CHAPTER 4

TBB and the Parallel Algorithms of the C++ Standard Template Library

To use the Threading Building Blocks (TBB) library effectively, it is important to understand how it supports and augments the C++ standard. We discuss three aspects of TBB's relationship with standard C++ in this chapter:

1. The TBB library has often included parallelism-related features that are new to the C++ standard. Including such features in TBB provides developers early access to them before they are widely available in all compilers. In this vein, all pre-built distributions of TBB now include Intel's implementation of the parallel algorithms of the C++ Standard Template Library (STL). These implementations use TBB tasks to implement multithreading and use SIMD instructions to implement vectorization. Discussion of Parallel STL makes up the bulk of this chapter.

2. The TBB library also provides features that are not included in the C++ standard but make expressing parallelism easier for developers. The generic parallel algorithms and flow graph are examples of these. In this chapter, we discuss custom iterators that have been included in TBB to widen the applicability of the Parallel STL algorithms.

© Intel Corporation 2019
M. Voss, R. Asenjo, J. Reinders, *Pro TBB*, https://doi.org/10.1007/978-1-4842-4398-5_4

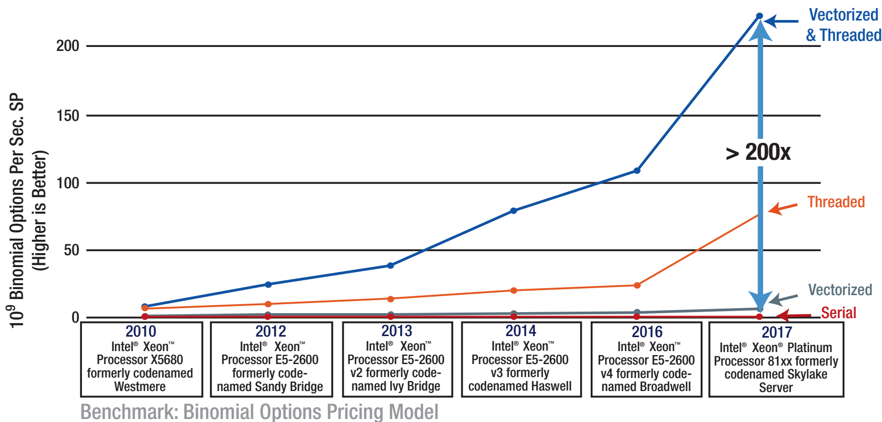
3. Finally, we note throughout the chapter that some additions to the C++ standard may displace the need for certain TBB features. However, we also note that TBB has value that will likely not be subsumed by the C++ standard for the foreseeable future. The features that TBB provides that will be of continuing benefit include its work-stealing task scheduler, its thread-safe containers, its flow graph API, and its scalable memory allocators for example.

Does the C++ STL Library Belong in This Book?

Does a chapter about additions to the C++ Standard Template Library really belong in a book about TBB? Yes, indeed it does! TBB is a C++ library for parallelism, and it does not exist in a vacuum. We need to understand how it relates to the C++ standard.

The execution policies we discuss in this chapter are similar in some ways to the TBB parallel algorithms covered in Chapter 2 because they let us express that an algorithm is safe to execute in parallel — but they ***do not*** prescribe the exact implementation details. If we want to mix TBB algorithms and Parallel STL algorithms in an application and still have efficient, composable parallelism (see Chapter 9), we benefit from using a Parallel STL implementation that uses TBB as its parallel execution engine! Therefore, when we talk about the parallel execution policies in this chapter, we will focus on TBB-based implementations. When we use a Parallel STL that uses TBB underneath, then Parallel STL becomes just another path for us to use TBB tasks in our code.

Back in Figure 1-3 in Chapter 1, we noted that many applications have multiple levels of parallelism available, including a Single Instruction Multiple Data (SIMD) layer that is best executed on vector units. Exploiting this level of parallelism can be critical as demonstrated by the performance results for the binomial options application shown in in Figure 4-1. Vector parallelism can only improve performance by a small factor when used alone; it's limited by the vector width. Figure 4-1 reminds us, however, that the multiplicative effect of using both task parallelism and vector parallelism should not be overlooked.



Benchmark results were obtained prior to implementation of recent software patches and firmware updates that are intended to address exploits referred to as "Spectre" and "Meltdown." Implementation of these updates may make these results inapplicable to your device or system.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations, and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information, visit Performance Benchmark Test Disclosure.

Figure 4-1. *The performance of a binomial options pricing application when executed serial, vectorized, threaded, and vectorized and threaded*

In Chapter 1, we implemented an example that used a top-level TBB flow graph layer to introduce threading, nested generic TBB parallel algorithms within the graph nodes to get more threading, and then nested STL algorithms that use vector policies in the bodies of the parallel algorithms to introduce vectorization. When we combine TBB with Parallel STL and its execution policies, we not only get composable messaging and fork-join layers, but we also gain access to the SIMD layer.

It is for these reasons that execution policies in the STL library are an important part of our exploration of TBB!

TBB AND THE C++ STANDARD

The team that develops TBB is a strong proponent of support for threading in the C++ language itself. In fact, TBB has often included parallelism features modeled after those proposed for standardization in C++ to allow developers to migrate to these interfaces before they are widely supported by mainstream compilers. An example of this is `std::thread`. The developers of TBB recognized the importance of `std::thread` and so made an implementation available as a migration path for developers before it was available in all of the C++ standard libraries, injecting the feature directly into the `std` namespace. Today, TBB's implementation of `std::thread` simply includes the platform's implementation of `std::thread` if available and only falls back to its own implementation if the platform's standard C++ library does not include an implementation. A similar story can be told for other now-standard C++ features like atomic variables, mutex objects, and `std::condition_variable`.

A Parallel STL Execution Policy Analogy

To help think about the different execution policies that are provided by the Parallel STL library, we can visualize a multilane highway as shown in Figure 4-2. As with most analogies, this is not perfect, but it can help us see the benefits of the different policies.

