

SAXPY in Cilk Plus using `cilk_for` and array notation for explicit vectorization.

Uniform inputs are handled by **scalar promotion:** When a scalar and an array are combined with an operator, the scalar is conceptually "promoted" to an array of the same length by replication.

4.2.6 OpenMP

Like TBB and Cilk Plus, the map pattern is expressed in OpenMP using a "parallel for" construct. This is done by adding a `pragma` as in Listing 4.5 just before the loop to be parallelized. OpenMP uses a "team" of threads and the work of the loop is distributed over the team when such a pragma is used. How exactly the distribution of work is done is given by the current scheduling option.

The advantage of the OpenMP syntax is that the code inside the loop does not change, and the annotations can usually be safely ignored and a correct serial program will result. However, as with the equivalent Cilk Plus construct, the form of the `for` loop is more restricted than in the serial case. Also, as with Cilk Plus and TBB, implementations of OpenMP generally do not check for incorrect parallelizations that can arise from dependencies between loop iterations, which can lead to race conditions. If these exist and are not correctly accounted for in the pragma, an incorrect parallelization will result.

4.2.7 ArBB Using Vector Operations

ArBB operates only over data stored in ArBB containers and requires using ArBB types to represent elements of those containers. The ArBB `dense` container represents multidimensional arrays. It is a template with the first argument being the element type and the second the dimensionality. The dimensionality default is 1 so the second template argument can be omitted for 1D arrays.

The simplest way to implement SAXPY in ArBB is to use arithmetic operations directly over `dense` containers, as in Listing 4.6. Actually, this gives a **sequence** of **maps**. However, as will be explained in Section 4.4, ArBB automatically optimizes this into a **map** of a **sequence**.

In ArBB, we have to include some extra code to move data into "ArBB data space" and to invoke the above function. Moving data into ArBB space is required for two reasons: **safety** and **offload**. Data stored in ArBB containers can be managed in such a way that race conditions are avoided. For example, if the same container is both an input and an output to a function, ArBB will make sure that

```
1  void saxpy_openmp(
2      int n,      // the number of elements in the vectors
3      float a,    // scale factor
4      float x[],  // the first input vector
5      float y[]   // the output vector and second input vector
6  ) {
7  #pragma omp parallel for
8      for (int i = 0; i < n; ++i)
9          y[i] = a * x[i] + y[i];
10 }
```

LISTING 4.5

SAXPY in OpenMP.

```
void saxpy_call_arbb(
    f32 t,          // uniform input
    dense<f32> p,   // varying input
    dense<f32>& q   // uniform input and also output
) {
    q = t * p + q;
}
```

LISTING 4.6

SAXPY in ArBB, using a vector expression. One way the map pattern can be expressed in ArBB is by using a sequence of vector operations over entire collections.

```
void saxpy_arbb(
    size_t n,       // number of elements
    float a,        // uniform input
    const float x[], // varying input
    float y[]       // varying input and also output
) {
    f32 aa = a; // copy scalar to ArBB type
    dense<f32> xx(n), yy(n); // ArBB storage for arrays
    memcpy(&xx.write_only_range()[0], x, sizeof(float)*n);
    memcpy(&yy.write_only_range()[0], y, sizeof(float)*n);
    call(saxpy_call_arbb)(aa, xx, yy);
    memcpy(y, &yy.read_only_range()[0], sizeof(float)*n);
}
```

LISTING 4.7

SAXPY in ArBB, using binding code for vector expression implementation. This code is necessary to move data in and out of ArBB data space.

the "alias" does not cause problems with the parallelization. Second, data stored in ArBB containers may in fact be maintained in a remote memory, such as on an **attached co-processor**, rather than in the host memory. Keeping data in ArBB containers for a sequence of operations allows ArBB to avoid copying data back to the host unnecessarily.

Listing 4.7 shows the necessary code to move data into ArBB space, to invoke the function given in Listing 4.6, and to move the result back out of ArBB space.

