CHAPTER 2

Generic Parallel Algorithms

What is the best method for scheduling parallel loops? How do we process data structures in parallel that do not support random-access iterators? What's the best way to add parallelism to applications that look like pipelines? If the TBB library only provided tasks and a task scheduler, we would need to answer these questions ourselves. Luckily, we don't need to plow through the many master's theses and doctoral dissertations written on these topics. The TBB library developers have already done this dirty work for us! They provide the best-known methods for addressing these scenarios as template functions and template classes, a group of features known as the TBB generic parallel algorithms. These algorithms capture many of the processing patterns that are the cornerstones of multithreaded programming.

Note The TBB library developers have historically used the term generic parallel *algorithms* to describe this set of features. By *algorithm*, they do not mean a specific computation like matrix multiplication, LU decomposition, or even something like `std::find`, but instead they mean common execution patterns. It has been argued by some reviewers of this book that these features would therefore be more accurately referred to as *patterns* and not algorithms. However, to align with the terminology that the TBB library has been using for many years, we refer to these features as generic parallel *algorithms* in this book.

M. Voss, R. Asenjo, J. Reinders, *Pro TBB*, https://doi.org/10.1007/978-1-4842-4398-5_2

We should have a strong preference for using these prewritten algorithms, whenever they apply, instead of writing our own implementations. The developers of TBB have spent years testing and improving their performance! The set of algorithms included in the TBB library do not, of course, exhaustively cover every possible scenario, but if one of them does match our processing pattern, we should use it. The algorithms provided by TBB capture the majority of the scalable parallelism in applications. In Chapter 8, we discuss design patterns for parallel programming, such as those described in *Patterns for Parallel Programming by Mattson, Sanders and Massingill (Addison-Wesley)*, and how we can implement them using the TBB generic parallel algorithms.

As shown in Figure 2-1, all of the TBB generic algorithms start from a single thread of execution. When a thread encounters a parallel algorithm, it spreads the work associated with that algorithm across multiple threads. When all of the pieces of work are done, the execution merges back together and continues again on the initial single thread.

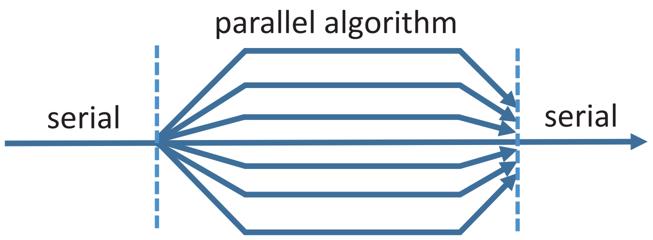


Figure 2-1. *The fork-join nature of the TBB parallel algorithms*

TBB algorithms provide a powerful but relatively easy parallel model to apply because they can often be added incrementally and with a fairly local view of the code under consideration. We can look for the most time-consuming region of a program, add a TBB parallel algorithm to speed up that region, then look for the next most time-consuming region, add parallelism there, and so on.

It must be understood however that TBB algorithms ***do not*** guarantee parallel execution! Instead, they only communicate to the library that parallel execution is allowed. If we look at Figure 2-1 from a TBB perspective, this means that all of the worker threads may participate in executing parts of the computation, only a subset of threads may participate, or just the master thread may participate. Programs and libraries that assume that the parallelism is optional, like TBB does, are referred to as having *relaxed sequential semantics*.

A parallel program has *sequential semantics* if executing it using only a single thread does not change the program's semantics. As we will note several times in this book though, the results of a sequential and parallel execution of a program may not always match exactly due to rounding issues and other sources of inexactness. We acknowledge these potential, nonsemantic differences by using the term *relaxed sequential semantics.* While models like TBB and the OpenMP API offer relaxed sequential semantics, some other models, such as MPI, let us write applications that have cyclic relationships that require parallel execution. The relaxed sequential semantics of TBB are an important part of what makes it useful for writing composable applications, as introduced in Chapter 1 and described in more detail in Chapter 9. For now, we should just remember that any of the algorithms described in this chapter will spread the work across one or more threads, but not necessarily all of the threads available in the system.

The set of algorithms available in the Threading Building Blocks 2019 distribution is shown in the table in Figure 2-2. They are all in namespace tbb and are available when the tbb.h header file is included. The basics of the **boldface** algorithms are covered in this chapter, with the other algorithms described in later chapters. We also provide a sidebar **Lambda expressions –vs- user-defined classes** that explains that while we almost exclusively use lambda expressions to pass code to the TBB algorithms in our examples in this book, these arguments can almost always be replaced by user-defined function objects if so desired.

Category	*Generic Algorithm*	*Brief Description*
Functional parallelism	**`parallel_invoke`**	Evaluates several functions in parallel.
Simple loops	**`parallel_for`**	Performs parallel iteration over a range of values.
	`parallel_reduce`	Computes a reduction over a range of values.
	`parallel_deterministic_reduce`	Computes a reduction over a range of values, with a deterministic split / join behavior.
	`parallel_scan`	Computes a parallel prefix over a range of values.
	`parallel_for_each`	A parallel implementation of `std::for_each`. Described in more detail in Chapter 4.
Complex loops	**`parallel_do`**	Processes work items in a container in parallel, with the ability to add additional work items dynamically.
Pipelines	`pipeline`	A class for performing pipelined execution of filters. Described in Appendix B.
	`parallel_pipeline`	A strongly typed function for performing pipelined execution of filters.
Sorting	`parallel_sort`	A function to perform a parallel sort of a sequence. Described in more detail in Chapter 4.

Figure 2-2. *The Generic Algorithms in the Threading Building Blocks library. The bold-face algorithms are described in more detail in this chapter.*

LAMBDA EXPRESSIONS –VS- USER-DEFINED CLASSES

Since the first release of TBB predates the C++11 standard that introduced lambda expressions into the language, the TBB generic algorithms do not require the use of lambda expressions. Sometimes we can use the same interface with lambda expressions or with function objects (functors). In other cases, there are two sets of interfaces for an algorithm: a set that is more convenient with lambda expressions and a set that is more convenient with user-defined objects.

For example, in place of

```
#include <vector>
#include <tbb/tbb.h>

void f(int v);

void sidebar_pfor_lambda(int N, const std::vector<int> &a) {
  tbb::parallel_for(0, N, 1,
    [&a](int i) {
         f(a[i]);
    }
  );
}
```

we can use a user-defined class and write

```
class Body {
  const std::vector<int> &myVector;
public:
  Body(const std::vector<int> &v) : myVector{v} {}
  void operator()(int i) const {
    f(myVector[i]);
  }
};

void sidebar_pfor_functor(int N, const std::vector<int> &a) {
  tbb::parallel_for(0, N, 1, Body{a});
}
```

Often the choice between using a lambda expression or a user-defined object is simply a matter of preference.

Functional / Task Parallelism

Perhaps the simplest algorithm provided by the TBB library is parallel_invoke, a function that allows us to execute as few as two functions in parallel, or as many as we wish to specify:

```
template<typename Func0, [...,] typename FuncN>
void parallel_invoke(const Func0& f0, [...,] const FuncN& fN);
```

The pattern name for this concept is *map* – which we will discuss more in Chapter 8 when we discuss patterns explicitly. The independence expressed by this algorithm/pattern allows it to scale very well, making it the preferred parallelism to use when we can apply it. We will also see that parallel_for, because the loop bodies must be independent, can be used to similar effect.

A complete description of the interfaces available for parallel_invoke can be found in Appendix B. If we have a set of functions that we need to invoke and it is safe to execute the invocations in parallel, we use a parallel_invoke. For example, we can sort two vectors, v1 and v2, by calling a serialQuicksort on each vector, one after the other:

```
serialQuicksort(serial_v1.begin(), serial_v1.end());
serialQuicksort(serial_v2.begin(), serial_v2.end());
```

Or, since these calls are independent of each other, we can use a parallel_invoke to allow the TBB library to create tasks that can be executed in parallel by different worker threads to overlap the two calls, as shown in Figure 2-3.

```
#include <iostream>
#include <vector>
#include <tbb/tbb.h>

struct DataItem {
  int id;
  double value;
  DataItem(int i, double v) : id{i}, value{v} {}
};

using QSVector = std::vector<DataItem>;

void serialQuicksort(QSVector::iterator b, QSVector::iterator e);

void fig_2_3(QSVector &v1, QSVector &v2) {
  tbb::parallel_invoke(
    [&v1]() {serialQuicksort(v1.begin(), v1.end());},
    [&v2]() {serialQuicksort(v2.begin(), v2.end());}
  );
}
```

Figure 2-3. *Using parallel_invoke to execute two serialQuicksort calls in parallel*

If the two invocations of serialQuicksort execute for roughly the same amount of time and there are no resource constraints, this parallel implementation can be completed in half the time it takes to sequentially invoke the functions one after the other.

Note We as developers are responsible for invoking functions in parallel only when they can be safely executed in parallel. That is, TBB will ***not*** automatically identify dependencies and apply synchronization, privatization, or other parallelization strategies to make the code safe. This responsibility is ours when we use `parallel_invoke` or any of the parallel algorithms we discuss in this chapter.

Using `parallel_invoke` is straightforward, but a single invocation of `parallel_invoke` is not very *scalable*. A scalable algorithm makes effective use of additional cores and hardware resources as they become available.

An algorithm shows *strong scaling* if it takes less time to solve a problem with a fixed size as additional cores are added. For example, an algorithm that shows good strong scaling may complete the processing of a given data set two times faster than the sequential algorithm when two cores are available but complete the processing of the same data set 100 times faster than the sequential algorithm when 100 cores are available.

