CHAPTER 1

Jumping Right In: "Hello, TBB!"

Over 10 years after its first release, the Threading Building Blocks (TBB) library has become one of the most widely used C++ libraries for parallel programming. While it has retained its core philosophy and features, it continues to expand to address new opportunities and challenges as they arise.

In this chapter, we discuss the motivations for TBB, provide a brief overview of its main components, describe how to get the library, and then jump right into a few simple examples.

Why Threading Building Blocks?

Parallel programming has a long history, stretching back to the 1950s and beyond. For decades, scientists have been developing large-scale parallel simulations for supercomputers, and businesses have been developing enterprise applications for large multiprocessor mainframes. But a little over 10 years ago, the first multicore chips intended for desktops and laptops started to enter the marketplace. This was a game changer.

The number of processors in these first multicore desktop and laptop systems was small – only two cores – but the number of developers that had to become parallel programmers was huge. If multicore processors were to become mainstream, parallel programming had to become mainstream too, especially for developers who care about performance.

The TBB library was first released in September of 2006 to address the unique challenges of mainstream parallel programming. Its goal now, and when it was first introduced over 10 years ago, is to provide an easy and powerful way for developers to

© Intel Corporation 2019
M. Voss, R. Asenjo, J. Reinders, *Pro TBB*, https://doi.org/10.1007/978-1-4842-4398-5_1

build applications that continue to scale as new platforms, with different architectures and more cores, are introduced. This "future-proofing" has paid off as the number of cores in mainstream processors has grown from two in 2006 to more than 64 in 2018!

To achieve this goal of future-proofing parallel applications against changes in the number and capabilities of processing cores, the key philosophy behind TBB is to make it easy for developers to express the parallelism in their applications, while limiting the control they have over the mapping of this parallelism to the underlying hardware. This philosophy can seem counterintuitive to some experienced parallel programmers. If we believe parallel programming must get maximum performance at all costs by programming to the bare metal of a system, and hand-tuning and optimizing applications to squeeze out every last bit of performance, then TBB may not be for us. Instead, the TBB library is for developers that want to write applications that get great performance on today's platforms but are willing to give up a little performance to ensure that their applications continue to perform well on future systems.

To achieve this end, the interfaces in TBB let us express the parallelism in our applications but provide flexibility to the library so it can effectively map this parallelism to current and future platforms, and to adapt it to dynamic changes in system resources at runtime.

Performance: Small Overhead, Big Benefits for C++

We do not mean to make too big a deal about performance loss, nor do we wish to deny it. For simple C++ code written in a "Fortran" style, with a single layer of well-balanced parallel loops, the dynamic nature of TBB may not be needed at all. However, the limitations of such a coding style are an important factor in why TBB exists. TBB was designed to efficiently support nested, concurrent, and sequential composition of parallelism and to dynamically map this parallelism on to a target platform. Using a *composable* library like TBB, developers can build applications by combining components and libraries that contain parallelism without worrying that they will negatively interfere with each other. Importantly, TBB does not require us to restrict the parallelism we express to avoid performance problems. For large, complicated applications using C++, TBB is therefore easy to recommend without disclaimers.

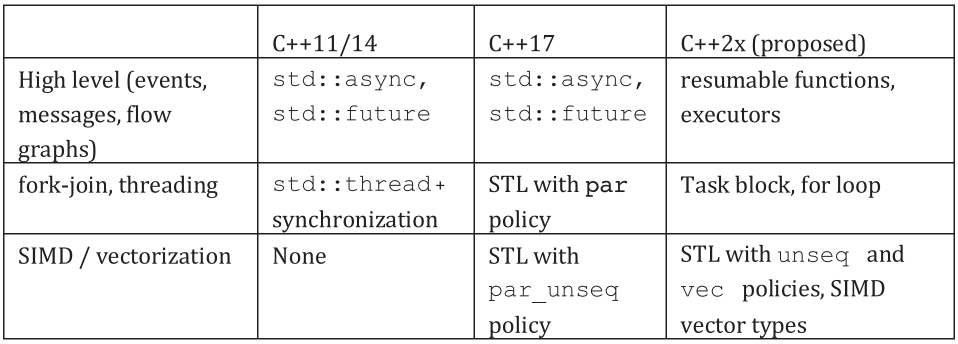
The TBB library has evolved over the years to not only adjust to new platforms but also to demands from developers that want a bit more control over the choices the library makes in mapping parallelism to the hardware. While TBB 1.0 had very few performance controls for users, TBB 2019 has quite a few more – such as affinity controls, constructs for work isolation, hooks that can be used to pin threads to cores, and so on. The developers of TBB worked hard to design these controls to provide just the right level of control without sacrificing composability.

The interfaces provided by the library are nicely layered – TBB provides high-level templates that suit the needs of most programmers, focusing on common cases. But it also provides low-level interfaces so we can drill down and create tailored solutions for our specific applications if needed. TBB has the best of both worlds. We typically rely on the default choices of the library to get great performance but can delve into the details if we need to.

Evolving Support for Parallelism in TBB and C++

Both the TBB library and the C++ language have evolved significantly since the introduction of the original TBB. In 2006, C++ had no language support for parallel programming, and many libraries, including the Standard Template Library (STL), were not easily used in parallel programs because they were not thread-safe.

The C++ language committee has been busy adding features for threading directly to the language and its accompanying Standard Template Library (STL). Figure 1-1 shows new and planned C++ features that address parallelism.



	C++11/14	C++17	C++2x (proposed)
High level (events, messages, flow graphs)	`std::async,` `std::future`	`std::async,` `std::future`	resumable functions, executors
fork-join, threading	`std::thread+` synchronization	STL with `par` policy	Task block, for loop
SIMD / vectorization	None	STL with `par_unseq` policy	STL with `unseq` and `vec` policies, SIMD vector types

Figure 1-1. *The features in the C++ standard as well as some proposed features*

Even though we are big fans of TBB, we would in fact prefer if all of the fundamental support needed for parallelism is in the C++ language itself. That would allow TBB to utilize a consistent foundation on which to build higher-level parallelism abstractions. The original versions of TBB had to address a lack of C++ language support, and this is an area where the C++ standard has grown significantly to fill the foundational voids that TBB originally had no choice but to fill with features such as portable locks and atomics. Unfortunately, for C++ developers, the standard still lacks features needed for full support of parallel programming. Fortunately, for readers of this book, this means that TBB is still relevant and essential for effective threading in C++ and will likely stay relevant for many years to come.

It is very important to understand that we are not complaining about the C++ standard process. Adding features to a language standard is best done very carefully, with careful review. The C++11 standard committee, for instance, spent huge energy on a memory model. The significance of this for parallel programming is critical for every library that builds upon the standard. There are also limits to what a language standard should include, and what it should support. We believe that the tasking system and the flow graph system in TBB is not something that will directly become part of a language standard. Even if we are wrong, it is not something that will happen anytime soon.