4.2.8 ArBB Using Elemental Functions

It is also possible to specify an **elemental function** for the map pattern directly in ArBB. Replicas of this function can then be applied in parallel to all elements of an ArBB collection using a `map` operation. The `map` operation can only be invoked from inside an ArBB `call`, so we need to define another function for the call. The call function, however, can have an entire sequence of map

```
1  void saxpy_map_arbb(
2     f32 t, // input
3     f32 p, // input
4     f32& q // input and output
5  ) {
6     q = t * p + q;
7  }
```

LISTING 4.8

SAXPY in ArBB, using an elemental function. The `map` pattern can also be expressed in ArBB using an elemental function called through a `map` operation.

```
1  void saxpy_call2_arbb(
2     f32 a,          // uniform input
3     dense<f32> x,   // varying input
4     dense<f32>& y   // varying input and also output
5  ) {
6     map(saxpy_map_arbb)(a,x,y);
7  }
```

LISTING 4.9

SAXPY in ArBB, `call` operation. A `map` operation in ArBB can only be invoked from inside an ArBB context, so we have to use `call` first to open an ArBB context.

operations. It can also include vector operations and control flow, although we will not show that in this example. Listing 4.8 shows the definition of the elemental function for SAXPY, and Listing 4.9 shows the necessary call function. The binding code is identical to the previous example except for a change in the call function name.

When we define the elemental function used for the map in ArBB we do not have to decide at the point of definition of the function which parameters are uniform and which are varying. In ArBB, elemental functions are polymorphic and can be invoked with each parameter either being a scalar or being a collection. All the collections do have to be the same shape (dimensionality and number of elements), however.

4.2.9 OpenCL

Listing 4.10 gives kernel code for an OpenCL implementation of SAXPY. **Kernels** are equivalent to what we have been calling elemental functions, except that in OpenCL they always operate on the device and are given in a separate "kernel language" which is a superset (and a subset) of C99. Three OpenCL-specific keywords are used in this example: `__kernel`, `__global`, `__constant`. The `__kernel` keyword simply identifies a particular function as being invoked as an elemental

```
__kernel void
saxpy_opencl(
    __constant float a,
    __global float* x,
    __global float* y
) {
    int i = get_global_id(0);
    y[i] = a * x[i] + y[i];
}
```

LISTING 4.10

SAXPY in OpenCL kernel language.

function/kernel. The OpenCL programming model also includes multiple memory spaces, and `__global` and `__constant` identify the use of those spaces. In PCIe-based coprocessor implementations of OpenCL devices, global data is stored in the device's off-chip DRAM, while constant data is stored in a small on-chip memory. Access to data elements in the arrays is done explicitly with ordinary array indexing operations. In other programming models supporting elemental functions, such as ArBB, this indexing is handled by the system. In OpenCL the addresses are computed directly from the global ID, which is an element identifier for each instance of the kernel, with instances numbered starting from zero.

The host code (not shown, but available online) takes care of transferring the data to the device, invoking the computation, and transferring the data back. Since SAXPY is such a simple computation, offloading it alone to a co-processor will not be performant. More likely, the SAXPY kernel will be used as part of a larger computation. OpenCL provides a way to queue up a number of kernels to be executed in sequence to make this mode of operation more efficient.

4.3 MANDELBROT

The computation of the Mandelbrot set is a simple example that shows how the map pattern can include serial control flow and how elemental functions can be used to express this. It is also a good example of the kind of calculation that can lead to a **load imbalance**.

4.3.1 Description of the Problem

The Mandelbrot set is the set of all points c in the complex plane that do *not* go to infinity when the quadratic function $z \leftarrow z^2 + c$ is iterated. In practice, it is hard to prove that this recurrence will never diverge so we iterate up to some maximum number of times. We can also prove that once z leaves a circle of radius 2 it will be guaranteed to diverge. If this happens, we can terminate the computation early. In practice, we will compute the following function, up to some maximum value of K. We can

then use a lookup table to map different counts to colors to generate an image.

$$z_0 = 0,$$

$$z_{k-1} = z_k^2 + c,$$

$$\text{count}(c) = \min_{0 \le k < K} (|z_k| \ge 2).$$

Computing the Mandelbrot set has little practical value. However, we are including it here because, while it can be implemented using the map pattern, it includes data-dependent control flow. This leads to a load imbalance: Different pixels in the computation can take different numbers of iterations to diverge. In fact, different regions of the complex plane will have different behaviors, because some regions are smooth while other regions require very different number of iterations for nearby pixels.