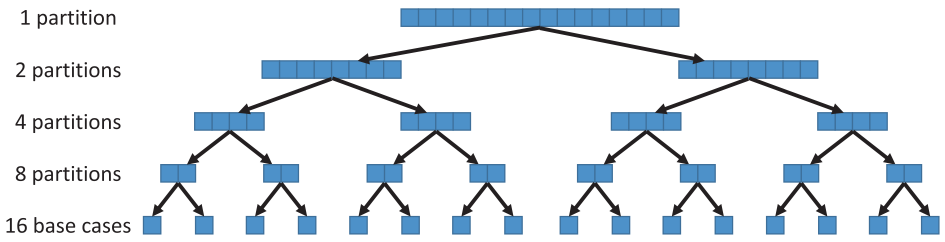
An algorithm shows *weak scaling* if it takes the same amount of time to solve a problem with a fixed data set size *per processor* as more processors are added. For example, an algorithm that shows good weak scaling may be able to process two times the data than its sequential version in a fixed period of time using two processors and 100 times the data than its sequential version in that same fixed period of time when using 100 processors.

Using a `parallel_invoke` to execute two sorts in parallel will demonstrate neither strong nor weak scaling, since the algorithm can at most make use of two processors. If we have 100 processors available, 98 of them will be idle because we have not given them anything to do. Instead of writing code like our example, we should develop scalable applications that allow us to implement parallelism once without the need to revisit the implementation each time new architectures containing more cores become available.

Luckily, TBB can handle nested parallelism efficiently (described in detail in Chapter 9), and so we can create scalable parallelism by using `parallel_invoke` in recursive divide-and-conquer algorithms (a pattern we discuss in Chapter 8). TBB also includes additional generic parallel algorithms, covered later in this chapter, to target patterns that haven proven effective for achieving scalable parallelism, such as loops.

A Slightly More Complicated Example: A Parallel Implementation of Quicksort

A well-known example of a recursive divide-and-conquer algorithm is quicksort, as shown in Figure 2-4. Quicksort works by recursively shuffling an array around pivot values, placing the values that are less than or equal to the pivot value in the left partition of the array and the values that are greater than the pivot value in the right partition of the array. When the recursion reaches the base case, arrays of size one, the whole array has been sorted.



(a) The recursive partitioning performed by quicksort

```
void serialQuicksort(QSVector::iterator b, QSVector::iterator e) {
  if (b >= e) return;

  // do shuffle
  double pivot_value = b->value;
  QSVector::iterator i = b, j = e-1;
  while (i != j) {
    while (i != j && pivot_value < j->value) --j;
    while (i != j && i->value <= pivot_value) ++i;
    std::iter_swap(i, j);
  }
  std::iter_swap(b, i);

  // recursive call
  serialQuicksort(b, i);
  serialQuicksort(i+1, e);
}
```

(b) The source code for a serial implementation of quicksort

Figure 2-4. *A serial implementation of quicksort*

We can develop a parallel implementation of quicksort as shown in Figure 2-5 by replacing the two recursive calls to `serialQuicksort` with a `parallel_invoke`. In addition to the use of `parallel_invoke`, we also introduce a cutoff value. In the original serial quicksort, we recursively partition all the way down to arrays of a single element.

Note Spawning and scheduling a TBB task is not free – a rule of thumb is that a task should execute for at least 1 microsecond or 10,000 processor cycles in order to mitigate the overheads associated with task creation and scheduling. We provide experiments that demonstrate and justify this rule of thumb in more detail in Chapter 16.

To limit overheads in our parallel implementation, we recursively call `parallel_invoke` only until we dip below 100 elements and then directly call `serialQuicksort` instead.

```
void parallelQuicksort(QSVector::iterator b, QSVector::iterator e) {
  const int cutoff = 100;

  if (e - b < cutoff) {
    serialQuicksort(b, e);
  } else {
    // do shuffle
    double pivot_value = b->value;
    QSVector::iterator i = b, j = e - 1;
    while (i != j) {
      while (i != j && pivot_value < j->value) --j;
      while (i != j && i->value <= pivot_value) ++i;
      std::iter_swap(i, j);
    }
    std::iter_swap(b, i);

    // recursive call
    tbb::parallel_invoke(
      [=]() { parallelQuicksort(b, i); },
      [=]() { parallelQuicksort(i + 1, e); }
    );
  }
}
```

Figure 2-5. *A parallel implementation of quicksort using `parallel_invoke`*

You may notice that the parallel implementation of quicksort has a big limitation – the shuffle is done completely serially. At the top level, this means we have an `O(n)` operation that is done on a single thread before any of the parallel work can begin. This can limit the speedup. We leave it up to those that are interested to see how this limitation might be addressed by known parallel partitioning implementations (see the "For More Information" section at the end of this chapter).

Loops: parallel_for, parallel_reduce, and parallel_scan

For many applications, the execution time is dominated by time spent in loops. There are several TBB algorithms that express parallel loops, letting us quickly add scalable parallelism to the important loops in an application. The algorithms labeled as "Simple Loops" in Figure 2-2 are ones where the beginning and end of the iteration space can easily be determined by the time that the loop starts.

For example, we know there will be exactly N iterations in the following loop, so we classify it as a simple loop:

```
for (int i = 0; i < N; ++i)
  f(a[i]);
```

All of the simple loop algorithms in TBB are based on two important concepts, a *Range* and a *Body*. A Range represents a recursively divisible set of values. For a loop, a Range is typically the indices in the iteration space or the values that an iterator will take on as it iterates through a container. The Body is the function we apply to each value in the Range; in TBB, the Body is typically provided as a C++ lambda expression but can also be provided as a function object (see ***Lambda expressions –vs- user-defined classes***).

parallel_for: Applying a Body to Each Element in a Range

Let's start with a small serial `for` loop that applies a function to an element of an array in each iteration:

```
for (int i = 0; i < N; ++i)
  f(a[i]);
```

We can create a parallel version of this loop by using a `parallel_for`:

```
template<typename Index, typename Func>
Func parallel_for(Index first, Index last, [Index step,]
                  const Func& f);
```

A complete description of the interfaces available for `parallel_for` can be found in Appendix B. In the small example loop, the Range is the half-open interval `[0, N)`, the step is 1, and the Body is `f(a[i])`. We can express this as shown in Figure 2-6.

```
#include <tbb/tbb.h>

void f(int v);

void fig_2_6(int N, const std::vector<int> &a) {
  tbb::parallel_for(0, N, 1, [a](int i) {
    f(a[i]);
  });
}
```

***Figure 2-6.** Creating a parallel loop using `parallel_for`*

When TBB executes a `parallel_for`, the Range is divided up into chunks of iterations. Each chunk, paired with a Body, becomes a task that is scheduled onto one of the threads that participate in executing the algorithm. The TBB library handles the scheduling of tasks for us, so all we need to do is to use the `parallel_for` function to express that the iterations of the loop should be executed in parallel. In later chapters, we discuss tuning the behavior of TBB parallel loops. For now, let us assume that TBB generates a good number of tasks for the range size and number of available cores. In most cases, this is a good assumption to make.

It is important to understand that by using a `parallel_for`, we are asserting that it's safe to execute the iterations of the loop in any order and in parallel with each other. The TBB library does nothing to check that executing the iterations of a `parallel_for` (or in fact any of the generic algorithms) in parallel will generate the same results as a serial execution of the algorithm – it is our job as developers to be sure that this is the case when we choose to use a parallel algorithm. In Chapter 5, we discuss synchronization mechanisms in TBB that can be used to make some unsafe code, safe. In Chapter 6, we discuss concurrent containers that provide thread-safe data structures that can also sometimes help us make code thread-safe. But ultimately, we need to ensure when we use a parallel algorithm that any potential changes in read and write access patterns do not change the validity of the results. We also need to ensure that we are using only thread-safe libraries and functions from within our parallel code.

For example, the following loop is ***not*** safe to execute as a `parallel_for` since each iteration depends on the result of the previous iteration. Changing the order of execution of this loop will alter the final values stored in the elements of array `a`:

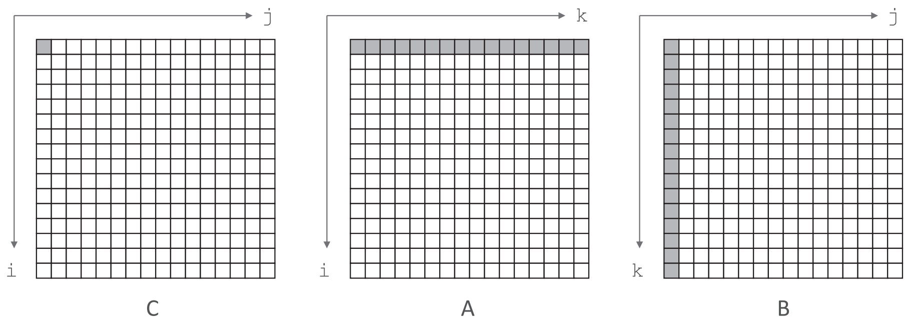
```
for (int i = 1; i < N; ++i)
  a[i] = a[i-1] + 1;
```

Imagine if the array `a={1,0,0,0,...,0}`. After executing this loop sequentially, it will hold `{1,2,3,4,...,N}`. But if the loop executes out-of-order, the results will be different. A mental exercise, when looking for loops that are safe to execute in parallel, is to ask yourself whether the results will be the same if the loop iterations are executed all at once, or in random order, or in reverse order. In this case, if `a={1,0,0,0,...,0}` and the iterations of the loop are executed in reverse order, `a` will hold `{1,2,1,1,...,1}` when the loop is complete. Obviously, execution order matters for this loop!

Formal descriptions of data dependence analysis are beyond the scope of this book but can be found in many compiler and parallel programming books, including *High-Performance Compilers for Parallel Computing* by Michael Wolfe (Pearson) and *Optimizing Compilers for Modern Architectures* by Allen and Kennedy (Morgan Kaufmann). Tools like Intel Inspector in Intel Parallel Studio XE can also be used to find and debug threading errors, including in applications that use TBB.