We can think of each lane in a multilane highway as a thread of execution, each person as an operation to accomplish (i.e., the person needs to get from point A to point B), each car as a processor core, and each seat in the car as an element in a (vector) register. In a *serial execution*, we only use a single lane of the highway (a single thread) and each person gets their own car (we are not using vector units). Whether people are traveling the same route or not, they each take their own car and all travel in the same lane.

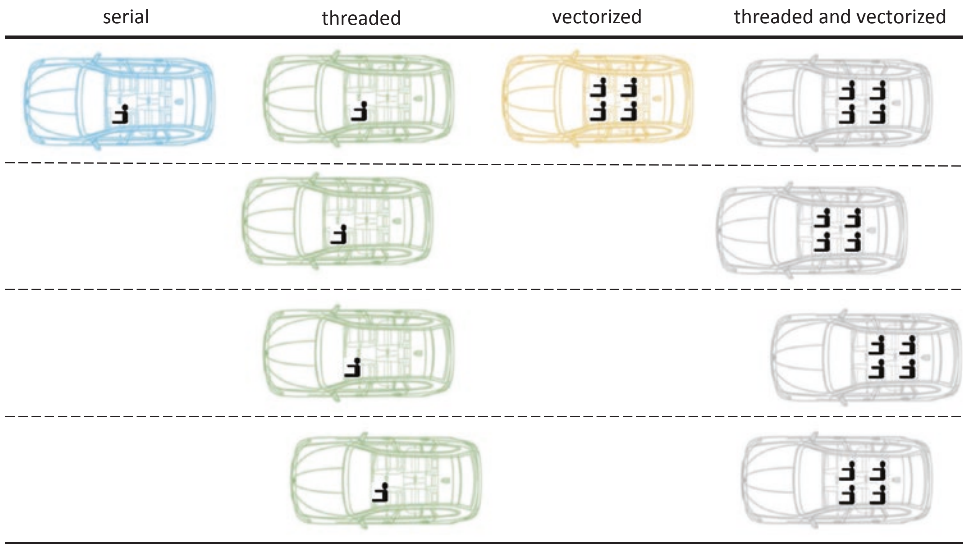


Figure 4-2. *A multilane highway analogy for the execution policies in Parallel STL*

In a *threaded execution*, we use more than one lane of the highway (i.e., more than one thread of execution). Now, we get more tasks accomplished per unit of time, but still, no carpooling is allowed. If several people are coming from the same starting point and traveling to the same destination, they each take their own car. We are more effectively utilizing the highway, but our cars (cores) are being used inefficiently.

A *vectorized execution* is like carpooling. If several people need to travel the same exact route, they share a car. Many modern processors support vector instructions, for example SSE and AVX in Intel processors. If we don't use vector instructions, we are underutilizing our processors. The vector units in these cores can apply the same operation to multiple pieces of data simultaneously. The data in vector registers are like people sharing a car, they take the same exact route.

Lastly, a threaded and vectorized execution is like using all the lanes in the highway (all of the cores) and also carpooling (using the vector units in each core).

A Simple Example Using `std::for_each`

Now that we have a general idea about execution policies but before we get into all of the gory details, let's start by applying a function `void f(float &e)` to all of the elements in a vector `v` as shown in Figure 4-3(a). Using `std::for_each`, one of the algorithms in the C++ STL library, we can do the same thing, as shown in Figure 4-3(b). Just like with

a range-based for, the for_each iterates from v.begin() to v.end() and invokes the lambda expression on each item in the vector. This is the default sequenced behavior of for_each.

With Parallel STL, however, we can inform the library that it is okay to relax these semantics in order to exploit parallelism or, as shown in Figure 4-3(c), we can make it explicitly known to the library that we want the sequenced semantics. When using Intel's Parallel STL, we need to include both the algorithms and the execution policy headers into our code, for example:

```
#include <pstl/execution>
#include <pstl/algorithm>
```

In C++17, leaving out the execution policy or passing in the sequenced_policy object, seq, results in the same default execution behavior: it *appears as if* the lambda expression is invoked on each item in the vector in order. We say "as if" because the hardware and compiler are permitted to parallelize the algorithm, but only if doing so is invisible to a standard-conforming program.

The power of Parallel STL comes from the other execution policies that relax this sequenced constraint. We say that the operations can be overlapped or vectorized from within a single thread of execution by using the unsequenced_policy object, unseq, as shown in Figure 4-3(d). The library can then overlap operations in a single thread, for example, by using Single Instruction Multiple Data (SIMD) extensions such as SSE or AVX to vectorize the execution. Figure 4-4 shows this behavior using side-by-side boxes to indicate that these operations execute simultaneously using vector units. The unseq execution policy allows "carpooling."

```
for (auto &i : v ) {
  f(i);
}
(a) a for loop

std::for_each(v.begin(), v.end(),
 [](float &i) {
   f(i);
 }
);
(b) std::for_each and no execution policy

std::for_each(pstl::execution::seq, v.begin(), v.end(),
  [](float &i) {
    f(i);
  }
);
(c) std::for_each and seq execution policy

std::for_each(pstl::execution::unseq, v.begin(), v.end(),
  [](float &i) {
    f(i);
  }
);
(d) std::for_each and unseq execution policy

std::for_each(pstl::execution::par, v.begin(), v.end(),
  [](float &i) {
    f(i);
  }
);
(e) std::for_each and par execution policy

std::for_each(pstl::execution::par_unseq, v.begin(), v.end(),
  [](float &i) {
    f(i);
  }
);
(f) std::for_each and par_unseq execution policy
```

Figure 4-3. *A simple loop implemented with `std::for_each` and using various Parallel STL execution policies*