Recent C++ Additions for Parallelism

As shown in Figure 1-1, the C++11 standard introduced some low-level, basic building blocks for threading, including `std::async`, `std::future`, and `std::thread`. It also introduced atomic variables, mutual exclusion objects, and condition variables. These extensions require programmers to do a lot of coding to build up higher-level abstractions – but they do allow us to express basic parallelism directly in C++. The C++11 standard was a clear improvement when it comes to threading, but it doesn't provide us with the high-level features that make it easy to write portable, efficient parallel code. It also does not provide us with tasks or an underlying work-stealing task scheduler.

The C++17 standard introduced features that raise the level of abstraction above these low-level building blocks, making it easier for us to express parallelism without having to worry about every low-level detail. As we discuss later in this book, there are still some significant limitations, and so these features are not yet sufficiently expressive or performant – there's still a lot of work to do in the C++ standard.

The most pertinent of these C++17 additions are the *execution policies* that can be used with the Standard Template Library (STL) algorithms. These policies let us choose whether an algorithm can be safely parallelized, vectorized, parallelized and vectorized, or if it needs to retain its original sequenced semantics. We call an STL implementation that supports these policies a Parallel STL.

Looking into the future, there are proposals that might be included in a future C++ standard with even more parallelism features, such as resumable functions, executors, task blocks, parallel for loops, SIMD vector types, and additional execution policies for the STL algorithms.

The Threading Building Blocks (TBB) Library

The Threading Building Blocks (TBB) library is a C++ library that serves two key roles: (1) it fills foundational voids in support for parallelism where the C++ standard has not sufficiently evolved, or where new features are not fully supported by all compilers, and (2) it provides higher-level abstractions for parallelism that are beyond the scope of what the C++ language standard will likely ever include. TBB contains a number of features, as shown in Figure 1-2.

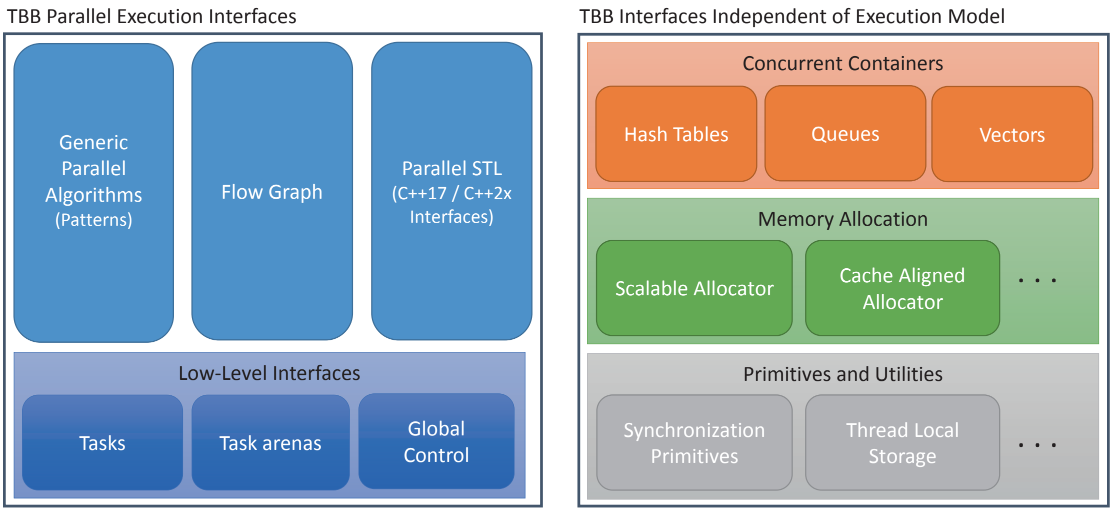


Figure 1-2. *The features of the TBB library*

These features can be categorized into two large groups: interfaces for expressing parallel computations and interfaces that are independent of the execution model.

Parallel Execution Interfaces

When we use TBB to create parallel programs, we express the parallelism in the application using one of the high-level interfaces or directly with tasks. We discuss tasks in more detail later in this book, but for now we can think of a TBB task as a lightweight object that defines a small computation and its associated data. As TBB developers, we express our application using tasks, either directly or indirectly through the prepackaged TBB algorithms, and the library schedules these tasks on to the platform’s hardware resources for us.

It’s important to note that as developers, we may want to express different kinds of parallelism. The three most common layers of parallelism that are expressed in parallel applications are shown in Figure 1-3. We should note that some applications may contain all three layers and others may contain only one or two of them. One of the most powerful aspects of TBB is that it provides high-level interfaces for each of these different parallelism layers, allowing us to exploit all of the layers using the same library.

The message-driven layer shown in Figure 1-3 captures parallelism that is structured as relatively large computations that communicate to each other through explicit messages. Common patterns in this layer include streaming graphs, data flow graphs, and dependency graphs. In TBB, these patterns are supported through the Flow Graph interfaces (described in Chapter 3).

The fork-join layer shown in Figure 1-3 supports patterns in which a serial computation branches out into a set of parallel tasks and then continues only when the parallel subcomputations are complete. Examples of fork-join patterns include functional parallelism (task parallelism), parallel loops, parallel reductions, and pipelines. TBB supports these with its Generic Parallel Algorithms (described in Chapter 2).