A Slightly More Complicated Example: Parallel Matrix Multiplication

Figure 2-7 shows a nonoptimized serial implementation of a matrix multiplication loop nest that computes `C = AB` for `MxM` matrices. We use this kernel here for demonstration purposes – if you ever need to use matrix multiply in a real application and do not consider yourself to be an optimization guru – you will almost certainly be better served by using a highly optimized implementation from a math library that implements the Basic Linear Algebra Subprograms (BLAS) like the Intel Math Kernel Library (MKL), BLIS, or ATLAS. Matrix multiplication is a good example here because it is a small kernel and performs a basic operation we are all familiar with. With these disclaimers covered, let us continue with Figure 2-7.



(a) The elements traversed by the first pass through the inner k-loop to calculate a single element of the C matrix.

```
void fig_2_7(int M, double *a, double *b, double *c) {
  for (int i = 0; i < M; ++i) {
    for (int j = 0; j < M; ++j) {
      int c_index = i*M+j;
      for (int k = 0; k < M; ++k) {
        c[c_index] += a[i*M + k] * b[k*M+j];
      }
    }
  }
}
```

(b) The serial implementation.

Figure 2-7. *A nonoptimized implementation of matrix multiplication*

We can quickly implement a parallel version of the matrix multiplication in Figure 2-7 by using `parallel_for` as shown in Figure 2-8. In this implementation, we make the outer `i` loop parallel. An iteration of the outer `i` loop executes the enclosed `j` and `k` loops and so, unless `M` is very small, will have enough work to exceed the 1 microsecond rule of thumb. It is often better to make outer loops parallel when possible to keep overheads low.

```
void fig_2_8(int M, double *a, double *b, double *c) {
  tbb::parallel_for( 0, M, [=](int i) {
    for (int j = 0; j < M; ++j) {
      int c_index = i*M+j;
      for (int k = 0; k < M; ++k) {
        c[c_index] += a[i*M + k] * b[k*M+j];
      }
    }
  });
}
```

Figure 2-8. *A simple `parallel_for` implementation of matrix multiply*

The code in Figure 2-8 quickly gets us a basic parallel version of matrix multiply. While this is a correct parallel implementation, it will leave a lot of performance on the table because of the way it is traversing the arrays. In Chapter 16, we will talk about the advanced features of `parallel_for` that can be used to tune performance.

`parallel_reduce`: Calculating a Single Result Across a Range

Another very common pattern found in applications is a reduction, commonly known as the "reduce pattern" or "map-reduce" because it tends to be used with a *map* pattern (see more about pattern terminology in Chapter 8).

A reduction computes a single value from a collection of values. Example applications include calculating a sum, a minimum value, or a maximum value.

Let's consider a loop that finds the maximum value in an array:

```
int max_value = std::numeric_limits<int>::min();
for (int i = 0; i < a.size(); ++i) {
  max_value = std::max(max_value,a[i]);
}
```

Computing a maximum from a set of values is an associative operation; that is, it's legal to perform this operation on groups of values and then combine those partial results, in order, later. Computing a maximum is also commutative, so we do not even need to combine the partial results in any particular order.

For loops that perform associative operations, TBB provides the function `parallel_reduce`:

```
template<typename Range, typename Value,
         typename Func, typename Reduction>
Value parallel_reduce(const Range& range, const Value& identity,
                      const Func& func, const Reduction& reduction);
```

A complete description of the `parallel_reduce` interfaces is provided in Appendix B.

Many common mathematical operations are associative, such as addition, multiplication, computing a maximum, and computing a minimum. Some operations are associative in theory but are not associative when implemented on real systems due to limitations in numerical representations. We should be aware of the implications of depending on associativity for parallelism (see ***Associativity and floating-point types***).

ASSOCIATIVITY AND FLOATING-POINT TYPES

In computer arithmetic, it is not always practical to represent real numbers with exact precision. Instead, floating-point types such as `float`, `double`, and `long double` are used as an approximation. The consequence of these approximations is that mathematical properties that apply to operations on real numbers do not necessarily apply to their floating-point counterparts. For example, while addition is associative and commutative on real numbers, it is neither of these for floating-point numbers.

For example, if we compute the sum of N real values, each of which is equal to 1.0, we would expect the result to be N.

```
float r = 0.0;
for (uint64_t i = 0; i < N; ++i)
  r += 1.0;
std::cout << "in-order sum == " << r << std::endl;
```

But there is a limited number of significant digits in the `float` representation and so not all integer values can be represented exactly. So, for example, if we run this loop with `N == 10e6` (10 million), we will get an output of `10000000`. But if we execute this loop with `N == 20e6`, we get an output of `16777216`. The variable r simply cannot represent `16777217` since the standard `float` representation has a 24-bit mantissa (significand) and `16777217` requires 25 bits. When we add `1.0`, the result rounds down to `16777216`, and each subsequent addition of `1.0` also rounds down to `16777216`. To be fair, at each step, the result of `16777216` is a good approximation of `16777217`. It is the accumulation of these rounding errors that makes the final result so bad.

If we break this sum into two loops and combine partial results, we get the right answer in both cases:

```
float tmp1 = 0.0, tmp2 = 0.0;
for (uint64_t i = 0; i < N/2; ++i)
  tmp1 += 1.0;
for (uint64_t i = N/2; i < N; ++i)
  tmp2 += 1.0;
float r2 = tmp1 + tmp2;
std::cout << "associative sum == " << r2 << std::endl;
```

Why? Because `r` can represent larger numbers, just not always exactly. The values in `tmp1` and `tmp2` are of similar magnitude, and therefore the addition impacts the available significant digits in the representation, and we get a result that is a good approximation of 20 million. This example is an extreme case of how associativity can change the results of a computation using floating-point numbers.

The take-away of this discussion is that when we use a `parallel_reduce`, it uses associativity to compute and combine partial results in parallel. So, we may get different results when compared to a serial implementation when using floating-point numbers. And in fact, depending on the number of participating threads, the implementation of `parallel_reduce` may choose to create a different number of partial results from run to run. Therefore, we may also get different results from run to run in the parallel implementation, even on the same input.

Before we panic and conclude that we should never use a `parallel_reduce`, we should keep in mind that implementations that use floating-point numbers generally result in an approximation. Getting different results on the same input does not necessarily mean that at least one of the results is wrong. It just means that the rounding errors accumulated differently for two different runs. It is up to us as developers to decide whether or not the differences matter for an application.

If we want to ensure that we at least get the same results on each run on the same input data, we can choose to use a `parallel_deterministic_reduce` as described in Chapter 16. This deterministic implementation always creates the same number of partial results and combines them in the same order for the same input, so the approximation will be the same from run to run.

As with all of the simple loop algorithms, to use a TBB `parallel_reduce`, we need to provide a Range (`range`) and Body (`func`). But we also need to provide an Identity Value (`identity`) and a Reduction Body (`reduction`).

To create parallelism for a `parallel_reduce`, the TBB library divides the `range` into chunks and creates tasks that apply `func` to each chunk. In Chapter 16, we discuss how to use Partitioners to control the size of the chunks that are created, but for now, we can assume that TBB creates chunks of an appropriate size to minimize overheads and balance load. Each task that executes `func` starts with a value `init` that is initialized with `identity` and then computes and returns a partial result for its chunk. The TBB library combines these partial results by calling the `reduction` function to create a single final result for the whole loop.

The `identity` argument is a value that leaves other values unchanged when they are combined with it using the operation that is being parallelized. It is well known that the identity element with respect to addition (additive identity) is "`0`" (since `x + 0 = x`) and that the identity element with respect to multiplication (multiplicative identity) is "`1`" (since `x * 1 = x`). The `reduction` function takes two partial results and combines them.

Figure 2-9 shows how `func` and `reduction` functions may be applied to compute the maximum value from an array of 16 elements if the Range is broken into four chunks. In this example, the associative operation applied by `func` to the elements of the array is max() and the identity element is -∞, since `max(x,-∞)`=x. In C++, we can use `std::max` as the operation and `std::numeric_limits<int>::min()` as the programmatic representation of -∞.

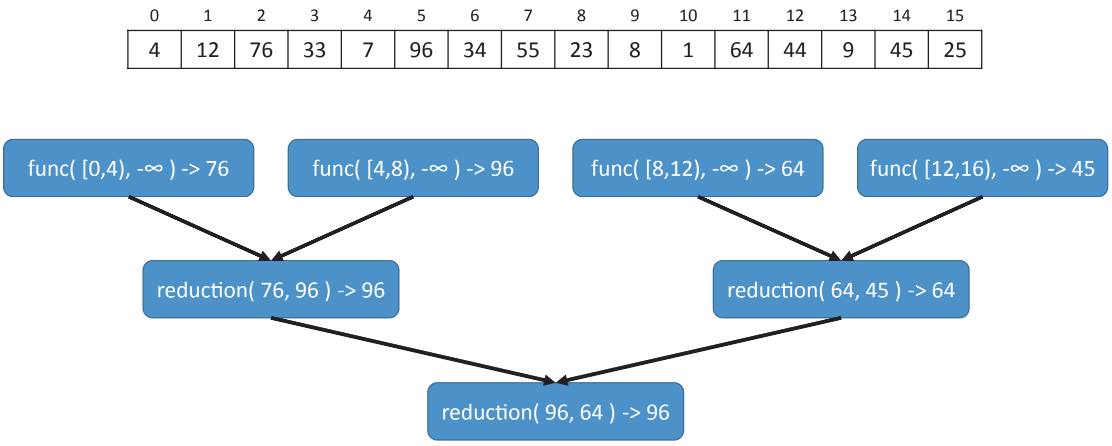


Figure 2-9. *How the `func` and `reduction` functions are called to compute a maximum value*