In other words, the SAXPY example in Section 4.2 could be implemented efficiently using SIMD mechanisms, but the Mandelbrot example is best implemented using SPMD or tiled SIMD mechanisms, including load balancing and early termination of finished tiles.

4.3.2 Serial Implementation

We provide a serial implementation of the Mandelbrot computation in Listing 4.11. We need to use complex numbers, and there are two options: the C99 `Complex`, and the C++ `std::complex`. In this section, we use `Complex` for the serial version, Cilk Plus, and TBB but will switch to `std::complex` for ArBB. Note that in this listing we use separate variables for the iteration index and the count. In some of the parallel versions we can remove this redundancy. We also break out the body of the Mandelbrot computation as a separate function, since it is this function that we will convert to an elemental function in the map pattern.

4.3.3 TBB

The TBB implementation of the Mandelbrot example follows exactly the same template as the example in Section 4.2.3. We can invoke the elemental function for each element in a block given by the `blocked_range` argument to the lambda function as shown in Listing 4.12.

4.3.4 Cilk Plus

Listing 4.13 gives a Cilk Plus implementation of the Mandelbrot example, using the `cilk_for` construct. Note that only the outer loop is parallelized. We could parallelize both loops but in this case parallelizing over only the rows will probably be sufficient, and leaving the inner loop serial will reduce the task management overhead. In addition, in the case of Mandelbrot the execution times for rows are more uniform than the execution times for pixels, making load balancing easier. However, there certainly might be applications that use two nested loops where we would want to parallelize both.

```
int mandel(
   Complex c,
   int depth
) {
   int count = 0;
   Complex z = 0;
   for (int k = 0; k < depth; k++) {
      if (abs(z) >= 2.0) {
         break;
      }
      z = z*z + c;
      count++;
   }
   return count;
}

void serial_mandel(
   int p[][],
   int max_row,
   int max_col,
   int depth
) {
for (int i = 0; i < max_row; ++i) {
   for (int j = 0; j < max_col; ++j)
      p[i][j] = mandel(Complex(scale(i), scale(j)),
                  depth);
}
```

LISTING 4.11

Serial implementation of Mandelbrot in C.

```
parallel_for( blocked_range<int>(0, max_row),
   [&](blocked_range<int> r) {
      for (size_t i = r.begin(); i != r.end(); ++i)
         for (int j = 0; j < max_col; ++j)
            p[i][j] =
               mandel(Complex(scale(i), scale(j)), depth);
   }
);
```

LISTING 4.12

Tiled implementation of Mandelbrot in TBB.

```
1  cilk_for (int i = 0; i < max_row; ++i)
2     for (int j = 0; j < max_col; ++j)
3        p[i][j] =
4           mandel(Complex(scale(i), scale(j)), depth);
```

LISTING 4.13

Mandelbrot using `cilk_for` in Cilk Plus.

4.3.5 Cilk Plus with Array Notations

The Mandelbrot set computation can also be implemented using Cilk Plus array notation. An implementation is shown in Listing 4.14. This actually combines thread parallelism over rows invoked with `cilk_for` with vector parallelism within each row, invoked with array notation. Within each row, we break the work up into chunks of 8 pixels.[1] Then, within each chunk, we invoke the `mandel` function. Within the `mandel` function, we now use an explicit SIMD implementation over the entire chunk, using vector operations. The `__sec_reduce_add` function computes the sum of all the elements in an array section—in this case, the results of the test. This is actually an instance of the reduction pattern, covered in detail in the next chapter. Note that the `break` will only be taken when *all* the pixels in a chunk have diverged. This implementation will therefore do more work than necessary, since all pixels in the chunk will have to be updated if even one needs to continue to iterate. However, if the pixels in a chunk have spatially coherent termination counts, this is often more efficient than serially computing each pixel.