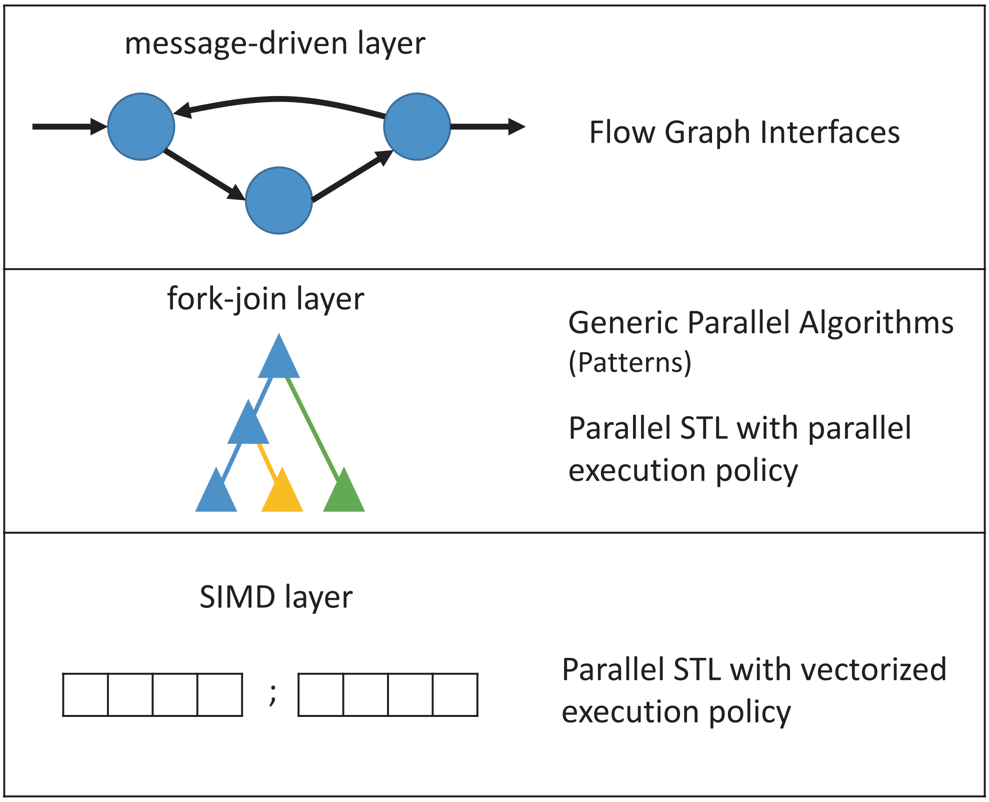

message-driven layer
Flow Graph Interfaces
fork-join layer
Generic Parallel Algorithms (Patterns)
Parallel STL with parallel execution policy
SIMD layer
;
Parallel STL with vectorized execution policy

Figure 1-3. *The three layers of parallelism commonly found in applications and how they map to the high-level TBB parallel execution interfaces*

Finally, the Single Instruction, Multiple Data (SIMD) layer is where data parallelism is exploited by applying the same operation to multiple data elements simultaneously. This type of parallelism is often implemented using vector extensions such as AVX, AVX2, and AVX-512 that use the vector units available in each processor core. There is a Parallel STL implementation (described in Chapter 4) included with all of the TBB distributions that provides vector implementations, among others, that take advantage of these extensions.

TBB provides high-level interfaces for many common parallel patterns, but there may still be cases where none of the high-level interfaces matches a problem. If that's the case, we can use TBB tasks directly to build our own algorithms.

The true power of the TBB parallel execution interfaces comes from the ability to mix them together, something usually called "composability." We can create applications that have a Flow Graph at the top level with nodes that use nested Generic Parallel Algorithms. These nested Generic Parallel Algorithms can, in turn, use Parallel STL algorithms in their bodies. Since the parallelism expressed by all of these layers is exposed to the TBB library, this one library can schedule the corresponding tasks in an efficient and composable way, making best use of the platform's resources.

One of the key properties of TBB that makes it composable is that it supports *relaxed sequential semantics*. Relaxed sequential semantics means that the parallelism we express using TBB tasks is in fact only a hint to the library; there is no guarantee that any of the tasks actually execute in parallel with each other. This gives tremendous flexibility to the TBB library to schedule tasks as necessary to improve performance. This flexibility lets the library provide scalable performance on systems, whether they have one core, eight cores, or 80 cores. It also allows the library to adapt to the dynamic load on the platform; for example, if one core is oversubscribed with work, TBB can schedule more work on the other cores or even choose to execute a parallel algorithm using only a single core. We describe in more detail why TBB is considered a composable library in Chapter 9.

Interfaces That Are Independent of the Execution Model

Unlike the parallel execution interfaces, the second large group of features in Figure 1-2 are completely independent of the execution model and of TBB tasks. These features are as useful in applications that use native threads, such as `pthreads` or `WinThreads`, as they are in applications that use TBB tasks.

These features include concurrent containers that provide thread-friendly interfaces to common data structures like hash tables, queues, and vectors. They also include features for memory allocation like the TBB scalable memory allocator and the cache aligned allocator (both described in Chapter 7). They also include lower-level features such as synchronization primitives and thread-local storage.

Using the Building Blocks in TBB

As developers, we can pick and choose the parts of TBB that are useful for our applications. We can, for example, use just the scalable memory allocator (described in Chapter 7) and nothing else. Or, we can use concurrent containers (described in Chapter 6) and a few Generic Parallel Algorithms (Chapter 2). And of course, we can also choose to go all in and build an application that combines all three high-level execution interfaces and makes use of the TBB scalable memory allocator and concurrent containers, as well as the many other features in the library.

Let's Get Started Already!

Getting the Threading Building Blocks (TBB) Library

Before we can start using TBB, we need to get a copy of the library. There are a few ways to do this. At the time of the writing of this book, these ways include

- Follow links at `www.threadingbuildingblocks.org` or `https://software.intel.com/intel-tbb` to get a free version of the TBB library directly from Intel. There are precompiled versions available for Windows, Linux, and macOS. The latest packages include both the TBB library and an implementation of the Parallel STL algorithms that uses TBB for threading.
- Visit `https://github.com/intel/tbb` to get the free, open-source version of the TBB library. The open-source version of TBB is in no way a lite version of the library; it contains all of the features of the commercially supported version. You can choose to checkout and build from source, or you can click "releases" to download a version that has been built and tested by Intel. At GitHub, pre-built and tested versions are available for Windows, Linux, macOS, and Android. Again, the latest packages for the pre-built versions of TBB include both the TBB library and an implementation of Parallel STL that uses TBB for threading. If you want the source code for Parallel STL, you will need to download that separately from `https://github.com/intel/parallelstl`.
- You can download a copy of the Intel Parallel Studio XE tool suite `https://software.intel.com/intel-parallel-studio-xe`. TBB and a Parallel STL that uses TBB is currently included in all editions of this tool suite, including the smallest Composer Edition. If you have a recent version of the Intel C++ compiler installed, you likely already have TBB installed on your system.

We leave it to readers to select the most appropriate route for getting TBB and to follow the directions for installing the package that are provided at the corresponding site.

Getting a Copy of the Examples

All of the code examples used in this book are available at `https://github.com/Apress/pro-TBB`. In this repository, there are directories for each chapter. Many of the source files are named after the figure they appear in, for example `ch01/fig_1_04.cpp` contains code that matches Figure 1-4 in this chapter.

Writing a First “Hello, TBB!” Example

Figure 1-4 provides a small example that uses a `tbb::parallel_invoke` to evaluate two functions, one that prints `Hello` and the other that prints `TBB!` in parallel. This example is trivial and will not benefit from parallelization, but we can use it to be sure that we have set up our environment properly to use TBB. In Figure 1-4, we include the tbb.h header to get access to the TBB functions and classes, all of which are in namespace tbb. The call to `parallel_invoke` asserts to the TBB library that the two functions passed to it are independent of each other and are safe to execute in parallel on different cores or threads and in any order. Under these constraints, the resulting output may contain either `Hello` or `TBB!` first. We might even see that there is no newline character between the two strings and two consecutive newlines at the end of the output, since the printing of each string and its `std::endl` do not happen atomically.