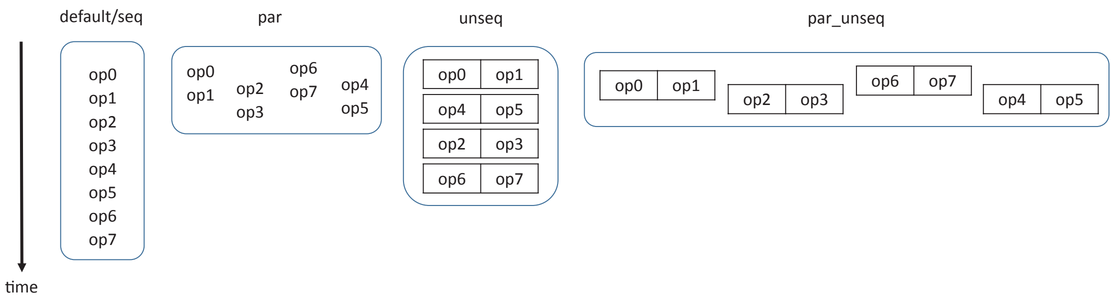


***Figure 4-4.** Applying operations using different execution policies*

In Figure 4-3(e), we tell the library that it is safe to execute this function on all of the elements in the vector using multiple threads of execution using the `parallel_policy` object, `par`. As shown in Figure 4-4, the `par` policy allows the operations to be spread across different threads of execution, but, within each thread, the operations are not overlapped (i.e., they are not vectorized). Thinking back to our multilane highway example, we are now using all of the lanes in the highway but are not yet carpooling.

Finally, in Figure 4-3(f), the `parallel_unsequenced_policy` object, `par_unseq`, is used to communicate that the application of the lambda expression to the elements can be both parallelized and vectorized. In Figure 4-4, the `par_unseq` execution uses multiple threads of execution *and* overlaps operations within each thread. We are now fully utilizing all of the cores in our platform and effectively utilizing each core by making use of its vector units.

In practice, we must be careful when we use execution policies. Just like with the generic TBB parallel algorithms, when we use an execution policy to relax the execution order for an STL algorithm, we are asserting to the library that this relaxation is legal and profitable. The library does not check that we are correct. Likewise, the library does not guarantee that performance is not degraded by using a certain execution policy.

Another point to notice in Figure 4-3 is that the STL algorithms themselves are in namespace `std`, but the execution policies, as provided by Intel's Parallel STL, are in namespace `pstl::execution`. If you have a fully compliant C++17 compiler, other implementations that may not use TBB will be selected if you use the standard execution policies in the `std::execution` namespace.

What Algorithms Are Provided in a Parallel STL Implementation?

The C++ Standard Template Library (STL) primarily includes operations that are applied to sequences. There are some outliers like `std::min` and `std::max` that can be applied to values, but for the most part, the algorithms, such as `std::for_each`, `std::find`, `std::transform`, `std::copy`, and `std::sort`, are applied to sequences of items. This focus on sequences is convenient when we want to operate on containers that support iterators, but can be somewhat cumbersome if we want to express something that does not operate on containers. Later in this chapter, we will see that sometimes we can "think outside the box" and use custom iterators to make some algorithms act more like general loops.

Explaining what each STL algorithm does is beyond the scope of this chapter and book. There are entire books written about the C++ Standard Template Library and how to use it, including *The C++ Standard Library: A Tutorial and Reference by Nicolai Josuttis (Addison-Wesley Professional)*. In this chapter, we only focus on what the execution policies, first introduced in C++17, mean for these algorithms and how they can be used together with TBB.

Most of the STL algorithms specified in the C++ standard have overloads in C++17 that accept execution polices. In addition, a few new algorithms were added because they are especially useful in parallel programs or because the committee wanted to avoid changes in semantics. We can find exactly which algorithms support execution policies by looking at the standard itself or online at web sites like `http://en.cppreference.com/w/cpp/algorithm`.

How to Get and Use a Copy of Parallel STL That Uses TBB

Detailed instructions for downloading and installing Intel's Parallel STL are provided in Chapter 1 in the section "Getting the Threading Building Blocks (TBB) Library." If you download and install a pre-built copy of TBB 2018 update 5 or later, whether a commercially license copy obtained through Intel or an open-source binary distribution downloaded from GitHub, then you also get Intel's Parallel STL. Parallel STL comes with all of the pre-built TBB packages.

If however, you want to build the TBB library from sources obtained from GitHub, then you will need to download the Parallel STL sources from GitHub separately, since the source code for the two libraries are maintained in separate repositories `https://github.com/intel/tbb` and `https://github.com/intel/parallelstl`.

As we have already seen, Parallel STL supports several different execution policies, some that support parallelized execution, some that support vectorized execution, and some that support both. Intel's Parallel STL supports parallelism with TBB and vectorization using OpenMP 4.0 SIMD constructs. To get the most out of Intel's Parallel STL, you must have a C++ compiler that supports C++11 and OpenMP 4.0 SIMD constructs – and of course you also need TBB. We strongly recommend using the Intel compiler that comes with any version of Intel Parallel Studio XE 2018 or later. Not only do these compilers include both the TBB library and support OpenMP 4.0 SIMD constructs, but they also include optimizations that were specifically included to increase performance for some C++ STL algorithms when used with the `unseq` or `par_unseq` execution policies.

To build an application that uses Parallel STL on the command line, we need to set the environment variables for compilation and linkage. If we installed Intel Parallel Studio XE, we can do this by calling the suite-level environment scripts such as `compilervars.{sh|csh|bat}`. If we just installed Parallel STL, then we can set the environment variables by running `pstlvars.{sh|csh|bat}` in `<pstl_install_dir>/{linux|mac|windows}/pstl/bin`. Additional instructions are provided in Chapter 1.

Algorithms in Intel's Parallel STL

Intel's Parallel STL does not yet support all execution policies for every single STL algorithm. An up-to-date list of which algorithms are provided by the library and which policies each algorithm supports can be found at `https://software.intel.com/en-us/get-started-with-pstl`.