4.3.6 OpenMP

Listing 4.15 shows the OpenMP parallelization of the Mandelbrot example. As with SAXPY in Section 4.2, in this case we are able to perform the parallelization with the addition of a single annotation. However, here we add a `collapse` attribute to the *pragma* annotation to indicate that we want to parallelize both loops at once. This allows OpenMP to parallelize the computation over the combined iteration space. This gives OpenMP more potential parallelism to work with. On the other hand, for systems with relatively small core counts, parallelizing over just the rows might be sufficient and might even have higher performance, as we have argued for the Cilk Plus implementation. If this is desired, the `collapse` clause can be omitted. To get the effect equivalent to the `collapse` clause in Cilk Plus, we would simply nest `cilk_for` constructs.

4.3.7 ArBB

For the ArBB version of Mandelbrot, we will switch to using `std::complex` and also specify the region of interest by giving two points in the complex plane. The implementation is given in Listings 4.16, 4.17, and 4.18. Listing 4.16 gives the elemental function, Listing 4.17 gives the call

[1] For simplicity, we do not show the extra code to handle partial chunks when the row length is not a multiple of 8.

```
void cilkplus_an_mandel(
    int n,
    std::complex c[n],
    int count[n],
    int max_count
) {
    std::complex z[n];
    int test[n];
    z[:] = 0;
    for (int k = 0; k < max_count; k++) {
        // test for divergence for all pixels in chunk
        test[:] = (abs(z[:] < 2.0);
        if (0 == __sec_reduce_add(test[:])) {
            // terminates loop only if all have diverged
            break;
        }
        // increment counts only for pixels that have not diverged
        count[:] += test[:];
        // unconditionally update state of iteration
        z[:] = z[:]*z[:] + c[:];
    }
}

void cilkplus_mandel(
    int p[][],
    int max_row,
    int max_col,
    int depth
) {
    // parallelize over rows
    cilk_for (int i = 0; i < max_row; ++i)
        // loop over the row in chunks of 8
        for (int j = 0; j < max_col; j += 8)
            // compute the Mandelbrot counts for a chunk
            cilkplus_an_mandel(8, p[i]+j, points[i]+j, depth);
}
```

LISTING 4.14

Mandelbrot in Cilk Plus using `cilk_for` and array notation for explicit *vectorization*.

function (which also does a little bit of setup for the map), and Listing 4.18 invokes the call and synchronizes the result with the host.

The overall organization of the Mandelbrot code is similar to the elemental function version of SAXPY. However, control flow that depends on values computed by ArBB needs to be expressed in a special way. This is because ArBB is really an API for expressing computation at runtime and will

```
#pragma omp parallel for collapse(2)
   for (int i = 0; i < max_row; i++)
      for (int j = 0; j < max_col; j++)
         p[i][j] = mandel(Complex(scale(i), scale(j)),
                          depth);
```

LISTING 4.15

Mandelbrot in OpenMP.

```
void arbb_mandelbrot_map(
   f64 x0, f64 y0, // lower left corner of region
   f64 dx, f64 dy, // step size
   i32 depth,    // maximum number of iterations
   i32& output   // output: escape count
) {
   i32 i = 0;
   // obtain stream index and cast from usize to f64
   const array<f64, 2> pos = position<2>().as<f64>();
   // use index to compute position of sample in the complex plane
   const std::complex<f64> c(x0 + pos[0] * dx,
                             y0 + pos[1] * dy);
   std::complex<f64> z = c;
   // if the loop reaches depth
   // assume c is an element of the Mandelbrot set
   _while (i < depth) {
      _if (norm(z) > 4.0) {
         _break; // escaped from a circle of radius 2
      } _end_if;
      z = z * z + c; // Mandelbrot recurrence
      ++i;
   } _end_while;
   // record the escape count
   output = i;
}
```