We can express our simple maximum value loop using a `parallel_reduce` as shown in Figure 2-10.

```
int fig_2_10(const std::vector<int> &a) {
  int max_value = tbb::parallel_reduce(
    /* the range = */ tbb::blocked_range<int>(0, a.size()),
    /* identity = */ std::numeric_limits<int>::min(),
    /* func = */
    [&](const tbb::blocked_range<int> &r, int init) -> int {
      for (int i = r.begin(); i != r.end(); ++i) {
        init = std::max(init, a[i]);
      }
      return init;
    },
    /* reduction = */
    [](int x, int y) -> int {
      return std::max(x,y);
    }
  );
  return max_value;
}
```

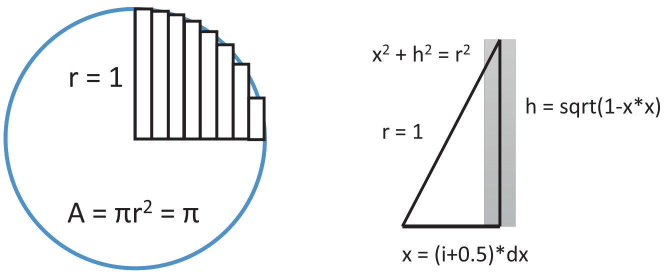
Figure 2-10. *Using `parallel_reduce` to compute a maximum value*

You may notice in Figure 2-10 that we use a `blocked_range` object for the Range, instead of just providing the beginning and ending of the range as we did with `parallel_for`. The `parallel_for` algorithm provides a simplified syntax that is not available with `parallel_reduce`. For `parallel_reduce`, we must pass a Range object directly, but luckily we can use one of the predefined ranges provided by the library, which include `blocked_range`, `blocked_range2d`, and `blocked_range3d` among others. These other range objects will be described in more detail in Chapter 16, and their complete interfaces are provided in Appendix B.

A `blocked_range`, used in Figure 2-10, represents a 1D iteration space. To construct one, we provide the beginning and ending value. In the Body, we use its `begin()` and `end()` functions to get the beginning and ending values of the chunk of values that this body execution has been assigned and then iterate over that subrange. In Figure 2-8, each individual value in the Range was sent to the `parallel_for` Body, and so there is no need for an `i`-loop to iterate over a range. In Figure 2-10, the Body receives a `blocked_range` object that represents a chunk of iterations, and therefore we still have an `i`-loop that iterates over the entire chunk assigned to it.

A Slightly More Complicated Example: Calculating π by Numerical Integration

Figure 2-11 shows an approach to calculate π by numerical integration. The height of each rectangle is calculated using the Pythagorean Theorem. The area of one quadrant of a unit circle is computed in the loop and multiplied by 4 to get the total area of the circle, which is equal to π.



(a) Using Numerical Integration to calculate π

```
#include <cmath>

double fig_2_11(int num_intervals) {
  double dx = 1.0 / num_intervals;
  double sum = 0.0;
  for (int i = 0; i < num_intervals; ++i) {
    double x = (i+0.5)*dx;
    double h = std::sqrt(1-x*x);
    sum += h*dx;
  }
  double pi = 4 * sum;
  return pi;
}
```

(b) The serial implementation

Figure 2-11. *A serial π calculation using the rectangular integral method*

The code in Figure 2-11 computes the sum of the areas of all of the rectangles, a reduction operation. To use a TBB `parallel_reduce`, we need to identify the `range`, `body`, `identity` value, and `reduction` function. For this example, the `range` is `[0, num_intervals)`, and the body will be similar to the `i`-loop in Figure 2-11. The `identity` value is `0.0` since we are performing a sum. And the `reduction` body, which needs to combine partial results, will return the sum of two values. The parallel implementation using a TBB parallel_reduce is shown in Figure 2-12.

```
#include <cmath>
#include <tbb/tbb.h>

double fig_2_12(int num_intervals) {
  double dx = 1.0 / num_intervals;
  double sum = tbb::parallel_reduce(
    /* range = */ tbb::blocked_range<int>(0, num_intervals),
    /* idenity = */ 0.0,
    /* func */
    [=](const tbb::blocked_range<int> &r, double init)
    -> double {
      for (int i = r.begin(); i != r.end(); ++i) {
        double x = (i + 0.5)*dx;
        double h = std::sqrt(1 - x*x);
        init += h*dx;
      }
      return init;
    },
    /* reduction */
    [](double x, double y) -> double {
      return x + y;
    }
  );
  double pi = 4 * sum;
  return pi;
}
```

Figure 2-12. *Implementation of pi using* `tbb::parallel_reduce`

As with parallel_for, there are advanced features and options that can be used with parallel_reduce to tune performance and to manage rounding errors (see **Associativity and floating-point types**). These advanced options are covered in Chapter 16.

parallel_scan: A Reduction with Intermediate Values

A less common, but still important, pattern found in applications is a scan (sometimes called a prefix). A scan is similar to a reduction, but not only does it compute a single value from a collection of values, it also calculates an intermediate result for each element in the Range (*the prefixes*). An example is a running sum of the values x_0, x_1, ... x_N. The results include each value in the running sum, y_0, y_1, ... y_N, and the final sum y_N.

$y_0 = x_0$

$y_1 = x_0 + x_1$

...

$y_N = x_0 + x_1 + \ldots + x_N$

A serial loop that computes a running sum from a vector v follows:

```
int serialImpl(const std::vector<int> &v,
               std::vector<int> &rsum) {
  int N = v.size();
  rsum[0] = v[0];
  for (int i = 1; i < N; ++i) {
    rsum[i] = rsum[i-1] + v[i];
  }
  int final_sum = rsum[N-1];
  return final_sum;
}
```

On the surface, a scan looks like a serial algorithm. Each prefix depends on the results computed in all of the previous iterations. While it might seem surprising, there are however efficient parallel implementations of this seemingly serial algorithm. The TBB parallel_scan algorithm implements an efficient parallel scan. Its interface requires that we provide a range, an identity value, a scan body, and a combine body:

```
template<typename Range, typename Value,
         typename Scan, typename Combine>
Value parallel_scan( const Range& range, const Value& identity,
                     const Scan& scan, const Combine& combine);
```

The range, identity value, and combine body are analogous to the range, identity value, and reduction body of parallel_reduce. And, as with the other simple loop algorithms, the range is divided by the TBB library into chunks and TBB tasks are created to apply the body (scan) to these chunks. A complete description of the parallel_scan interfaces is provided in Appendix B.

What is different about parallel_scan is that the scan body may be executed more than once on the same chunk of iterations – first in a *pre-scan* mode and then later in a *final-scan* mode.

In *final-scan* mode, the body is passed an accurate prefix result for the iteration that immediately precedes its subrange. Using this `value`, the body computes and stores the prefixes for each iteration in its subrange and returns the accurate prefix for the last element in its subrange.

However, when the scan body is executed in *pre-scan* mode, it receives a starting prefix value that is not the final value for the element that precedes its given range. Just like with `parallel_reduce`, a `parallel_scan` depends on associativity. In pre-scan mode, the starting prefix value may represent a subrange that precedes it, but not the complete range that precedes it. Using this value, it returns a (not yet final) prefix for the last element in its subrange. The returned value represents a partial result for the starting prefix combined with its subrange. By using these *pre-scan* and *final-scan* modes, it is possible to exploit useful parallelism in a scan algorithm.

How Does This Work?

Let's look at the running sum example again and think about computing it in three chunks `A`, `B`, and `C`. In a sequential implementation, we compute all of the prefixes for `A`, then `B`, and then `C` (three steps done in order). We can do better with a parallel scan as shown in Figure 2-13.

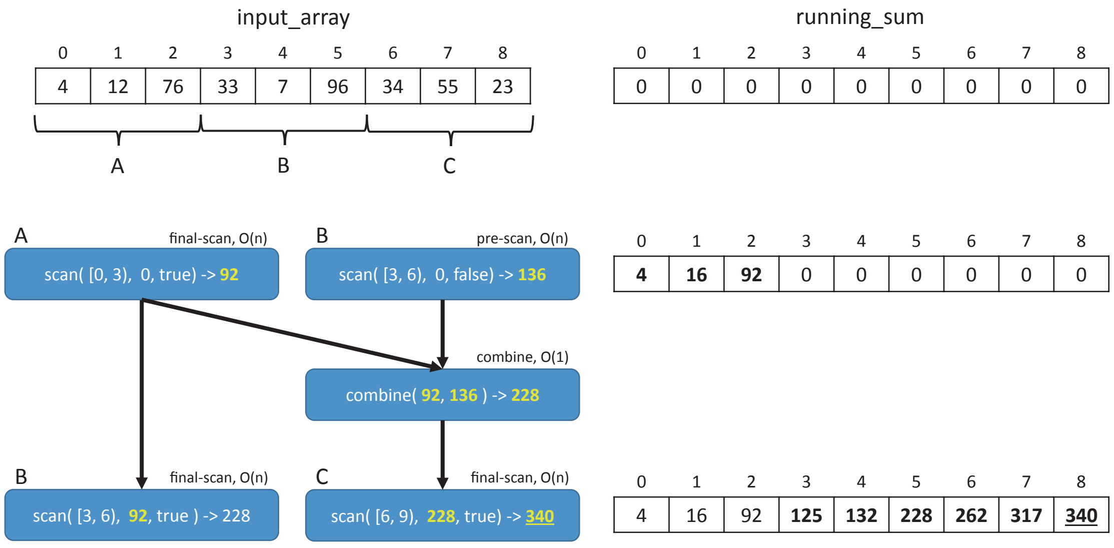


***Figure 2-13.** Performing a scan in parallel to compute a sum*

First, we compute the scan of A in final-scan mode since it is the first set of values and so its prefix values will be accurate if it is passed an initial value of identity. At the same time that we start A, we start B in pre-scan mode. Once these two scans are done, we can now calculate accurate starting prefixes for both B and C. To B we provide the final result from A (92), and to C we provide the final-scan result of A combined with the pre-scan result of B (92+136 = 228).