Figure 4-5 shows the algorithms and execution policies that were supported at the time this book was written.

Algorithm Categories	
Non-modifying sequence operations	**header: <algorithm>**
`adjacent_find, all_of, any_of, count, count_if, find, find_end, find_first_of, find_if, find_if_not, for_each, for_each_n, mismatch, none_of, search, search_n`	
Modifying sequence operations	**header: <algorithm>**
`copy, copy_if, copy_n, fill, fill_n, generate, generate_n, move, remove, remove_copy, remove_copy_if, remove_if, replace, replace_copy, replace_copy_if, replace_if, reverse, reverse_copy, rotate, rotate_copy, shuffle, swap_ranges, transform, unique*, unique_copy`	
Partitioning operations	**header: <algorithm>**
`is_partitioned, partition*, partition_copy, stable_partition*`	
Sorting operations	**header: <algorithm>**
`is_sorted, is_sorted_until, nth_element*, partial_sort*, partial_sort_copy*, sort*, stable_sort*`	
Operations on sorted ranges	**header: <algorithm>**
`includes*, inplace_merge*, merge*, set_difference*, set_intersection*, set_symmetric_difference*, set_union*`	
Heap operations	**header: <algorithm>**
`is_heap, is_heap_until`	
Minimum/maximum operations	**header: <algorithm>**
`max_element, min_element, minmax_element`	
Comparison operations	**header: <algorithm>**
`equal, lexicographical_compare`	
Numeric operations	**header: <numeric>**
`adjacent_difference, exclusive_scan, inclusive_scan, reduce, transform_exclusive_scan, transform_inclusive_scan, transform_reduce`	
Operations on uninitialized memory	**header: <memory>**
`uninitialized_copy, uninitialized_copy_n, uninitialized_default_construct, uninitialized_default_construct_n, uninitialized_fill, uninitialized_fill_n, uninitialized_move, uninitialized_move_n, uninitialized_value_construct, uninitialized_value_construct_n`	

*** Supports threading but not SIMD execution**

Figure 4-5. *The algorithms that support execution policies in Intel's Parallel STL as of January 2019. Additional algorithms and policies may be supported later. See* `https://software.intel.com/en-us/get-started-with-pstl` *for updates.*

Figure 4-6 shows the policies supported by Intel's Parallel STL including those that are part of the C++17 standard as well as those that have been proposed for inclusion in a future standard. The C++17 policies let us select a sequenced execution (`seq`), a parallel execution using TBB (`par`), or a parallel execution using TBB that is also vectorized (`par_unseq`). The unsequenced (`unseq`) policy let us select an implementation that is only vectorized.

Policy	Standard	Class	Global Object
sequenced	C++17	sequenced_policy	seq
parallel	C++17	parallel_policy	par
parallel + unsequenced	C++17	parallel_unsequenced_policy	par_unseq
unsequenced	Proposed	unsequenced_policy	unseq

Figure 4-6. *The execution policies supported by Intel's Parallel STL*

Capturing More Use Cases with Custom Iterators

Earlier in this chapter, we introduced a simple use of `std::for_each` and showed how the different execution policies can be used with it. Our simple example with `par_unseq` in Figure 4-3(f) looked like

```
std::for_each(pstl::execution::par_unseq, v.begin(), v.end(),
  [](float &i) {
    f(i);
  }
);
```

At first glance, the `for_each` algorithm appears fairly limited, it visits the elements in a sequence and applies a unary function to each element. When used on a container in this expected way, it is in fact limited in applicability. It does not, for example, accept a range like the TBB `parallel_for`.

However, C++ is a powerful language, and we can use STL algorithms in creative ways to stretch their applicability. As we discussed earlier in Chapter 2, 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. Many standard C++ containers provide `begin` and `end` functions that return iterators that let us traverse the elements of the containers. One common way to apply STL algorithms to more use cases is by using *custom* iterators. These classes implement the iterator interface but are not included in the C++ Standard Template Library.

Three commonly used custom iterators are included in the TBB library to assist in using the STL algorithms. These iterator types are described in Figure 4-7 and are available in the `iterators.h` header or through the all-inclusive `tbb.h` header file.

Type	Description
`template<typename IntType> class counting_iterator;`	A random access iterator that maintains an internal counter that changes based on the arithmetic operations performed on the iterator. The `operator*()` function returns the current counter value. It is commonly used to obtain an index into a container or set of containers.
`template<typename... Types> class zip_iterator;`	A random access iterator that iterates over several iterators simultaneously. The `operator*()` function returns a tuple of the dereferenced values of the individual iterators.
`template<typename UnaryFunc, typename Iter> class transform_iterator;`	A random access iterator that applies a transformation to a sequence. The `operator*()` function applies a unary transformation function to the value returned by dereferencing the underlying iterator.

Figure 4-7. *The custom iterator classes available in TBB*

As an example, we can pass custom iterators to `std::for_each` to make it act more like a general `for` loop. Let's consider the simple loop shown in Figure 4-8(a). This loop writes `a[i]+b[i]*b[i]` back to `a[i]` for each `i` in the range `[0,n)`.

```
std::vector<float> a(n, 1.0), b(n, 3.0);

for (int i = 0; i < n; ++i) {
  a[i] = a[i] + b[i]*b[i];
}
```
(a) The initial serial for loop

```
std::for_each(tbb::counting_iterator<int>(0),
              tbb::counting_iterator<int>(n),
  [&a, &b](int i) {
    a[i] = a[i] + b[i]*b[i];
  });
```
(b) Using tbb::counting_iterator

```
auto begin = tbb::make_zip_iterator(a.begin(), b.begin());
auto end = tbb::make_zip_iterator(a.end(), b.end());

std::for_each(begin, end,
  [](const std::tuple<float&, float&>& v) {
    float &a = std::get<0>(v);
    float b = std::get<1>(v);
    a = a + b*b;
  });
```
(c) Using tbb::zip_iterator

```
auto zbegin = tbb::make_zip_iterator(a.begin(), b.begin());
auto zend = tbb::make_zip_iterator(a.end(), b.end());
auto square_b = [](const std::tuple<float &, float &>& v) {
  float b = std::get<1>(v);
  return std::tuple<float &, float>(std::get<0>(v), b*b);
};
auto begin = tbb::make_transform_iterator(zbegin, square_b);
auto end = tbb::make_transform_iterator(zend, square_b);

std::for_each(begin, end,
  [](const std::tuple<float&, float>& v) {
    float &a = std::get<0>(v);
    a = a + std::get<1>(v);
  });
```
(d) Using tbb::transform_iterator