LISTING 4.16

Mandelbrot elemental function for ArBB `map` operation.

compile computations specified using this API to machine language. We have to differentiate between control flow used in the generation of the code from control flow meant to be included in the generated code. The `_for`, `_if`, etc. keywords are there to tell ArBB to insert data-dependent control flow into the code it generates. Other than this change in syntax, the logic of the ArBB elemental function is quite similar to that of the original serial version. However, the internal implementation will, in fact, be similar to that generated by the version given in Listing 4.14. ArBB will automatically block the

```
void arbb_mandelbrot_call(
    f64 x0, f64 y0,         // lower left corner of region
    f64 x1, f64 y1,         // upper right corner of region
    i32 depth,              // maximum number of iterations
    dense<i32, 2>& output // output image (scaled escape count)
) {
    usize width = output.num_cols();
    usize height = output.num_rows();
    // step size for width by height equally spaced samples
    f64 dx = (x1 - x0) / f64(width);
    f64 dy = (y1 - y0) / f64(height);
    // apply the map
    map(arbb_mandelbrot_map)(x0, y0, dx, dy,
                        depth, output);
}
```

LISTING 4.17

Mandelbrot call code for ArBB implementation. This code computes the pixel spacing and then maps the elemental function to compute the escape count for each pixel.

```
void arbb_mandelbrot(
    double x0, double y0,
    double x1, double y1,
    int depth,
    int width, int height,
    int* result
) {
    // allocate buffer for result
    dense<i32,2> output(width, height);
    // compute the Mandelbrot set
    call(arbb_mandelbrot_call)(f64(x0), f64(y0),
                         f64(x1), f64(y1),
                         i32(depth), output);
    // synchronize and read back output
    memcpy(result, &output.read_only_range()[0],
         width * height * sizeof(int));
}
```

LISTING 4.18

Mandelbrot binding code for ArBB implementation. This code performs the call, then synchronizes the output with the host array.

work and emulate control flow using SIMD operations over chunks. It will also include any extra code needed to handle data misalignments at the edges of the arrays, for example, due to rows that are not a multiple of the hardware vector size.

4.3.8 OpenCL

An OpenCL implementation of the Mandelbrot computation is given in Listing 4.19. In this code, the necessary complex number operations are implemented manually. This kernel also includes an optimization that we could have used in the other implementations: We test the square of the magnitude of z for divergence, rather than the actual magnitude. This avoids a square root. Only the kernel code is shown here, although a complete application also requires host code using the OpenCL API to set up data, invoke the kernel, and read the results back.

The Mandelbrot computation allows for fine-grained 2D parallelization that is appropriate for the device's OpenCL targets. We do this here with a 2D kernel. Inside the kernel we can access the index from the appropriate dimension using the argument to `get_global_id` to select the argument. This

```
int mandel(
   float cx, float cy,
   int depth
) {
   int count = 0;
   float zx = cx;
   float zy = cy;
   while (count < depth) {
      if (zx*zx + zy*zy > 4.0)
         break;
      float zsqx = zx*zx - zy*zy;
      float zsqy = 2*zx*zy;
      zx = zsqx + cx;
      zy = zsqy + cy;
      count++;
   }
   return count;
}

__kernel void
do_mandel(
   __global int* p,
   float x0, float y0, float dx, float dy
) {
   int i = get_global_id(0);
   int j = get_global_id(1);
   float cx = x0 + i * dx;
   float cy = y0 + j * dy;
   int count = mandel(cx, cy, max_count);
   p[j*width+i] = count;
}
```

LISTING 4.19

Mandelbrot kernel code for OpenCL implementation.

particular interface (using a numerical value to select the index component desired) was chosen because it provides a straightforward extension to higher dimensionalities.