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

int main() {
  tbb::parallel_invoke(
    [](）{ std::cout << " Hello " << std::endl; },
    []() { std::cout << " TBB! " << std::endl; }
  );
  return 0;
}
```

Figure 1-4. *A Hello TBB example*

Figure 1-5 provides an example that uses a Parallel STL `std::for_each` to apply a function in parallel to two items in a `std::vector`. Passing a `pstl::execution::par` policy to the `std::for_each` asserts that it is safe to apply the provided function in parallel on different cores or threads to the result of dereferencing every iterator in the range `[v.begin(), v.end())`. Just like with Figure 1-4, the output that results from running this example might have either string printed first.

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

int main() {
  std::vector<std::string> v = {" Hello ",
                                " Parallel STL! "};
  std::for_each(pstl::execution::par, v.begin(), v.end(),
    [](std::string &s) { std::cout << s << std::endl; }
  );
  return 0;
}
```

Figure 1-5. *A Hello Parallel STL example*

In both Figures 1-4 and 1-5, we use C++ *lambda expressions* to specify the functions. Lambda expressions are very useful when using libraries like TBB to specify the user code to execute as a task. To help review C++ lambda expressions, we offer a callout box “A Primer on C++ Lambda Expressions” with an overview of this important modern C++ feature.

A PRIMER ON C++ LAMBDA EXPRESSIONS

Support for lambda expressions was introduced in C++11. They are used to create anonymous function objects (although you can assign them to named variables) that can capture variables from the enclosing scope. The basic syntax for a C++ lambda expression is

[*capture-list* **] (** *params* **)** -> *ret* **{** *body* **}**

where

- *capture-list* is a comma-separated list of captures. We capture a variable by value by listing the variable name in the capture-list. We capture a variable by reference by prefixing it with an ampersand, for example, &v. And we can use `this` to capture the current object by reference. There are also defaults: [=] is used to capture all automatic variables used in the body by value and the current object by reference, [&] is used to capture all automatic variables used in the body as well as the current object by reference, and [] captures nothing.

- *params* is the list of function parameters, just like for a named function.
- *ret* is the return type. If *->ret* is not specified, it is inferred from the return statements.
- *body* is the function body.

This next example shows a C++ lambda expression that captures one variable, i, by value and another, j, by reference. It also has a parameter k0 and another parameter l0 that is received by reference:

```
int i = 1, j = 10, k = 100, l = 1000;
auto lambdaExpression = [i, &j] (int k0, int &l0) -> int {
  j = 2 * j;
  k0 = 2 * k0;
  l0 = 2 * l0;
  return i + j + k0 + l0;
};

printValues(i, j, k, l);
std::cout << "First call returned " << lambdaExpression(k, l)
          << std::endl;
printValues(i, j, k, l);
std::cout << "Second call returned " << lambdaExpression(k, l)
          << std::endl;
printValues(i, j, k, l);
return 0;
```

Running the example will result in the following output:

```
i == 1
j == 10
k == 100
l == 1000
First call returned 2221
i == 1
j == 20
k == 100
l == 2000
Second call returned 4241
i == 1
j == 40
k == 100
l == 4000
```

We can think of a lambda expression as an instance of a function object, but the compiler creates the class definition for us. For example, the lambda expression we used in the preceding example is analogous to an instance of a class:

```
class Functor {
  int my_i;
  int &my_jRef;

public:
  Functor(int i, int &j) : my_i{i}, my_jRef{j} { }

  int operator()(int k0, int &l0) {
    my_jRef = 2 * my_jRef;
    k0 = 2 * k0;
    l0 = 2 * l0;
    return my_i + my_jRef + k0 + l0;
  }
};
```

Wherever we use a C++ lambda expression, we can substitute it with an instance of a function object like the preceding one. In fact, the TBB library predates the C++11 standard and all of its interfaces used to require passing in instances of objects of user-defined classes. C++ lambda expressions simplify the use of TBB by eliminating the extra step of defining a class for each use of a TBB algorithm.

Building the Simple Examples

Once we have written the examples in Figures 1-4 and 1-5, we need to build executable files from them. The instructions for building an application that uses TBB are OS and compiler dependent. However, in general, there are two necessary steps to properly configure an environment.

Steps to Set Up an Environment

1. We must inform the compiler about the location of the TBB headers and libraries. If we use Parallel STL interfaces, we must also inform the compiler about the location of the Parallel STL headers.

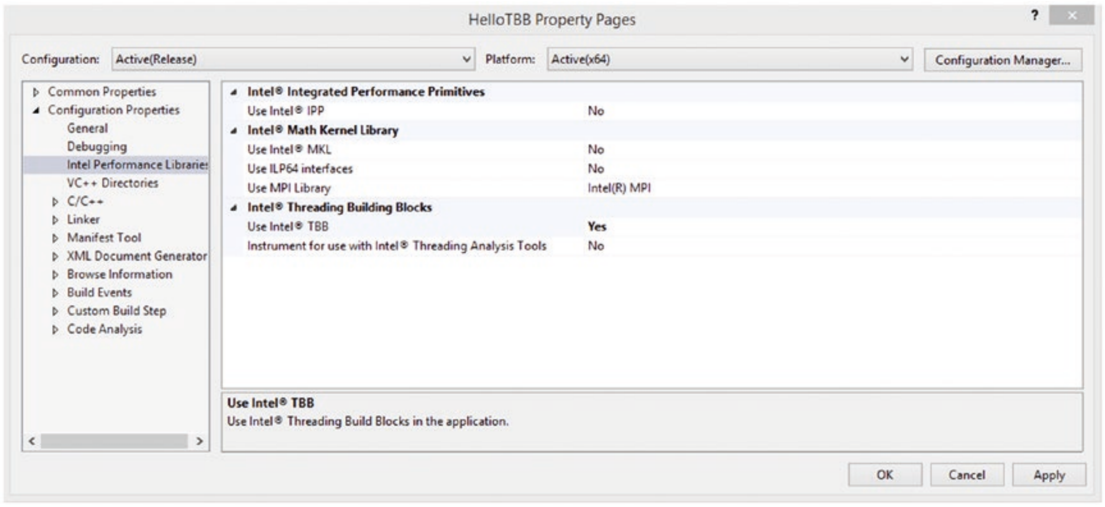
2. We must configure our environment so that the application can locate the TBB libraries when it is run. TBB is shipped as a dynamically linked library, which means that it is not directly embedded into our application; instead, the application locates and loads it at runtime. The Parallel STL interfaces do not require their own dynamically linked library but do depend on the TBB library.