The combine operation takes constant time, so it is much less expensive than the scan operations. Unlike the sequential implementation that takes three large steps that are applied one after the other, the parallel implementation executes final-scan of A and pre-scan of B in parallel, then performs a constant-time combine step, and then finally computes final-scan of B and C in parallel. If we have at least two cores and N is sufficiently large, a parallel prefix sum that uses three chunks can therefore be computed in about two thirds of the time of the sequential implementation. And parallel_prefix can of course execute with more than three chunks to take advantage of more cores.

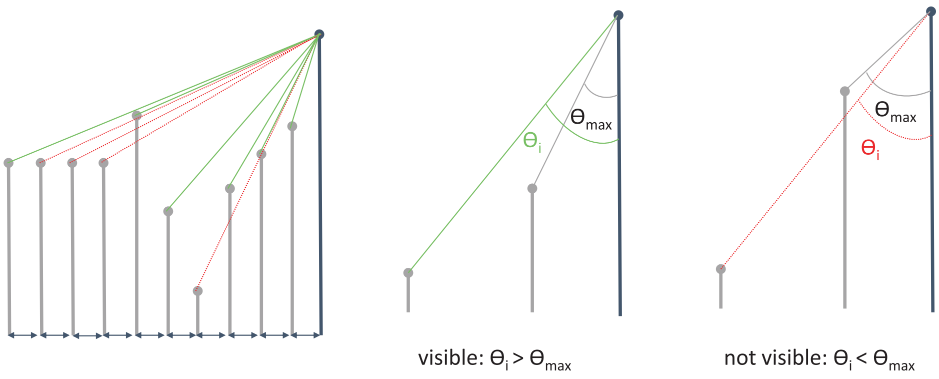
Figure 2-14 shows an implementation of the simple partial sum example using a TBB parallel_scan. The range is the interval [1, N), the identity value is 0, and the combine function returns the sum of its two arguments. The scan body returns the partial sum for all of the values in its subrange, added to the initial sum it receives. However, only when its is_final_scan argument is true does it assign the prefix results to the running_sum array.

```
int fig_2_14(const std::vector<int> &v, std::vector<int> &rsum) {
  int N = v.size();
  rsum[0] = v[0];
  int final_sum = tbb::parallel_scan(
    /* range = */ tbb::blocked_range<int>(1, N),
    /* identity = */ (int)0,
    /* scan body */
    [&v, &rsum](const tbb::blocked_range<int> &r,
                int sum, bool is_final_scan) -> int {
      for (int i = r.begin(); i < r.end(); ++i) {
        sum += v[i];
        if (is_final_scan)
          rsum[i] = sum;
      }
      return sum;
    },
    /* combine body */
    [](int x, int y) {
      return x + y;
    }
  );
  return final_sum;
}
```

Figure 2-14. *Implementation of a running sum using parallel_scan*

A Slightly More Complicated Example: Line of Sight

Figure 2-15 shows a serial implementation of a line of sight problem similar to the one described in *Vector Models for Data-Parallel Computing, Guy E. Blelloch (The MIT Press)*. Given the altitude of a viewing point and the altitudes of points at fixed intervals from the viewing point, the line of sight code determines which points are visible from the viewing point. As shown in Figure 2-15, a point is not visible if any point between it and the viewing point, `altitude[0]`, has a larger angle ϴ. The serial implementation performs a scan to compute the maximum ϴ value for all points between a given point and the viewing point. If the given point's ϴ value is larger than this maximum angle, then it is a visible point; otherwise, it is not visible.



Θ_{max}
Θ_i
visible: $\Theta_i > \Theta_{max}$
Θ_{max}
Θ_i
not visible: $\Theta_i < \Theta_{max}$

(a) Calculating points that are visible from a viewing point.

```cpp
void fig_2_15(const std::vector<double> &altitude,
              std::vector<bool> &is_visible, double dx) {
  const int N = altitude.size();

  double max_angle = std::atan2(dx, altitude[0] - altitude[1]);
  double my_angle = 0.0;

  for (int i = 2; i < N; ++i ) {
    my_angle = std::atan2(i * dx, altitude[0] - altitude[i]);
    if (my_angle >= max_angle) {
      max_angle = my_angle;
    } else {
      is_visible[i] = false;
    }
  }
}
```

(b) The serial implementation.

Figure 2-15. *A line of sight example*

Figure 2-16 shows a parallel implementation of the line of sight example that uses a TBB `parallel_scan`. When the algorithm completes, the `is_visible` array will contain the visibility of each point (`true` or `false`). It is important to note that the code in Figure 2-16 needs to compute the maximum angle at each point in order to determine the point's visibility, but the final output is the visibility of each point, not the maximum angle at each point. Because the `max_angle` is needed but is not a final result, it is computed in both `pre-scan` and `final-scan` mode, but the `is_visible` values are stored for each point only during `final-scan` executions.

```
void fig_2_16(const std::vector<double> &altitude,
              std::vector<bool> &is_visible, double dx) {
  const int N = altitude.size();
  double max_angle = std::atan2(dx, altitude[0] - altitude[1]);

  double final_max_angle = tbb::parallel_scan(
    /* range = */ tbb::blocked_range<int>(1, N),
    /* identity */ 0.0,
    /* scan body */
    [&altitude, &is_visible, dx](const tbb::blocked_range<int> &r,
                                 double max_angle,
                                 bool is_final_scan) -> double {
      for (int i = r.begin(); i != r.end(); ++i) {
        double my_angle = atan2(i*dx, altitude[0] - altitude[i]);
        if (my_angle >= max_angle)
          max_angle = my_angle;
        if (is_final_scan && my_angle < max_angle)
          is_visible[i] = false;
      }
      return max_angle;
    },
    [](double a, double b) {
      return std::max(a,b);
    }
  );
}
```

Figure 2-16. *An implementation of the line of sight using `parallel_scan`*

Cook Until Done: parallel_do and parallel_pipeline

For some applications, simple loops get us full coverage of the useful parallelism. But for others, we need to express parallelism in loops where the range cannot be fully computed before the loop starts. For example, consider a while loop:

```
while(auto i = get_image()) {
  f(i);
}
```

This loop keeps reading in images until there are no more images to read. After each image is read, it is processed by the function `f`. We cannot use a `parallel_for` because we don't know how many images there will be and so cannot provide a range.

A more subtle case is when we have a container that does not provide random-access iterators:

```
std::list<image_type> my_images = get_image_list();
for (auto &i : my_list) {
  f(i);
}
```

Note In C++, an iterator is an object that points to an element in a range of elements and defines operators that provide the ability to iterate through the elements of the range. There are different categories of iterators including *forward*, *bidirectional*, and *random-access* iterators. A random-access iterator can be moved to point to any element in the range in constant time.

Because a `std::list` does not support random access to its elements, we can obtain the delimiters of the range `my_images.begin()` and `my_images.end()`, but we cannot get to elements in between these points without sequentially traversing the list. The TBB library therefore cannot quickly (in constant time) create chunks of iterations to hand out as tasks since it cannot quickly point to the beginning and ending points of these chunks.

To handle complex loops like these, The TBB library provides two generic algorithms: `parallel_do` and `parallel_pipeline`.

`parallel_do`: Apply a Body Until There Are No More Items Left

A TBB `parallel_do` applies a Body to work items until there are no more items to process. Some work items can be provided up front when the loop begins, and others can be added by Body executions as they are processing other items.

The `parallel_do` function has two interfaces, one that accepts a first and last iterator and another that accepts a container. A complete description of the `parallel_do` interfaces is provided in Appendix B. In this section, we will look at the version that receives a container:

```
template<typename Container, typename Body>
void parallel_do(Container c, Body body);
```

As a simple example, let us start with a `std::list` of `std::pair<int, bool>` elements, each of which contains a random integer `value` and `false`. For each element, we will calculate whether or not the `int value` is a prime number; if so, we store `true` to the `bool value`. We will assume that we are given functions that populate the container and determine if a number is prime. A serial implementation follows:

```
using PrimesValue = std::pair<int, bool>;
using PrimesList = std::list<PrimesValue>;

bool isPrime(int n);

void serialImpl(PrimesList &values) {
  for (PrimesList::reference v : values) {
    if (isPrime(v.first))
      v.second = true;
  }
}
```

We can create a parallel implementation of this loop using a TBB `parallel_do` as shown in Figure 2-17.

```
void fig_2_17(PrimesList &values) {
  tbb::parallel_do(values,
    [](PrimesList::reference v) {
      if (isPrime(v.first))
        v.second = true;
    }
  );
}
```

Figure 2-17. *An implementation of the prime number loop using a `parallel_do`*

The TBB `parallel_do` algorithm will safely traverse the container sequentially, while creating tasks to apply the body to each element. Because the container has to be traversed sequentially, a `parallel_do` is not as scalable as a `parallel_for`, but as long as the body is relatively large (> 100,000 clock cycles), the traversal overhead will be negligible compared to the parallel executions of the body on the elements.

In addition to handling containers that do not provide random access, the parallel_do also allows us to add additional work items from within the body executions. If bodies are executing in parallel and they add new items, these items can be spawned in parallel too, avoiding the sequential task spawning limitations of parallel_do.