4.4 SEQUENCE OF MAPS VERSUS MAP OF SEQUENCE

A **sequence of map** operations over collections of the same shape should be combined whenever possible into a single larger operation. In particular, vector operations are really map operations using very simple operations like addition and multiplication. Implementing these one by one, writing to and from memory, would be inefficient, since it would have low arithmetic intensity. If this organization was implemented literally, data would have to be read and written for each operation, and we would consume memory bandwidth unnecessarily for intermediate results. Even worse, if the maps were big enough, we might exceed the size of the cache and so each map operation would go directly to and from main memory.

If we fuse the operations used in a sequence of maps into a sequence inside a single map, we can load only the input data at the start of the map and keep intermediate results in registers rather than wasting memory bandwidth on them. We will call this approach **code fusion**, and it can be applied to other patterns as well. Code fusion is demonstrated in Figure 4.2.

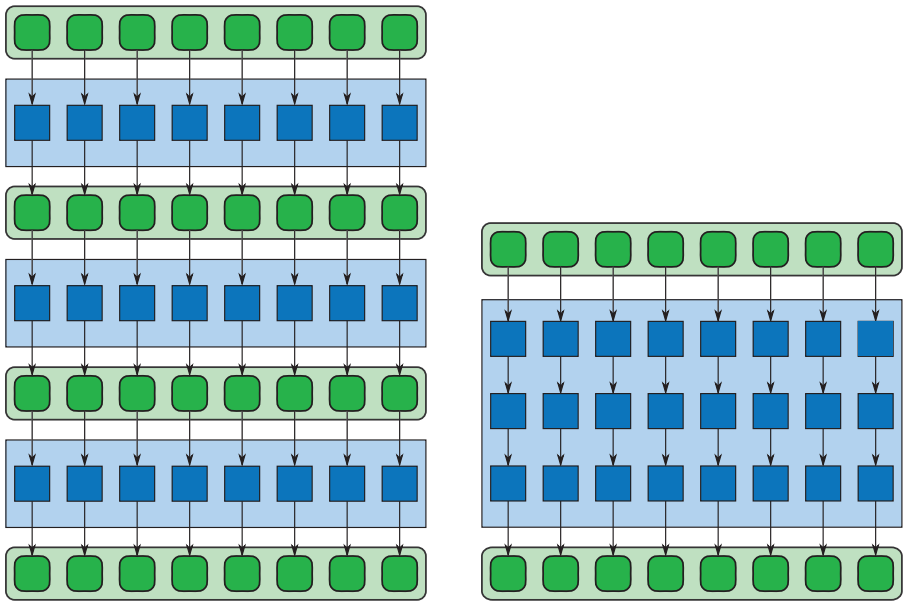

FIGURE 4.2

Code fusion optimization: Convert a sequence of maps into a map of sequences, avoiding the need to write intermediate results to memory. This can be done automatically by ArBB and explicitly in other programming models.