Figure 4-8. *Using `std::for_each` with custom iterators*

In Figure 4-8(b), the counting_iterator class is used to create something like a range. The arguments passed to the for_each lambda expression will be the integer values from 0 to n-1. Even though the for_each still iterates over only a single sequence, we use these values as an index into the two vectors, a and b.

In Figure 4-8(c), the zip_iterator class is used to iterate over the a and b vectors simultaneously. The TBB library provides a make_zip_iterator function to simplify construction of the iterators:

```
template<typename... T>
zip_iterator<T...> make_zip_iterator(T&&... args);
```

In Figure 4-8(c), we still use only a single sequence in the call to for_each. But now, the argument passed to the lambda expression is a std::tuple of references to float, one from each vector.

Finally, in Figure 4-8(d), we add uses of the transform_iterator class. We first combine the two sequences from vector a and b into a single sequence using the zip_iterator class, like we did in Figure 4-8(c). But, we also create a lambda expression, which we assign to square_b. The lambda expression will be used to transform the std::tuple of references to float that are obtained by dereferencing the zip_iterator. We pass this lambda expression to our calls to the make_tranform_iterator function:

```
template<typename UnaryFunc, typename Iter>
transform_iterator<UnaryFunc, Iter>
make_transform_iterator(Iter it, UnaryFunc unary_func);
```

When the transform_iterator objects in Figure 4-8(d) are dereferenced, they receive an element from the underlying zip_iterator, square the second element of the tuple, and create a new std::tuple that contains a reference to the float from a and the squared value from b. The argument passed to the for_each lambda expression includes an already squared value, and so the function does not need to compute b[i]*b[i].

Because custom iterators like those in Figure 4-7 are so useful, they are not only available in the TBB library but also through other libraries such as the Boost C++ libraries (www.boost.org) and Thrust (https://thrust.github.io/doc/group__fancyiterator.html). They are currently not available directly in the C++ Standard Template Library.

Highlighting Some of the Most Useful Algorithms

With the preliminaries out of the way, we can now discuss the more useful and general algorithms provided by Parallel STL in more depth, including `for_each`, `transform`, `reduce`, and `transform_reduce`. As we discuss each algorithm, we point out analogs in the TBB generic algorithms. The advantage of the Parallel STL interfaces over the TBB-specific interfaces is that Parallel STL is part of the C++ standard. The disadvantage of the Parallel STL interfaces is that they are less expressive and less tunable than the generic TBB algorithms. We point out some of these drawbacks as we talk about the algorithms in this section.

`std::for_each`, `std::for_each_n`

We've already talked a lot about `for_each` in this chapter. In addition to `for_each`, Parallel STL also provides a `for_each_n` algorithm that only visits the first `n` elements. Both algorithms `for_each` and `for_each_n` have several interfaces; the ones that accept execution policies are as follows:

```
template<class ExecutionPolicy, class ForwardIt,
         class UnaryFunction>
void for_each( ExecutionPolicy&& policy,
               ForwardIt first, ForwardIt last,
               UnaryFunction f);

template<class ExecutionPolicy, class ForwardIt, class Size,
         class UnaryFunction>
ForwardIt for_each_n(ExecutionPolicy&& policy,
                     ForwardIt first, Size n, UnaryFunction f);
```

Combined with custom iterators, `for_each` can be quite expressive, as we demonstrated earlier in Figure 4-8. We can, for example, take the simple matrix multiplication example from Chapter 2 and re-implement it in Figure 4-9 using the `counting_iterator` class.

```
std::for_each(
  /* policy */ pstl::execution::par,
  /* first */  tbb::counting_iterator<int>(0),
  /* last */   tbb::counting_iterator<int>(M),
  [&a, &b, &c, 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 4-9. *Using `std::for_each` with `tbb::counting_iterator` to create a parallel version of matrix multiplication*

If we use an STL that uses TBB underneath, like Intel's Parallel STL, the `par` policy is implemented using a `tbb::parallel_for`, and so the performance of `std::for_each` and `tbb::parallel_for` will be similar for a simple example like this.

This of course begs a question. If `std::for_each` uses a `tbb::parallel_for` to implement its `par` policy but is a standard interface and also gives us access to the other policies, shouldn't we just *always* use `std::for_each` instead of a `tbb::parallel_for`?

Unfortunately, no. Not all code is as simple as this example. If we are interested in an effective threaded implementation, it's typically better to use a `tbb::parallel_for` directly. Even for this matrix multiplication example, as we noted back in Chapter 2, our simple implementation is not optimal. In Part 2 of this book, we discuss important optimization hooks available in TBB that we can use to tune our code. We will see in Chapter 16 that these hooks result in significant performance gains. Unfortunately, most of these advanced features cannot be applied when we use a Parallel STL algorithm. The standard C++ interfaces simply do not allow for them.

When we use a Parallel STL algorithm and choose a standard policy such as `par`, `unseq`, or `par_unseq`, we get whatever the implementation decides to give us. There are proposals for additions to C++, like executors, that may address this limitation sometime in the future. But for now, we have little control over STL algorithms. When using TBB generic algorithms, such as `parallel_for`, we have access to the rich set of optimization features described in Part 2 of this book, such as partitioners, different types of blocked ranges, grainsizes, affinity hints, priorities, isolation features, and so on.

For some simple cases, a standard C++ Parallel STL algorithm might be just as good as its TBB counterpart, but in more realistic scenarios, TBB provides us with the flexibility and control we need to get the performance we want.

std::transform

Another useful algorithm in Parallel STL is `transform`. It applies a unary operation to the elements from one sequence or a binary operation to the elements from two input sequences and writes the results to the elements in a single output sequence. The two interfaces that support parallel execution policies are as follows:

```
template< class ExecutionPolicy, class ForwardIt1, class ForwardIt2,
          class UnaryOperation >
ForwardIt2 transform(ExecutionPolicy&& policy, ForwardIt1 first1,
                     ForwardIt1 last1, ForwardIt2 d_first,
                     UnaryOperation unary_op);

template< class ExecutionPolicy, class ForwardIt1, class ForwardIt2,
          class ForwardIt3, class BinaryOperation >
ForwardIt3 transform(ExecutionPolicy&& policy, ForwardIt1 first1,
                     ForwardIt1 last1, ForwardIt2 first2,
                     ForwardIt3 d_first,
                     BinaryOperation binary_op);
```

In Figure 4-8, we used `for_each` and custom iterators to read from two vectors and write back to a single output vector, computing `a[i] = a[i] + b[i]*b[i]` in each iteration. This is a great candidate for `std::transform` as we can see in Figure 4-10. Because `transform` has an interface that supports two input sequences and one output sequence, this matches our example well.

```
std::vector<float> a(n, 1.0), b(n, 3.0);

std::transform(pstl::execution::par_unseq,
  /* in1 range */ a.begin(), a.end(),
  /* in2 first */ b.begin(),
  /* out first */ a.begin(),
  [](float ae, float be) -> float {
    return ae + be*be;
  }
);
```

Figure 4-10. *Using `std::transform` to add two vectors*

As with std::for_each, the applicability of this algorithm when used in a typical way is limited because there is at most two input sequences and only a single output sequence. If we have a loop that writes to more than one output array or container, it's awkward to express that with a single call to transform. Of course, it's possible – almost anything is in C++ – but it requires using custom iterators, like zip_iterator, and some pretty ugly code to access the many containers.

std::reduce

We discussed reductions when we covered tbb::parallel_reduce in Chapter 2. The Parallel STL algorithm reduce performs a reduction over the elements of a sequence. Unlike tbb::parallel_reduce however, it provides only a reduction operation. In the next section, we discuss transform_reduce, which is more like tbb::parallel_reduce because it provides both a transform operation and a reduce operation. The two interfaces to std::reduce that support parallel execution policies are as follows:

```
template<class ExecutionPolicy, class ForwardIt, class T>
T reduce(ExecutionPolicy&& policy,
         ForwardIt first, ForwardIt last, T init);

template<class ExecutionPolicy, class ForwardIt, class T,
         class BinaryOp>
T reduce(ExecutionPolicy&& policy, ForwardIt first,
         ForwardIt last, T init, BinaryOp binary_op);
```

The reduce algorithm performs a generalized sum of the elements of the sequence using binary_op and the identity value init. In the first interface, binary_op is not an input parameter, and std::plus<> is used by default. A generalized sum means a reduction where the elements can be grouped and rearranged in arbitrary order – so this algorithm assumes that the operation is both associative and commutative. Because of this, we can have the same floating-point rounding issues that we discussed in the sidebar in Chapter 2.

If we want to sum the elements in a vector, we can use std::reduce and any of the execution policies, as shown in Figure 4-11.

```
std::vector<float> a(n, 1.0);

float sum = std::reduce(pstl::execution::seq, a.begin(), a.end());
sum += std::reduce(pstl::execution::unseq, a.begin(), a.end());
sum += std::reduce(pstl::execution::par, a.begin(), a.end());
sum += std::reduce(pstl::execution::par_unseq, a.begin(), a.end());
```

Figure 4-11. *Using `std::reduce` to sum the elements of a vector four times*

std::transform_reduce

As mentioned in the previous section, `transform_reduce` is similar to a `tbb::parallel_reduce` because it provides both a transform operation and reduce operation. However, as with most STL algorithms, it can be applied to only one or two input sequences at a time:

```
template<class ExecutionPolicy, class ForwardIt,
class T, class BinaryOp, class UnaryOp>
T transform_reduce(ExecutionPolicy&& policy,
                   ForwardIt first, ForwardIt last,
                   T init, BinaryOp binary_op, UnaryOp unary_op);

template<class ExecutionPolicy,
         class ForwardIt1, class ForwardIt2,
         class T, class BinaryOp1, class BinaryOp2>
T transform_reduce(ExecutionPolicy&& policy,
                   ForwardIt1 first1, ForwardIt1 last1,
                   ForwardIt2 first2, T init,
                   BinaryOp1 binary_op1, BinaryOp2 binary_op2);
```

An important and common kernel we can implement with a `std::transform_reduce` is an inner product. It is so commonly used for this purpose that there is an interface that uses `std::plus<>` and `std::multiplies<>` for the two operations by default:

```
template<class ExecutionPolicy,
         class ForwardIt1, class ForwardIt2, class T>
T transform_reduce(ExecutionPolicy&& policy,
                   ForwardIt1 first1, ForwardIt1 last1,
                   ForwardIt2 first2, T init);
```

The serial code for an inner product of two vectors, a and b, is shown in Figure 4-12(a). We can use a `std::transform_reduce` and provide our own lambda expressions for the two operations as shown in Figure 4-12(b). Or, like in Figure 4-12(c), we can rely on the default operations.