Figure 2-18 provides a serial implementation that calculates whether values are prime numbers, but the values are stored in a tree instead of a list.

```
using PrimesValue = std::pair<int, bool>;

struct PrimesTreeElement {
  using Ptr = std::shared_ptr<PrimesTreeElement>;

  PrimesValue v;
  Ptr left;
  Ptr right;
  PrimesTreeElement(const PrimesValue &_v);
}

bool isPrime(int n);

void serialCheckPrimesElem(PrimesTreeElement::Ptr e) {
  if (e) {
    if (isPrime(e->v.first))
      e->v.second = true;
    if (e->left) serialCheckPrimesElem(e->left);
    if (e->right) serialCheckPrimesElem(e->right);
  }
}
```

Figure 2-18. *Checking for prime numbers in a tree of elements*

We can create a parallel implementation of this tree version using a parallel_do, as shown in Figure 2-19. To highlight the different ways to provide work items, in this implementation we use a container that holds a single tree of values. The parallel_do starts with only a single work item, but two items are added in each body execution, one to process the left subtree and the other to process the right subtree. We use the parallel_do_feeder.add method to add new work items to the iteration space. The class parallel_do_feeder is defined by the TBB library and is passed as the second argument to the body.

```
template<typename Item>
struct parallel_do_feeder {
  void add( const Item& item );
  // Supported since C++11
  void add( Item&& item );
};
```

The number of available work items increases exponentially as the bodies traverse down the levels of the tree. In Figure 2-19, we add new items through the `feeder` even before we check if the current element is a prime number, so that the other tasks are spawned as quickly as possible.

```
using PrimesValue = std::pair<int, bool>;

struct PrimesTreeElement {
  using Ptr = std::shared_ptr<PrimesTreeElement>;

  PrimesValue v;
  Ptr left;
  Ptr right;
  PrimesTreeElement(const PrimesValue &_v) : left{}, right{} {
    v.first = _v.first;
    v.second = _v.second;
  }
};

bool isPrime(int n);

void fig_2_19(PrimesTreeElement::Ptr root) {
  PrimesTreeElement::Ptr tree_array[] = {root};
  tbb::parallel_do(tree_array,
    [](PrimesTreeElement::Ptr e,
      tbb::parallel_do_feeder<PrimesTreeElement::Ptr>& feeder)
{
        if (e) {
          if (e->left) feeder.add(e->left);
          if (e->right) feeder.add(e->right);
          if (isPrime(e->v.first))
            e->v.second = true;
        }
      }
  );
}
```

Figure 2-19. *Checking for prime numbers in a tree of elements using a TBB `parallel_do`*

We should note that the two uses we considered of `parallel_do` have the potential to scale for different reasons. The first implementation, without the feeder in Figure 2-17, can show good performance if each body execution has enough work to do to mitigate the overheads of traversing the list sequentially. In the second implementation, with the feeder in Figure 2-19, we start with only a single work item, but the number of available work items grows quickly as the bodies execute and add new items.

A Slightly More Complicated Example: Forward Substitution

Forward substitution is a method to solve a set of equations `Ax = b`, where `A` is an `nxn` lower triangular matrix. Viewed as matrices, the set of equations looks like

$$\begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

and can be solved a row at a time:

$$x_1 = b_1 / a_{11}$$

$$x_2 = \left(b_2 - a_{21}x_1\right) / a_{22}$$

$$x_3 = \left(b_3 - a_{31}x_1 - a_{32}x_2\right) / a_{33}$$

$$\vdots$$

$$x_m = \left(b_n - a_{n1}x_1 - a_{n2}x_2 - \ldots - a_{nn-1}x_{n-1}\right) / a_{nn}$$

The serial code for a direct implementation of this algorithm is shown in Figure 2-20. In the serial code, b is destructively updated to store the sums for each row.

```
void fig_2_20(std::vector<double> &x,
              const std::vector<double> &a,
              std::vector<double> &b) {
  const int N = x.size();
  for (int i = 0; i < N; ++i) {
    for (int j = 0; j < i; ++j) {
      b[i] -= a[j + i*N] * x[j];
    }
    x[i] = b[i] / a[i + i*N];
  }
}
```

Figure 2-20. *The serial code for a direct implementation of forward substitution. This implementation is written to make the algorithm clear – not for best performance.*

Figure 2-21(a) shows the dependencies between the iterations of the body of the `i,j` loop nest in Figure 2-20. Each iteration of the inner `j` loop (shown by the rows in the figure) performs a reduction into `b[i]` and also depends on all of the elements of `x` that were written in earlier iterations of the `i` loop. We could use a `parallel_reduce` to parallelize the inner `j` loop, but there may not be enough work in the early iterations of the `i` loop to make this profitable. The dotted line in Figure 2-21(a) shows that there is another way to find parallelism in this loop nest by looking diagonally across the iteration space. We can exploit this parallelism by using a `parallel_do` to add iterations only as their dependencies are satisfied, similar to how we added new tree elements as we discovered them in Figure 2-19.

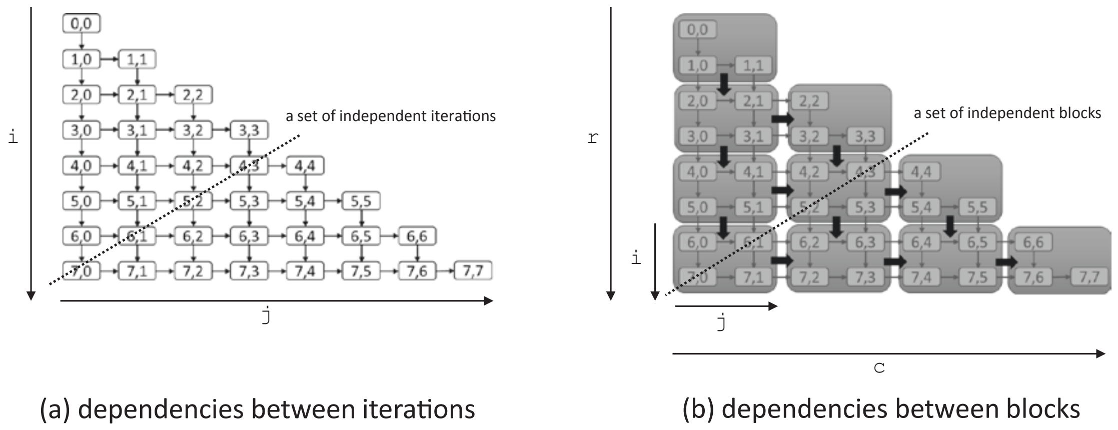

i
j
a set of independent iterations
(a) dependencies between iterations
r
i
j
c
a set of independent blocks
(b) dependencies between blocks

Figure 2-21. *The dependencies in forward substitution for a small 8 × 8 matrix. In (a), the dependencies between iterations are shown. In (b), the iterations are grouped into blocks to reduce scheduling overheads. In both (a) and (b), each block must wait for its neighbor above and its neighbor to its left to complete before it can safely execute.*

If we express the parallelism for each iteration separately, we will create tasks that are too small to overcome scheduling overheads since each task will only be a few floating-point operations. Instead, we can modify the loop nest to create blocks of iterations, as shown in Figure 2-21(b). The dependence pattern stays the same, but we will be able to schedule these larger blocks of iterations as tasks. A blocked version of the serial code is shown in Figure 2-22.

```
void fig_2_22(std::vector<double> &x, const std::vector<double> &a,
              std::vector<double> &b) {
  const int N = x.size();
  const int block_size = 512;
  const int num_blocks = N / block_size;

  for ( int r = 0; r < num_blocks; ++r ) {
    for ( int c = 0; c <= r; ++c ) {
      int i_start = r*block_size, i_end = i_start + block_size;
      int j_start = c*block_size, j_max = j_start + block_size - 1;
      for (int i = i_start; i < i_end; ++i) {
        int j_end = (i <= j_max) ? i : j_max + 1;
        for (int j = j_start; j < j_end; ++j) {
          b[i] -= a[j + i*N] * x[j];
        }
        if (j_end == i) {
          x[i] = b[i] / a[i + i*N];
        }
      }
    }
  }
}
```

Figure 2-22. *A blocked version of the serial implementation of forward substitution*

A parallel implementation that uses `parallel_do` is shown in Figure 2-23. Here, we use the interface of `parallel_do` that allows us to specify a beginning and ending iterator, instead of an entire container. You can see the details of this interface in Appendix B.

Unlike with the prime number tree example in Figure 2-19, we don't want to simply send every neighboring block to the feeder. Instead, we initialize an array of counters, `ref_count,` to hold the number of blocks that must complete before each block is allowed to start executing. Atomic variables will be discussed more in Chapter 5. For our purposes here, we can view these as variables that we can modify safely in parallel; in particular, the decrements are done in a thread-safe way. We initialize the counters so that the top-left element has no dependencies, the first column and the blocks along the diagonal have a single dependency, and all others have two dependencies. These counts match the number of predecessors for each block as shown in Figure 2-21.

```
void fig_2_23(std::vector<double> &x, const std::vector<double> &a,
              std::vector<double> &b) {
  const int N = x.size();
  const int block_size = 512;
  const int num_blocks = N / block_size;
  std::vector<tbb::atomic<char>> ref_count(num_blocks*num_blocks);
  for (int r = 0; r < num_blocks; ++r) {
    for (int c = 0; c <= r; ++c) {
      if (r == 0 && c == 0)
        ref_count[r*num_blocks + c] = 0;
      else if (c == 0 || r == c)
        ref_count[r*num_blocks + c] = 1;
      else
        ref_count[r*num_blocks + c] = 2;
    }
  }

  using BlockIndex = std::pair<size_t, size_t>;
  BlockIndex top_left(0,0);

  tbb::parallel_do( &top_left, &top_left+1,
    [&](const BlockIndex &bi,
        tbb::parallel_do_feeder<BlockIndex> &feeder) {
      size_t r = bi.first;
      size_t c = bi.second;
      int i_start = r*block_size, i_end = i_start + block_size;
      int j_start = c*block_size, j_max = j_start + block_size - 1;
      for (int i = i_start; i < i_end; ++i) {
        int j_end = (i <= j_max) ? i : j_max + 1;
        for (int j = j_start; j < j_end; ++j) {
          b[i] -= a[j + i*N] * x[j];
        }
        if (j_end == i) {
          x[i] = b[i] / a[i + i*N];
        }
      }
      // add successor to right if ready
      if (c + 1 <= r && --ref_count[r*num_blocks + c + 1] == 0) {
        feeder.add(BlockIndex(r, c + 1));
      }
      // add successor below if ready
      if (r + 1 < (size_t)num_blocks
          && --ref_count[(r+1)*num_blocks + c] == 0) {
        feeder.add(BlockIndex(r+1, c));
      }
    }
  );
}
```

Figure 2-23. *An implementation of forward substitution using* `parallel_do`

In the call to `parallel_do` in Figure 2-23, we initially provide only the top-left block, `[&top_left, &top_left+1)`. But in each body execution, the `if`-statements at the bottom decrement the atomic counters of the blocks that are dependent on the block that was just processed. If a counter reaches zero, that block has all of its dependencies satisfied and is provided to the feeder.