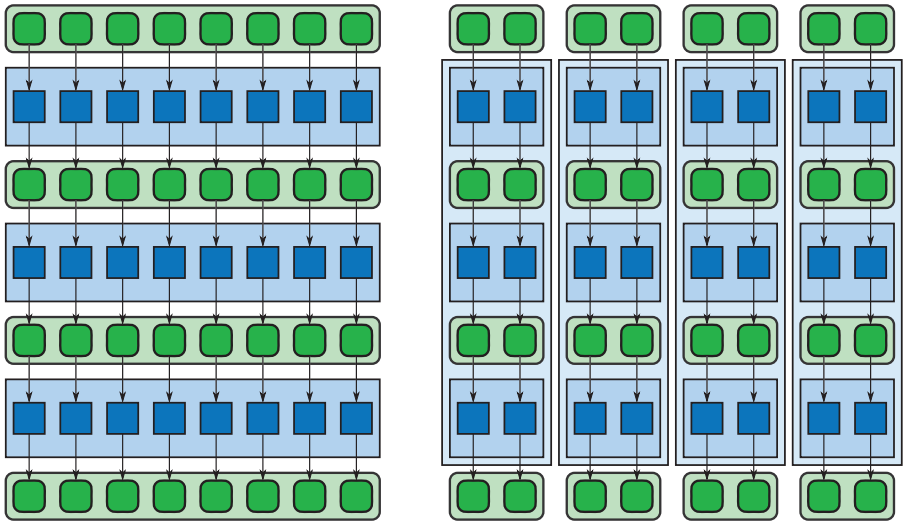

FIGURE 4.3

Cache fusion optimization: Process sequences of maps in small tiles sequentially. When code fusion is not possible, a sequence of maps can be broken into small tiles and each tile processed sequentially. This avoids the need for synchronization between each individual map, and, if the tiles are small enough, intermediate data can be held in cache.

Another approach that is often almost as effective as code fusion is **cache fusion**, shown in Figure 4.3. If the maps are broken into tiles and the entire sequence of smaller maps for one tile is executed sequentially on one core, then if the aggregate size of the tiles is small enough intermediate data will be resident in cache. In this case at least it will be possible to avoid going to main memory.

Both kinds of fusion also reduce the cost of synchronization, since when multiple maps are fused only one synchronization is needed after all the tiles are processed, instead of after every map. However, code fusion is preferred when it is possible since registers are still faster than cache, and with cache fusion there is still the "interpreter" overhead of managing the multiple passes. However, cache fusion is useful when there is no access to the code inside the individual maps—for example, if they are provided as precompiled user-defined functions without source access by the compiler. This is a common pattern in, for example, image processing plugins.

In Cilk Plus, TBB, OpenMP, and OpenCL the reorganization needed for either kind of fusion must generally be done by the programmer, with the following notable exceptions:

OpenMP: Cache fusion occurs when *all* of the following are true:

- A single parallel region executes all of the maps to be fused.
- The loop for each map has the same bounds and chunk size.
- Each loop uses the `static` scheduling mode, either as implied by the environment or explicitly specified.

TBB: Cache fusion can be achieved using `affinity_partitioner`, as explained in Appendix 3.2.
Cilk Plus: Sequences of vector operations expressed using array notation will generally be code-fused into efficient elemental functions.

In ArBB, not only will vector operations be fused together whenever possible, but ArBB will also code-fuse sequences of map operations using different elemental functions. This reorganization used by code fusion can be seen as an application of associativity between map and sequence. This is one of many possible high-level optimizations achieved by algebraically manipulating patterns. We will discuss other such optimizations in later chapters.

4.5 COMPARISON OF PARALLEL MODELS

As we have seen, Cilk Plus, TBB, and OpenMP use parallel `for` loop constructs to implement the map pattern, ArBB uses either vector operations or elemental functions, and OpenCL uses elemental functions. Both TBB and Cilk Plus can also use elemental functions (in fact, the TBB syntax is really implemented this way), but the need for separate declaration of these functions can be avoided through use of lambda functions in C++. Cilk Plus also supports the map pattern through sequences of vector operations. In ArBB and Cilk Plus, the fact that sequences of vector operations are automatically fused together is important for performance.

4.6 RELATED PATTERNS

There are several patterns related to map. We discuss three of them here: **stencil, workpile,** and **divide-and-conquer**. Stencil in particular is extremely common in practice. Chapter 7 discusses the stencil pattern in more detail, and a detailed example is given in Chapter 10. Divide-and-conquer is the basis of many recursive algorithms that can in turn be parallelized using the **fork–join** pattern.

4.6.1 Stencil

The **stencil** pattern is a map, except each instance of the elemental function accesses neighbors of its input, offset from its usual input. A **convolution** uses the stencil pattern but combines elements linearly using a set of weights. Convolutions are common, but generally the computation performed on the set of neighbors gathered in the stencil pattern need not be linear. Many forms of non-linear stencil exist—for example, the median filter for reducing impulse noise in images.

A stencil is still a map since the operations do not depend on each other. All that has been done is generalize the way that input is read. However, the stencil pattern is worth calling out for two reasons: It is common in imaging and PDE solvers, and many machine-dependent optimizations are possible for it.