We will now briefly discuss some of the most common ways to accomplish these steps on Windows and Linux. The instructions for macOS are similar to those for Linux. There are additional cases and more detailed directions in the documentation that ships with the TBB library.

Building on Windows Using Microsoft Visual Studio

If we download either the commercially supported version of TBB or a version of Intel Parallel Studio XE, we can integrate the TBB library with Microsoft Visual Studio when we install it, and then it is very simple to use TBB from Visual Studio.

To create a "Hello, TBB!" project, we create a project as usual in Visual Studio, add a "`.cpp`" file with the code contained in Figure 1-4 or Figure 1-5, and then go to the project's **Property Pages,** traverse to **Configuration Properties ➤ Intel Performance Libraries** and change **Use TBB** to **Yes**, as shown in Figure 1-6. This accomplishes step 1. Visual Studio will now link the TBB library into the project as it has the proper paths to the header files and libraries. This also properly sets the paths to the Parallel STL headers.



Configuration: Active(Release)   Platform: Active(x64)

Intel® Integrated Performance Primitives	
Use Intel® IPP	No
Intel® Math Kernel Library	
Use Intel® MKL	No
Use ILP64 interfaces	No
Use MPI Library	Intel(R) MPI
Intel® Threading Building Blocks	
Use Intel® TBB	**Yes**
Instrument for use with Intel® Threading Analysis Tools	No

Use Intel® TBB
Use Intel® Threading Build Blocks in the application.

Figure 1-6. *Setting Use **TBB** to **Yes** in the project Property Pages in Visual Studio*

On Windows systems, the TBB libraries that are dynamically loaded by the application executable at runtime are the "`.dll`" files. To complete step 2 in setting up our environment, we need to add the location of these files to our PATH environment variable. We can do this by adding the path to either our Users or System PATH variable. One place to find these settings is in the Windows Control Panel by traversing **System and Security ➤ System ➤ Advanced System Settings ➤ Environment Variables**. We can refer to the documentation for our installation of TBB for the exact locations of the "`.dll`" files.

Note Changes to the PATH variable in an environment only take effect in Microsoft Visual Studio after it is restarted.

Once we have the source code entered, have **Use TBB** set to **Yes**, and have the path to the TBB "`.dll`"s in our PATH variable, we can build and execute the program by entering **Ctrl-F5.**

Building on a Linux Platform from a Terminal

Using the Intel Compiler

When using the Intel C++ Compiler, the compilation process is simplified because the TBB library is included with the compiler and it supports a compiler flag `-tbb` that properly sets the include and library paths during compilation for us. Therefore, to compile our examples using the Intel C++ Compiler, we just add the `-tbb` flag to the compile line.

```
icpc -std=c++11 -tbb -o fig_1_04 fig_1_04.cpp
icpc -std=c++11 -tbb -o fig_1_05 fig_1_05.cpp
```

`tbbvars` and `pstlvars` Scripts

If we are not using the Intel C++ Compiler, we can use scripts that are included with the TBB and Parallel STL distributions to set up our environment. These scripts modify the `CPATH`, `LIBRARY_PATH` and `LD_LIBRARY_PATH` environment variables to include the directories needed to build and run TBB and Parallel STL applications. The `CPATH` variable adds additional directories to the list of directories the compiler searches when it looks for `#include` files. The `LIBRARY_PATH` adds additional directories to the list of directories the compiler searches when it looks for libraries to link against at compile time. And the `LD_LIBRARY_PATH` adds additional directories to the list of directories the executable will search when it loads dynamic libraries at runtime.

Let us assume that the root directory of our TBB installation is `TBB_ROOT`. The TBB library comes with a set of scripts in the `${TBB_ROOT}/bin` directory that we can execute to properly set up the environment. We need to pass our architecture type `[ia32|intel64|mic]` to this script. We also need to add a flag at compile time to enable the use of C++11 features, such as our use of lambda expressions.

Even though the Parallel STL headers are included with all of the recent TBB library packages, we need to take an extra step to add them to our environment. Just like with TBB, Parallel STL comes with a set of scripts in the `${PSTL_ROOT}/bin` directory. The `PSTL_ROOT` directory is typically a sibling of the `TBB_ROOT` directory. We also need to pass in our architecture type and enable the use of C++11 features to use Parallel STL.

The steps to build and execute the example in Figure 1-4 on a Linux platform with 64-bit Intel processors look like

```
source ${TBB_ROOT}/bin/tbbvars.sh intel64 linux auto_tbbroot
g++ -std=c++11 -o fig_1_04 fig_1_04.cpp -ltbb
./fig_1_04
```

The steps to build and execute the example in Figure 1-5 on a Linux platform with 64-bit Intel processors look like

```
source ${TBB_ROOT}/bin/tbbvars.sh intel64 linux auto_tbbroot
source ${PSTL_ROOT}/bin/pstlvars.sh intel64 auto_pstlroot
g++ -std=c++11 -o fig_1_05 fig_1_05.cpp -ltbb
./fig_1_05
```

Note Increasingly, Linux distributions include a copy of the TBB library. On these platforms, the GCC compiler may link against the platform's version of the TBB library instead of the version of the TBB library that is added to the LIBRARY_PATH by the `tbbvars` script. If we see linking problems when using TBB, this might be the issue. If this is the case, we can add an explicit library path to the compiler's command line to choose a specific version of the TBB library.

For example:

```
g++ -L${TBB_ROOT}/lib/intel64/gcc4.7 -ltbb ...
```

We can add `-Wl,--verbose` to the `g++` command line to generate a report of all of the libraries that are being linked against during compilation to help diagnose this issue.

Although we show commands for `g++`, except for the compiler name used, the command lines are the same for the Intel compiler (`icpc`) or LLVM (`clang++`).

Setting Up Variables Manually Without Using the `tbbvars` Script or the Intel Compiler

Sometimes we may not want to use the `tbbvars` scripts, either because we want to know exactly what variables are being set or because we need to integrate with a build system. If that's not the case for you, skip over this section unless you really feel the urge to do things manually.

Since you're still reading this section, let's look out how we can build and execute on the command line without using the `tbbvars` script. When compiling with a non-Intel compiler, we don't have the `-tbb` flag available to us, so we need to specify the paths to both the TBB headers and the shared libraries.

If the root directory of our TBB installation is `TBB_ROOT`, the headers are in `${TBB_ROOT}/include` and the shared library files are stored in `${TBB_ROOT}/lib/${ARCH}/${GCC_LIB_VERSION}`, where `ARCH` is the system architecture `[ia32|intel64|mic]` and the `GCC_LIB_VERSION` is the version of the TBB library that is compatible with your GCC or clang installation.

The underlying difference between the TBB library versions is their dependencies on features in the C++ runtime libraries (such as `libstdc++` or `libc++`).