Like the previous prime number examples, this example demonstrates the hallmark of applications that use `parallel_do`: the parallelism is constrained by the need to sequentially access a container or by the need to dynamically find and feed work items to the algorithm.

parallel_pipeline: Streaming Items Through a Series of Filters

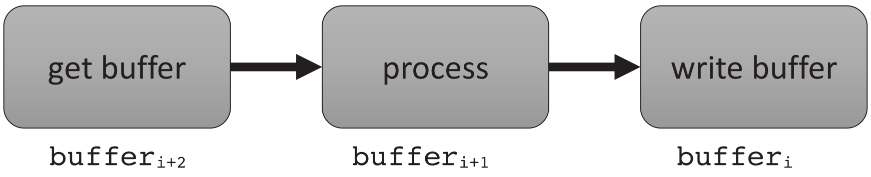
The second generic parallel algorithm in TBB used to handle complex loops is `parallel_pipeline`. A pipeline is a linear sequence of *filters* that transform *items* as they pass through them. Pipelines are often used to process data that stream into an application such as video or audio frames, or financial data. In Chapter 3, we will discuss the Flow Graph interfaces that let us build more complex graphs that include fan-in and fan-out to and from filters.

Figure 2-24 shows a small example loop that reads in arrays of characters, transforms the characters by changing all of the lowercase characters to uppercase and all of the uppercase characters to lowercase, and then writes the results in order to an output file.

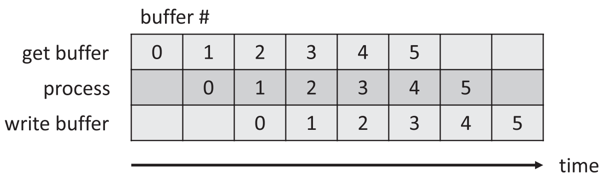
```
using CaseStringPtr = std::shared_ptr<std::string>;
CaseStringPtr getCaseString(std::ofstream &f);
void writeCaseString(std::ofstream &f, CaseStringPtr s);

void fig_2_24(std::ofstream &caseBeforeFile,
              std::ofstream &caseAfterFile) {
  while (CaseStringPtr s_ptr = getCaseString(caseBeforeFile)) {
    std::transform(s_ptr->begin(), s_ptr->end(), s_ptr->begin(),
      [](char c) -> char {
        if (std::islower(c))
          return std::toupper(c);
        else if (std::isupper(c))
          return std::tolower(c);
        else
          return c;
      }
    );
    writeCaseString(caseAfterFile, s_ptr);
  }
}
```

Figure 2-24. *A serial case change example*

The operations have to be done in order on each buffer, but we can overlap the execution of different filters applied to different buffers. Figure 2-25(a) shows this example as a pipeline, where the “write buffer” operates on `buffer`$_i$, while in parallel the “process” filter operates on `buffer`$_{i+1}$ and the “get buffer” filter reads in `buffer`$_{i+2}$.

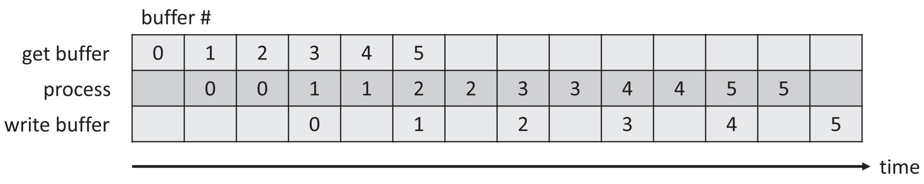


(a) Three buffers are being processed in parallel by different filters.



get buffer	0	1	2	3	4	5		
process		0	1	2	3	4	5	
write buffer			0	1	2	3	4	5

(b) In the steady-state, if all of the filters can work on only 1 item at a time, a maximum parallelism of 3 is possible.



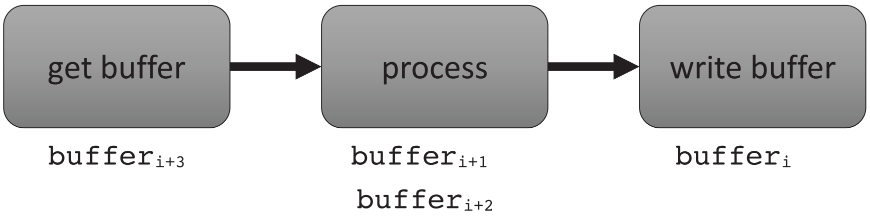
get buffer	0	1	2	3	4	5								
process		0	0	1	1	2	2	3	3	4	4	5	5	
write buffer				0		1		2		3		4		5

(c) Here the process filter takes 2 times longer than the other filters, and so the other filters are sometimes idle.

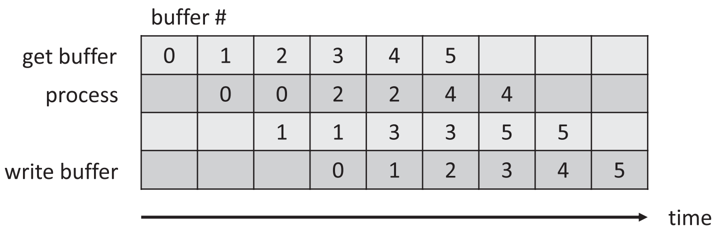
Figure 2-25. *The case change example using a pipeline*

As illustrated in Figure 2-25(b), in the steady state, each filter is busy, and their executions are overlapped. However, as shown in Figure 2-25(c), unbalanced filters decrease speedup. The performance of a pipeline of serial filters is limited by the slowest serial stage.

The TBB library supports both serial and parallel filters. A parallel filter can be applied in parallel to different items in order to increase the throughput of the filter. Figure 2-26(a) shows the “case change” example, with the middle/process filter executing in parallel on two items. Figure 2-26(b) illustrates that if the middle filter takes twice as long as the other filters to complete on any given item, then assigning two threads to this filter will allow it to match the throughput of the other filters.



(a) A parallel filter can process more than one item in parallel.


buffer #
time

get buffer	0	1	2	3	4	5			
process		0	0	2	2	4	4		
			1	1	3	3	5	5	
write buffer				0	1	2	3	4	5

(b) More threads can be provided to a parallel filter to increase its throughput to match the throughput of the other filters.

Figure 2-26. *The case change example using a pipeline with a parallel filter. By using two copies of the parallel filter, the pipeline maximizes throughput.*

A complete description of the `parallel_pipeline` interfaces is provided in Appendix B. The interface of `parallel_pipeline` we use in this section is shown as follows:

```
void parallel_pipeline( size_t max_number_of_live_tokens,
                        const filter_t<void,void>& filter_chain);
```

The first argument max_number_of_live_tokens is the maximum number of items that will be allowed to flow through the pipeline at any given time. This value is necessary to constrain resource consumption. For example, consider the simple three filter pipeline. What if the middle filter is a serial filter and it takes 1000 times longer than the filter that gets new buffers? The first filter might allocate 1000 buffers only to queue them up before the second filter – wasting a lot of memory.

The second argument to parallel_pipeline is filter_chain, a series of filters created by concatenating filters that are created using the make_filter function:

```
template<typename T, typename U, typename Func>
filter_t<T,U> make_filter(filter::mode mode, const Func& f );
```

The template arguments T and U specify the input and output types of the filter. The mode argument can be serial_in_order, serial_out_of_order, or parallel. And the f argument is the body of the filter. Figure 2-27 shows the implementation of the case change example using a TBB parallel_pipeline. A more complete description of the parallel_pipeline interfaces is provided in Appendix B.

We can note that the first filter, since its input type is void, receives a special argument of type tbb::flow_control. We use this argument to signal when the first filter in a pipeline is no longer going to generate new items. For example, in the first filter in Figure 2-27, we call stop() when the pointer returned by getCaseString() is null.