```
float v = 0.0;
for (int i = 0; i < n; ++i) {
  v += a[i]*b[i];
}
(a) serial code for an inner product of two vectors

std::transform_reduce(pstl::execution::par,
  /* in1 range */ a.begin(), a.end(),
  /* in2 range */ b.begin(),
  /* init */ 0.0,
  /* op1, the reduce */
  [](float ae, float be) -> float {
    return ae + be;
  },
  /* op2, the transform */
  [](float ae, float be) -> float {
    return ae * be;
  }
);
(b) std::transform_reduce with both operations specified

std::transform_reduce(pstl::execution::par,
  /* in1 range */ a.begin(), a.end(),
  /* in2 range */ b.begin(),
  /* init */ 0.0
);
(c) std::transform_reduce with default operations
```

Figure 4-12. *Using `std::transform_reduce` to calculate an inner product*

And again, as with the other Parallel STL algorithms, if we think slightly outside of the box, we can use custom iterators, like `counting_iterator`, to use this algorithm to process more than just elements in containers. For example, we can take the calculation of pi example that we implemented with `tbb::parallel_reduce` in Chapter 2 and implement it using a `std::transform_reduce`, as shown in Figure 4-13.

```
float fig_4_13() {
  constexpr float dx = 1.0 / num_intervals;
  float sum = std::transform_reduce(
    /* policy */ std::execution::par_unseq,
    /* first */ tbb::counting_iterator<int>(0),
    /* last */  tbb::counting_iterator<int>(num_intervals),
    /* init = */ 0.0,
    /* reduce */
    [](float x, float y) -> float {
      return x + y;
    },
    /* transform */
    [=](int i) -> float {
      float x = (i + 0.5)*dx;
      float h = sqrt(1 - x*x);
      return h*dx;
    }
  );
  return 4 * sum;
}
```

Figure 4-13. *Using `std::transform_reduce` to calculate pi*

Using a Parallel STL algorithm like `std::transform_reduce` instead of a `tbb::parallel_reduce` carries with it the same pros and cons as the other algorithms we've described. It uses a standardized interface so is potentially more portable. However, it doesn't allow us to use the optimization features that are described in Part 2 of this book to optimize its performance.

A Deeper Dive into the Execution Policies

The execution policies in Parallel STL let us communicate the constraints that we want to apply to the ordering of the operations during the execution of the STL algorithm. The standard set of policies did not come out of thin air – it captures the relaxed constraints necessary for executing efficient parallel/threaded or SIMD/vectorized code.

If you are happy enough to think of the `sequenced_policy` as meaning sequential execution, the `parallel_policy` as meaning parallel execution, the `unsequenced_policy` as meaning vectorized execution, and the `parallel_unsequenced_policy` as meaning parallel and vectorized execution, then you can skip the rest of this section. However, if you want to understand the subtleties implied by these policies, keep reading as we dive into the details.

The sequenced_policy

The sequenced_policy means that an algorithm's execution *appears as if* (1) all of the *element access functions* used by the algorithm are invoked on the thread that called the algorithm and (2) the invocations of the element access functions are not interleaved (i.e., they are sequenced with respect to each other within a given thread). An element access function is any function invoked during the algorithm that accesses the elements, such as functions in the iterators, comparison or swap functions, and any other user-provided functions that are applied to the elements. As mentioned earlier, we say "as if" because the hardware and compiler are permitted to break these rules, but only if doing so is invisible to a standard-conforming program.

One thing to note is that many of the STL algorithms do not specify that operations are applied in any specific sequence order even in the sequenced case. For example, while std::for_each does specify that the elements of a sequence are accessed in order in the sequenced case, std::transform does not. The std::transform visits all of the elements in a sequence, but not in any particular order. Unless stated otherwise, a sequenced execution means that the invocations of the element access functions are *indeterminately sequenced* in the calling thread. If two function calls are "indeterminately sequenced," it means that one of the function calls executes to completion before the other function call starts executing – but it doesn't matter which function call goes first. The result is that the library may not be able to interleave the execution of the operations from the two functions, preventing the use of SIMD operations for example.

The "as if" rule can sometimes lead to unexpected performance results. For example, a sequenced_policy execution may perform just as well as an unsequenced_policy because the compiler vectorizes both. If you get confusing results, you may want to inspect your compiler's optimization reports to see what optimizations have been applied.

The parallel_policy

The parallel_policy allows the element access functions to be invoked in the calling thread or from other threads created by the library to assist in parallel execution. However, any calls from within the same thread are sequenced with respect to each other, that is, the execution of access functions on the same thread cannot be interleaved.

When we use Intel's Parallel STL library, the `parallel_policy` is implemented using TBB generic algorithms and tasks. The threads that execute the operations are the main thread and the TBB worker threads.

The `unsequenced_policy`

The `unsequenced_policy` asserts that all of the element access functions must be invoked from the calling thread. However, within the calling thread, the execution of these functions can be interleaved. This relaxation of the sequenced constraint is important since it allows the library to aggregate operations in different function invocations into single SIMD instructions or to overlap operations.

SIMD parallelism can be implemented with vector instructions introduced through assembly code, compiler intrinsics, or compiler pragmas. In Intel's Parallel STL implementation, the library uses OpenMP SIMD pragmas.

Because the executions of the element access functions can be interleaved in a single thread, it is unsafe to use mutex objects in them (mutex objects are described in more detail in Chapter 5). Imagine, for example, interleaving several lock operations from different functions before executing any of the matching unlock operations.

The parallel_unsequenced_policy

As we might guess after learning about the preceding policies, the `parallel_unsequenced_policy` weakens execution constraints in two ways: (1) element access functions may be invoked by the calling thread or by other threads created to help with parallel execution and (2) the function executions within each thread may be interleaved.

Which Execution Policy Should We Use?

When we choose an execution policy, we first have to be sure it doesn't relax constraints to a point that the values computed by the algorithm will be wrong.

For example, we can use a `std::for_each` to compute `a[i] = a[i] + a[i-1]` for a vector `a`, but the code depends on the sequenced order of `for_each` (which, unlike some other indeterminately sequenced algorithms, applies the operator to the items in order):