Typically to find an appropriate TBB version to use, we can execute the command `gcc -version` in our terminal. We then select the closest GCC version available in `${TBB_ROOT}/lib/${ARCH}` that is not newer than our GCC version (this usually works even when we are using clang++). But because installations can vary from machine to machine, and we can choose different combinations of compilers and C++ runtimes, this simple approach may not always work. If it does not, refer to the TBB documentation for additional guidance.

For example, on a system with GCC 5.4.0 installed, we compiled the example in Figure 1-4 with

```
g++ -std=c++11 -o fig_1_04 fig_1_04.cpp    \
   -I ${TBB_ROOT}/include                  \
   -L ${TBB_ROOT}/lib/intel64/gcc4.7 -ltbb
```

And when using clang++, we used the same TBB version:

```
clang++ -std=c++11 -o fig_1_04 fig_1_04.cpp  \
    -I ${TBB_ROOT}/include                   \
    -L ${TBB_ROOT}/lib/intel64/gcc-4.7 -ltbb
```

To compile the example in Figure 1-5, we also need to add the path to the Parallel STL include directory:

```
g++ -std=c++11 -o fig_1_05 fig_1_05.cpp       \
    -I ${TBB_ROOT}/include                    \
    -I ${PSTL_ROOT}/include                   \
    -L ${TBB_ROOT}/lib/intel64/gcc4.7 -ltbb
```

Regardless of whether we have compiled with the Intel compiler, gcc, or clang++, we need to add the TBB shared library location to our `LD_LIBRARY_PATH` so that it can be found when the application runs. Again, assuming that the root directory of our TBB installation is `TBB_ROOT`, we can set this, for example, with

```
export LD_LIBRARY_PATH=${TBB_ROOT}/lib/${ARCH}/${GCC_LIB_VERSION}:${LD_
LIBRARY_PATH}
```

Once we have compiled our application using the Intel compiler, gcc, or clang++ and have set our `LD_LIBRARY_PATH` as required, we can then run the applications from the command line:

```
./fig_1_04
```

This should result in an output similar to

```
 Hello
 Parallel STL!
```

A More Complete Example

The previous sections provide the steps to write, build, and execute a simple TBB application and a simple Parallel STL application that each print a couple of lines of text. In this section, we write a bigger example that can benefit from parallel execution using all three of the high-level execution interfaces shown in Figure 1-2. We do not explain all of the details of the algorithms and features used to create this example, but instead we use this example to see the different layers of parallelism that can be expressed with TBB. This example is admittedly contrived. It is simple enough to explain in a few paragraphs but complicated enough to exhibit all of the parallelism layers described in Figure 1-3. The final multilevel parallel version we create here should be viewed as a syntactic demonstration, not a how-to guide on how to write an optimal TBB application. In subsequent chapters, we cover all of the features used in this section in more detail and provide guidance on how to use them to get great performance in more realistic applications.

Starting with a Serial Implementation

Let’s start with the serial implementation shown in Figure 1-7. This example applies a gamma correction and a tint to each image in a vector of images, writing each result to a file. The highlighted function, `fig_1_7`, contains a for-loop that processes the elements of a vector by executing `applyGamma`, `applyTint`, and `writeImage` functions on each image. The serial implementations of each of these functions are also provided in Figure 1-7. The definitions of the image representation and some of the helper functions are contained in `ch01.h`. This header file is available, along with all of the source code for the example, at `https://github.com/Apress/threading-building-blocks`.

```
#include <iostream>
#include <vector>
#include <tbb/tbb.h>
#include "ch01.h"

using ImagePtr = std::shared_ptr<ch01::Image>;

ImagePtr applyGamma(ImagePtr image_ptr, double gamma);
ImagePtr applyTint(ImagePtr image_ptr, const double *tints);
void writeImage(ImagePtr image_ptr);

void fig_1_7(const std::vector<ImagePtr> &image_vector) {
  const double tint_array[] = {0.75, 0, 0};
  for (ImagePtr img : image_vector) {
    img = applyGamma(img, 1.4);
    img = applyTint(img, tint_array);
    writeImage(img);
  }
}

ImagePtr applyGamma(ImagePtr image_ptr, double gamma) {
  auto output_image_ptr =
    std::make_shared<ch01::Image>(image_ptr->name() + "_gamma",
      ch01::IMAGE_WIDTH, ch01::IMAGE_HEIGHT);
  auto in_rows = image_ptr->rows();
  auto out_rows = output_image_ptr->rows();
  const int height = in_rows.size();
  const int width = in_rows[1] - in_rows[0];

  for ( int i = 0; i < height; ++i ) {
    for ( int j = 0; j < width; ++j ) {
      const ch01::Image::Pixel &p = in_rows[i][j];
      double v = 0.3*p.bgra[2] + 0.59*p.bgra[1] + 0.11*p.bgra[0];
      double res = pow(v, gamma);
      if(res > ch01::MAX_BGR_VALUE) res = ch01::MAX_BGR_VALUE;
      out_rows[i][j] = ch01::Image::Pixel(res, res, res);
    }
  }
  return output_image_ptr;
}

ImagePtr applyTint(ImagePtr image_ptr, const double *tints) {
  auto output_image_ptr =
    std::make_shared<ch01::Image>(image_ptr->name() + "_tinted",
      ch01::IMAGE_WIDTH, ch01::IMAGE_HEIGHT);
  auto in_rows = image_ptr->rows();
  auto out_rows = output_image_ptr->rows();
  int height = in_rows.size();
  const int width = in_rows[1] - in_rows[0];

  for ( int i = 0; i < height; ++i ) {
    for ( int j = 0; j < width; ++j ) {
      const ch01::Image::Pixel &p = in_rows[i][j];
      std::uint8_t b = (double)p.bgra[0] +
```

Figure 1-7. *A serial implementation of an example that applies a gamma correction and a tint to a vector of images*

```
                          (ch01::MAX_BGR_VALUE-p.bgra[0])*tints[0];
        std::uint8_t g = (double)p.bgra[1] +
                          (ch01::MAX_BGR_VALUE-p.bgra[1])*tints[1];
        std::uint8_t r = (double)p.bgra[2] +
                          (ch01::MAX_BGR_VALUE-p.bgra[2])*tints[2];
        out_rows[i][j] =
          ch01::Image::Pixel(
            (b > ch01::MAX_BGR_VALUE) ? ch01::MAX_BGR_VALUE : b,
            (g > ch01::MAX_BGR_VALUE) ? ch01::MAX_BGR_VALUE : g,
            (r > ch01::MAX_BGR_VALUE) ? ch01::MAX_BGR_VALUE : r
        );
      }
    }
    return output_image_ptr;
  }

  void writeImage(ImagePtr image_ptr) {
    image_ptr->write( (image_ptr->name() + ".bmp").c_str());
  }

  int main(int argc, char* argv[]) {
    std::vector<ImagePtr> image_vector;

    for ( int i = 2000; i < 20000000; i *= 10 )
      image_vector.push_back(ch01::makeFractalImage(i));

    tbb::tick_count t0 = tbb::tick_count::now();
    fig_1_7(image_vector);
    std::cout << "Time : " << (tbb::tick_count::now()-t0).seconds()
              << " seconds" << std::endl;
    return 0;
  }
```