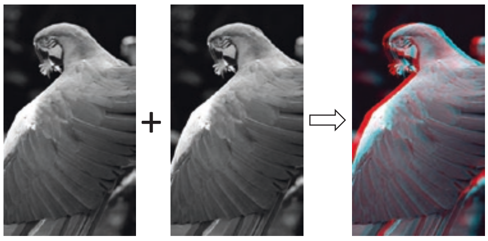
```
void fig_2_27(int num_tokens, std::ofstream &caseBeforeFile,
              std::ofstream &caseAfterFile) {
  tbb::parallel_pipeline(
    /* tokens */ num_tokens,
    /* the get filter */
    tbb::make_filter<void, CaseStringPtr>(
      /* filter node */ tbb::filter::serial_in_order,
      /* filter body */
      [&](tbb::flow_control &fc) -> CaseStringPtr {
        CaseStringPtr s_ptr = getCaseString(caseBeforeFile);
        if (!s_ptr)
          fc.stop();
        return s_ptr;
      }) & // concatenation operation
    /* make the change case filter */
    tbb::make_filter<CaseStringPtr, CaseStringPtr>(
      /* filter node */ tbb::filter::parallel,
      /* filter body */
      [](CaseStringPtr s_ptr) -> CaseStringPtr {
        std::transform(s_ptr->begin(), s_ptr->end(), s_ptr->begin(),
          [](char c) -> char {
            if (std::islower(c))
              return std::toupper(c);
            else if (std::isupper(c))
              return std::tolower(c);
            else
              return c;
          });
        return s_ptr;
      }) & // concatenation operation
    /* make the write filter */
    tbb::make_filter<CaseStringPtr, void>(
      /* filter node */ tbb::filter::serial_in_order,
      /* filter body */
      [&](CaseStringPtr s_ptr) -> void {
        writeCaseString(caseAfterFile, s_ptr);
      })
  );
}
```

Figure 2-27. *The case change example using a pipeline with a parallel middle filter*

In this implementation, the first and last filters are created using the `serial_in_order` mode. This specifies that both filters should run on only one item at a time and that the last filter should execute the items in the same order that the first filter generated them in. A `serial_out_of_order` filter is allowed to execute the items in any order. The middle filter is passed `parallel` as its mode, allowing it to execute on different items in parallel. The modes supported by `parallel_pipeline` are described in more detail in Appendix B.

A Slightly More Complicated Example: Creating 3D Stereoscopic Images

A more complicated example of a pipeline is shown in Figure 2-28. A while loop reads in frame numbers, and then for each frame it reads a left and right image, adds a red coloring to the left image and a blue coloring to the right image. It then merges the resulting two images into a single red-cyan 3D stereoscopic image.



(a) Images are combined into a red-cyan 3D stereoscopic image. The original photograph was taken by Elena Adams.

```
class PNGImage {
public:
  uint64_t frameNumber = -1;
  unsigned int width = 0, height = 0;
  std::shared_ptr<std::vector<unsigned char>> buffer;
  static const int numChannels = 4;
  static const int redOffset = 0;
  static const int greenOffset = 1;
  static const int blueOffset = 2;

  PNGImage() {}
  PNGImage(uint64_t frame_number, const std::string& file_name);
  PNGImage(const PNGImage &p);
  virtual ~PNGImage() {}
  void write() const;
};

int getNextFrameNumber();
PNGImage getLeftImage(uint64_t frameNumber);
PNGImage getRightImage(uint64_t frameNumber);
void increasePNGChannel(PNGImage& image, int channel_offset,
                        int increase);
void mergePNGImages(PNGImage& right, const PNGImage& left);
```

```
void fig_2_28() {
  while (uint64_t frameNumber = getNextFrameNumber()) {
    auto left = getLeftImage(frameNumber);
    auto right = getRightImage(frameNumber);
    increasePNGChannel(left, PNGImage::redOffset, 10);
    increasePNGChannel(right, PNGImage::blueOffset, 10);
    mergePNGImages(right, left);
    right.write();
  }
}
```

(b) A serial implementation that reads images and applies the 3D effect.

Figure 2-28. *A red-cyan 3D stereoscopic sample application*

Similar to the simple case change sample, we again have a series of inputs that pass through a set of filters. We identify the important functions and convert them to pipeline filters: `getNextFrameNumber`, `getLeftImage`, `getRightImage`, `increasePNGChannel` (to left image), `increasePNGChannel` (to right image), `mergePNGImages`, and `right.write()`. Figure 2-29 shows the example drawn as a pipeline. The `increasePNGChannel` filter is applied twice, first on the left image and then on the right image.

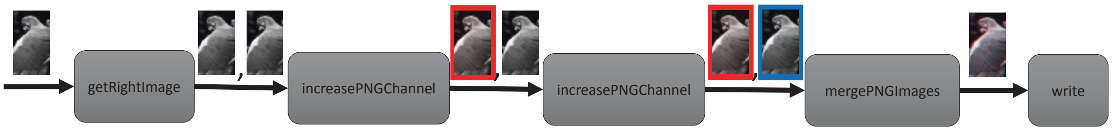


Figure 2-29. *The 3D stereoscopic sample application as a pipeline*

The parallel implementation using a TBB `parallel_pipeline` is shown in Figure 2-30.

```
void fig_2_30() {
  using Image = PNGImage;
  using ImagePair = std::pair<PNGImage, PNGImage>;
  tbb::parallel_pipeline(
    /* tokens */ 8,
    /* make the left image filter */
    tbb::make_filter<void, Image>(
      /* filter type */ tbb::filter::serial_in_order,
      [&](tbb::flow_control &fc) -> Image {
        if (uint64_t frame_number = getNextFrameNumber()) {
          return getLeftImage(frame_number);
        } else {
          fc.stop();
          return Image{};
        }
      }) &
    tbb::make_filter<Image, ImagePair>(
      /* filter type */ tbb::filter::serial_in_order,
      [&](Image left) -> ImagePair {
        return ImagePair(left, getRightImage(left.frameNumber));
      }) &
    tbb::make_filter<ImagePair, ImagePair>(
      /* filter type */ tbb::filter::parallel,
      [&](ImagePair p) -> ImagePair {
        increasePNGChannel(p.first, Image::redOffset, 10);
        return p;
      }) &
    tbb::make_filter<ImagePair, ImagePair>(
      /* filter type */ tbb::filter::parallel,
      [&](ImagePair p) -> ImagePair {
        increasePNGChannel(p.second, Image::blueOffset, 10);
        return p;
      }) &
    tbb::make_filter<ImagePair, Image>(
      /* filter type */ tbb::filter::parallel,
      [&](ImagePair p) -> Image {
        mergePNGImages(p.second, p.first);
        return p.second;
      }) &
    tbb::make_filter<Image, void>(
      /* filter type */ tbb::filter::parallel,
      [&](Image img) {
        img.write();
      })
  );
}
```

Figure 2-30. *The stereoscopic 3D example implemented using* `parallel_pipeline`

The TBB `parallel_pipeline` function imposes a linearization of the pipeline filters. The filters are applied one after the other as the input from the first stage flows through the pipeline. This is in fact a limitation for this sample. The processing of the left and right images is independent until the `mergeImageBuffers` filter, but because of the interface of `parallel_pipeline`, the filters must be linearized. Even so, only the filters that read in the images are serial filters, and therefore this implementation can still be scalable if the execution time is dominated by the later, parallel stages.

In Chapter 3, we introduce the TBB Flow Graph, which will allow us to more directly express applications that benefit from nonlinearized execution of filters.

Summary

This chapter offered a basic overview of the generic parallel algorithms provided by the TBB library, including patterns that capture functional parallelism, simple and complex loops, and pipeline parallelism. These prepackaged algorithms (patterns) provide well-tested and tuned implementations that can be applied incrementally to an application to improve performance.

The code shown in this chapter provides small examples that show how these algorithms can be used. In Part 2 of this book (starting with Chapter 9), we discuss how to get the most out of TBB by combining these algorithms in composable ways and tuning applications using the library features available for optimizing locality, minimizing overheads, and adding priorities. Part 2 of the book also discusses how to deal with exception handling and cancellation when using the TBB generic parallel algorithms.

We continue in the next chapter by taking a look at another one of TBB's high-level features, the Flow Graph.

For More Information

Here are some additional reading materials we recommend related to this chapter.

- We discussed design patterns for parallel programming and how these relate to the TBB generic parallel algorithms. A collection of design patterns can be found in

Timothy Mattson, Beverly Sanders, and Berna Massingill, *Patterns for Parallel Programming* (First ed.), 2004, Addison-Wesley Professional.

- When discussing the parallel implementation of quicksort, we noted that the partitioning was still a serial bottleneck. Papers that discuss parallel partitioning implementations include

  P. Heidelberger, A. Norton and J. T. Robinson, "Parallel Quicksort using fetch-and-add," in *IEEE Transactions on Computers*, vol. 39, no. 1, pp. 133-138, Jan 1990.

  P. Tsigas and Y. Zhang. A simple, fast parallel implementation of quicksort and its performance evaluation on SUN enterprise 10000. In 11th Euromicro Workshop on Parallel, Distributed and Network-Based Processing (PDP 2003), pages 372–381, 2003.

- You can learn more about data dependence analysis in a number of compiler or parallel programming books, including

  Michael Joseph Wolfe, *High-Performance Compilers for Parallel Computing,* 1995, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

  Kennedy and John R. Allen, *Optimizing Compilers for Modern Architectures,* 2001, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

- When we discussed matrix multiplication, we noted that unless we are optimization gurus, we should typically prefer to use prepackaged implementations of linear algebra kernels when available.

  Such packages include

  The Basic Linear Algebra Subprograms (BLAS) at `www.netlib.org/blas/`

  The Intel Math Kernel Library (Intel MKL) at `https://software.intel.com/mkl`

  Automatically Tuned Linear Algebra Software (ATLAS) found at `http://math-atlas.sourceforge.net/`

The FLAME project researches and develops dense linear algebra libraries. Their BLIS software framework can be used to create high-performance BLAS libraries. The FLAME project can be found at `www.cs.utexas.edu/~flame`.

- The line of sight example in this chapter was implemented using parallel scan based on the description provided in

  Vector Models for Data-Parallel Computing, Guy E. Blelloch (The MIT Press).

The photograph used in Figures 2-28a, 2-29, and 3-7, was taken by Elena Adams, and is used with permission from the Halide project's tutorials at `http://halide-lang.org`.