```
float previous_value = 0.0;
std::for_each(
  pstl::execution::par, a.begin(), a.end(),
    [&](float &in)  {
      float p = previous_value;
      previous_value = in;
      in += p;
  }
);
```

This sample stores the last value into the previous_value variable, which has been captured by reference by the lambda expression. This sample only works if we execute the operations in order within a single thread of execution. Using any of the policy objects other than seq will yield incorrect results.

But let's assume we do our due diligence and we know which policies are legal for our operations and the STL algorithm we are using. How do we choose between a sequenced_policy execution, an unsequenced_policy execution, a parallel_policy execution, or a parallel_unsequenced_policy execution?

Unfortunately, there's not a simple answer. But there are some guidelines we can use:

- We should use threaded execution only when the algorithm has enough work to profit from parallel execution. We discuss rules of thumb for when to use tasks or not in Chapter 16 in Part 2 of this book. These rules apply here as well. A parallel execution has some overhead, and if the work isn't large enough we will only add overhead with no performance gain.
- Vectorization has lower overhead and therefore can be used effectively for small, inner loops. Simple algorithms may benefit from vectorization when they do not benefit from threading.
- Vectorization can have overhead too though. To use vector registers in a processor, the data has to be packed together. If our data is not contiguous in memory or we cannot access it with a unit stride, the compiler may have to generate extra instructions to gather the data into the vector registers. In such cases, a vectorized loop may perform worse than its sequential counterpart. You should read compiler vectorization reports and look at runtime profiles to makes sure you don't make things worse by adding vectorization.

- Because we can switch policies easily with Parallel STL, the best option may be to profile your code and see which policy performs best for your platform.

Other Ways to Introduce SIMD Parallelism

Outside of using the parallel algorithms in the C++ STL, there are several ways to introduce SIMD parallelism into applications. The easiest, and preferred route, is to use optimized domain-specific or math-kernel libraries whenever possible. For example, the Intel Math Kernel Library (Intel MKL) provides highly tuned implementations of many math functions, like those found in BLAS, LAPACK, and FFTW. These functions take advantage of both threading and vectorization when profitable – so if we use these, we get both threading and vectorization for free. Free is good! Intel MLK supports TBB-based execution for many of its functions, so if we use these TBB versions they will compose well with the rest of our TBB-based parallelism.

Of course, we might need to implement algorithms that are not available in any prepackaged library. In that case, there are three general approaches to adding vector instructions: (1) inline assembly code, (2) `simd` intrinsics, and (3) compiler-based vectorization.

We can use inline assembly code to inject specific assembly instructions, including vector instructions, directly into our applications. This is a low-level approach that is both compiler and processor dependent and therefore is the least portable and most error-prone. But, it does give us complete control over the instructions that are used (for better or worse). We use this approach as a last resort!

An only slightly better approach is to use SIMD intrinsics. Most compilers provide a set of intrinsic functions that let us inject platform-specific instructions without resorting to inline assembly code. But other than making it easier to inject the instructions, the end result is still compiler and platform dependent and error-prone. We generally avoid this approach too.

The final approach is to rely on compiler-based vectorization. At one extreme, this can mean fully automated vectorization, where we turn on the right compiler flags, let the compiler do its thing, and hope for the best. If that works, great! We get the benefits

of vectorization for free. Remember, free is a good thing. However, sometimes we need to give guidance to the compiler so that it can (or will) vectorize our loops. There are some compiler specific ways to provide guidance, such as the Intel compiler's `#pragma ivdep` and `#pragma vector always` and some standardized approaches, such as using the OpenMP `simd` pragmas. Both fully automatic and user-guided compiler vectorization are much more portable than inserting platform-specific instructions directly into our code through inline assembly code or compiler intrinsics. In fact, even Intel's Parallel STL library uses OpenMP `simd` pragmas to support vectorization in a portable way for the `unseq` and `parallel_unseq` policies.

We provide links to learn more about the options for adding vector instructions in the "For More Information" section at the end of this chapter.

Summary

In this chapter, we provided an overview of Parallel STL, what algorithms and execution policies it supports, and how to get a copy that uses Threading Building Blocks as its execution engine. We then discussed the custom iterator classes provided by TBB that increase the applicability of the STL algorithms. We continued by highlighting some of the most useful and general algorithms for parallel programming: `std::for_each`, `std::transform`, `std::reduce`, and `std::transform_reduce`. We demonstrated that some of the samples we implemented in Chapter 2 can also be implemented with these algorithms. But we also warned that the STL algorithms are still not as expressive as TBB and that the important performance hooks we discuss in Part 2 of this book cannot be used with Parallel STL. While Parallel STL is useful for some simple cases, its current limitations make us hesitant to recommend it broadly for threading. That said, TBB tasks are not a path to SIMD parallelism. The `unseq` and `parallel_unseq` policies provided by Intel's Parallel STL, which is included with all of the recent TBB distributions, augment the threading provided by TBB with support for easy vectorization.

For More Information

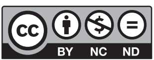
Vladimir Polin and Mikhail Dvorskiy, "Parallel STL: Boosting Performance of C++ STL Code: C++ and the Evolution Toward Parallelism," The Parallel Universe Magazine, Intel Corporation, Issue 28, pages 5–18, 2017.

Alexey Moskalev and Andrey Fedorov, “Getting Started with Parallel STL,” https://software.intel.com/en-us/get-started-with-pstl, March 29, 2018.

Pablo Halpern, Arch D Robison, Robert Geva, Clark Nelson and Jen Maurer, “Vector and Wavefront Policies,” Programming Language C++ (WG21), P0076r3, http://open-std.org/JTC1/SC22/WG21/docs/papers/2016/p0076r3.pdf, July 7, 2016.

The Intel 64 and IA-32 Architectures Software Developer Manuals: https://software.intel.com/en-us/articles/intel-sdm.

The Intel Intrinsics Guide: https://software.intel.com/sites/landingpage/IntrinsicsGuide/.



Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.