Figure 1-7. (*continued*)

Both the `applyGamma` function and the `applyTint` function traverse across the rows of the image in an outer for-loop and the elements of each row in an inner for-loop. New pixel values are computed and assigned to the output image. The `applyGamma` function applies a gamma correction. The `applyTint` function applies a blue tint to the image. The functions receive and return `std::shared_ptr` objects to simplify memory management; readers that are unfamiliar with `std::shared_ptr` can refer to the sidebar discussion “A note on smart pointers.” Figure 1-8 shows example outputs for an image fed through the example code.

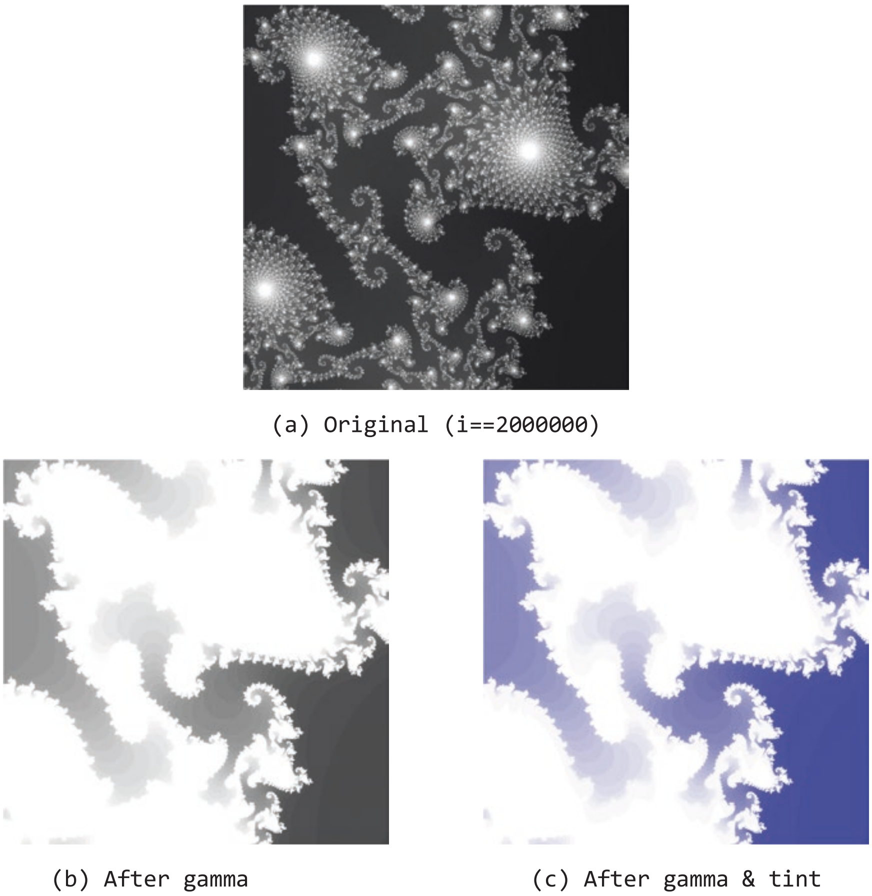


Figure 1-8. *Outputs for the example: (a) the original image generated by `ch01::makeFractalImage(2000000)`, (b) the image after it has been gamma corrected, and (c) the image after it has been gamma corrected and tinted*

A NOTE ON SMART POINTERS

One of the most challenging parts of programming in C/C++ can be dynamic memory management. When we use new/delete or malloc/free, we have to be sure we that we match them up correctly to avoid memory leaks and double frees. Smart pointers including `unique_ptr`, `shared_ptr`, and `weak_ptr` were introduced in C++11 to provide automatic, exception-safe memory management. For example, if we allocate an object by using `make_shared`, we receive a smart pointer to the object. As we assign this shared pointer to other shared pointers, the C++ library takes care of reference counting for us. When there are no outstanding references to our object through any smart pointers, then the object is automatically freed. In most of the examples in this book, including in Figure 1-7, we use smart pointers instead of raw pointers. Using smart pointers, we don't have to worry about finding all of the points where we need to insert a free or delete – we can just rely on the smart pointers to do the right thing.

Adding a Message-Driven Layer Using a Flow Graph

Using a top-down approach, we can replace the outer loop in function `fig_1_07` in Figure 1-7 with a TBB Flow Graph that streams images through a set of filters as shown in Figure 1-9. We admit that this is the most contrived of our choices in this particular example. We could have easily used an outer parallel loop in this case; or we could have merged the Gamma and Tint loop nests together. But for demonstration purposes, we choose to express this as a graph of separate nodes to show how TBB can be used to express message-driven parallelism, the top level of the parallelism in Figure 1-3. In Chapter 3, we will learn more about the TBB Flow Graph interfaces and discover more natural applications for this high-level, message-driven execution interface.

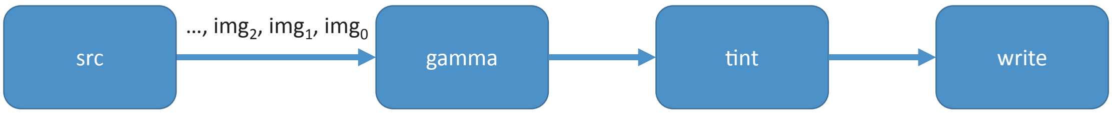


Figure 1-9. *A data flow graph that has four nodes: (1) a node that gets or generates images, (2) a node that applies the gamma correction, (3) a node that applies the tint, and (4) a node that writes out the resulting image*