Efficient implementation of the stencil pattern seeks to take advantage of data reuse. Adjacent invocations of the elemental function tend to reuse some number of inputs. The number of elements reused depends on the exact set of neighbors specified by the stencil but generally it is beneficial to **tile** the input domain into subdomains and slide a "window" across each subdomain so that data can

be reused. This is complicated to implement well in practice, and the optimal shape of the window can be machine dependent, as well as being dependent on the stencil shape.

In Cilk Plus, TBB, and OpenMP the stencil pattern can be implemented using random access. The sliding window optimization can be implemented as part of an overall tiling strategy. These three systems would then depend on the cache and perhaps the hardware prefetcher to take advantage of the spatial and temporal locality of the stencil. In OpenCL, the overall organization is the same, but depending on the hardware it may be necessary to manage data in on-chip "shared" memory explicitly. In ArBB, stencils are specified declaratively: Neighbors of an input can be accessed using the `neighbor` function inside a map. This allows ArBB to implement sliding windows internally, using a strategy appropriate for the machine being targeted without complicating the user's code.

4.6.2 Workpile

In the **workpile** pattern is an extension of the map pattern in which work items can be added to the map while it is in progress, from inside elemental function instances. This allows work to grow and be consumed by the map. The workpile pattern terminates when no more work is available.

The workpile pattern is supported natively in TBB, but not presently in ArBB, OpenMP, OpenCL, or Cilk Plus. It could be implemented in OpenCL and OpenMP using explicit work queues. Its implementation in ArBB might be possible but would probably not be efficient enough to be useful at present. In Cilk Plus, the implementation would be straightforward in terms of fork–join and work stealing.

4.6.3 Divide-and-conquer

The **divide-and-conquer** pattern is related to the **partition** pattern discussed in Chapter 6. Basically, the divide-and-conquer pattern applies if a problem can be divided into smaller subproblems recursively until a base case is reached that can be solved serially. Divide-and-conquer can be implemented by combining the partition and map patterns: the problem is partitioned and then a map is applied to compute solutions to each subproblem in the partition.

Recursive divide-and-conquer is extremely natural in Cilk Plus and TBB since they use the **fork–join** pattern extensively, and this pattern is easy to implement with fork–join. The fork–join pattern is discussed in Chapter 8. In OpenMP recursive divide-and-conquer can be implemented using the tasking model. It is extremely difficult to implement recursive divide-and-conquer in OpenCL and ArBB since these do not at present support nested parallelism, although it could probably (with great difficulty and probably inefficiently) be emulated with work queues. However, non-recursive partitioning is the basis of many algorithms implementable in OpenMP, OpenCL, and ArBB. In fact, the partitioned memory model of OpenCL practically demands at least one level of partitioning for most problems.

Recursive divide-and-conquer is used to implement map itself in Cilk Plus and TBB, and therefore indirectly in ArBB, since the latter uses TBB for task management. When implementing a map, we do not want to try and create all tasks from the task invoking the map since that would place all the task creation overhead in the invoking task. Instead, we split the problem domain into a small number of partitions and then recursively subdivide in each resulting task as needed.

4.7 SUMMARY

This chapter has described the map pattern, which is the simplest parallel pattern. We have described some important optimizations of the map pattern, including the fusion of a sequence of maps into a map of sequences. In some cases, this optimization can be done automatically; in other cases, it must be done manually. We have also introduced some patterns closely related to map: **stencil**, **workpile**, and **divide-and-conquer** patterns.

Chapter 5 discusses **collective** operations, including **reduction** and **scan**, and Chapter 6 discusses data reorganization patterns. These two classes of patterns either are often combined with map or, in the case of data reorganization, result from the combination of specific serial data access patterns with map. The **stencil** and **recurrence** patterns are important generalizations of the map pattern and are discussed in Chapter 7. **Divide-and-conquer** is discussed in more detail in Chapter 8.

This page intentionally left blank