By using the data flow graph in Figure 1-9, we can overlap the execution of different stages of the pipeline as they are applied to different images. For example, when a first image, img_0, completes in the `gamma` node, the result is passed to the `tint` node, while a new image img_1 enters the `gamma` node. Likewise, when this next step is done, img_0, which has now passed through both the `gamma` and `tint` nodes, is sent to the `write` node. Meanwhile, img_1 is sent to the `tint` node, and a new image, img_2, begins processing in the `gamma` node. At each step, the execution of the filters is independent of each other, and so these computations can be spread across different cores or threads. Figure 1-10 shows the loop from function `fig_1_7` now expressed as a TBB Flow Graph.

```
void fig_1_10(const std::vector<ImagePtr> &image_vector) {
  const double tint_array[] = {0.75, 0, 0};

  tbb::flow::graph g;

  int i = 0;
  tbb::flow::source_node<ImagePtr> src(g,
    [&i, &image_vector] (ImagePtr &out) -> bool {
      if ( i < image_vector.size() ) {
        out = image_vector[i++];
        return true;
      } else {
        return false;
      }
    }, false);

  tbb::flow::function_node<ImagePtr, ImagePtr> gamma(g,
    tbb::flow::unlimited,
    [] (ImagePtr img) -> ImagePtr {
      return applyGamma(img, 1.4);
    }
  );

  tbb::flow::function_node<ImagePtr, ImagePtr> tint(g,
    tbb::flow::unlimited,
    [tint_array] (ImagePtr img) -> ImagePtr {
      return applyTint(img, tint_array);
    }
  );

  tbb::flow::function_node<ImagePtr> write(g,
    tbb::flow::unlimited,
    [] (ImagePtr img) {
      writeImage(img);
    }
  );

  tbb::flow::make_edge(src, gamma);
  tbb::flow::make_edge(gamma, tint);
  tbb::flow::make_edge(tint, write);
  src.activate();
  g.wait_for_all();
}
```

Figure 1-10. *Using a TBB Flow Graph in place of the outer for-loop*

As we will see in Chapter 3, several steps are needed to build and execute a TBB Flow Graph. First, a graph object, g, is constructed. Next, we construct the nodes that represent the computations in our data flow graph. The node that streams the images to the rest of the graph is a source_node named src. The computations are performed by the function_node objects named gamma, tint, and write. We can think of a source_node as a node that has no input and continues to send data until it runs out of data to send. We can think of a function_node as a wrapper around a function that receives an input and generates an output.

After the nodes are created, we connect them to each other using edges. Edges represent the dependencies or communication channels between nodes. Since, in our example in Figure 1-10, we want the src node to send the initial images to the gamma node, we make an edge from the src node to the gamma node. We then make an edge from the gamma node to the tint node. And likewise, we make an edge from the tint node to the write node. Once we complete construction of the graph's structure, we call src.activate() to start the source_node and call g.wait_for_all() to wait until the graph completes.

When the application in Figure 1-10 executes, each image generated by the src node passes through the pipeline of nodes as described previously. When an image is sent to the gamma node, the TBB library creates and schedules a task to apply the gamma node's body to the image. When that processing is done, the output is fed to the tint node. Likewise, TBB will create and schedule a task to execute the tint node's body on that output of the gamma node. Finally, when that processing is done, the output of the tint node is sent to the write node. Again, a task is created and scheduled to execute the body of the node, in this case writing the image to a file. Each time an execution of the src node finishes and returns true, a new task is spawned to execute the src node's body again. Only after the src node stops generating new images and all of the images it has already generated have completed processing in the write node will the wait_for_all call return.

Adding a Fork-Join Layer Using a parallel_for

Now, let's turn our attention to the implementation of the applyGamma and applyTint functions. In Figure 1-11, we replace the outer i-loops in the serial implementations with calls to tbb::parallel_for. We use a parallel_for Generic Parallel Algorithm to execute across different rows in parallel. A parallel_for creates tasks that can be spread across multiple processor cores on a platform. This pattern is an example of the fork-join layer from Figure 1-3 and is described in more detail in Chapter 2.

```
ImagePtr applyGamma(ImagePtr image_ptr, double gamma) {
  auto output_image_ptr =
    std::make_shared<ch01::Image>(image_ptr->name() + "_gamma",
      ch01::IMAGE_WIDTH, ch01::IMAGE_HEIGHT);
  auto in_rows = image_ptr->rows();
  auto out_rows = output_image_ptr->rows();
  const int height = in_rows.size();
  const int width = in_rows[1] - in_rows[0];

  tbb::parallel_for( 0, height,
    [&in_rows, &out_rows, width, gamma](int i) {
      for ( int j = 0; j < width; ++j ) {
        const ch01::Image::Pixel &p = in_rows[i][j];
        double v = 0.3*p.bgra[2] + 0.59*p.bgra[1] + 0.11*p.bgra[0];
        double res = pow(v, gamma);
        if(res > ch01::MAX_BGR_VALUE) res = ch01::MAX_BGR_VALUE;
        out_rows[i][j] = ch01::Image::Pixel(res, res, res);
      }
    }
  );
  return output_image_ptr;
}

ImagePtr applyTint(ImagePtr image_ptr, const double *tints) {
  auto output_image_ptr =
    std::make_shared<ch01::Image>(image_ptr->name() + "_tinted",
      ch01::IMAGE_WIDTH, ch01::IMAGE_HEIGHT);
  auto in_rows = image_ptr->rows();
  auto out_rows = output_image_ptr->rows();
  const int height = in_rows.size();
  const int width = in_rows[1] - in_rows[0];

  tbb::parallel_for( 0, height,
    [&in_rows, &out_rows, width, tints](int i) {
      for ( int j = 0; j < width; ++j ) {
        const ch01::Image::Pixel &p = in_rows[i][j];
        std::uint8_t b = (double)p.bgra[0] +
                         (ch01::MAX_BGR_VALUE-p.bgra[0])*tints[0];
        std::uint8_t g = (double)p.bgra[1] +
                         (ch01::MAX_BGR_VALUE-p.bgra[1])*tints[1];
        std::uint8_t r = (double)p.bgra[2] +
                         (ch01::MAX_BGR_VALUE-p.bgra[2])*tints[2];
        out_rows[i][j] =
          ch01::Image::Pixel(
            (b > ch01::MAX_BGR_VALUE) ? ch01::MAX_BGR_VALUE : b,
            (g > ch01::MAX_BGR_VALUE) ? ch01::MAX_BGR_VALUE : g,
            (r > ch01::MAX_BGR_VALUE) ? ch01::MAX_BGR_VALUE : r
          );
      }
    }
  );
  return output_image_ptr;
}
```

Figure 1-11. *Adding parallel_for to apply the gamma correction and tint across rows in parallel*

Adding a SIMD Layer Using a Parallel STL Transform

We can further optimize our two computational kernels by replacing the inner `j`-loops with calls to the Parallel STL function `transform`. The `transform` algorithm applies a function to each element in an input range, storing the results into an output range. The arguments to `transform` are (1) the execution policy, (2 and 3) the input range of elements, (4) the beginning of the output range, and (5) the lambda expression that is applied to each element in the input range and whose result is stored to the output elements.

In Figure 1-12, we use the `unseq` execution policy to tell the compiler to use the SIMD version of the transform function. The Parallel STL functions are described in more detail in Chapter 4.

```
#include <iostream>
#include <vector>
#include <tbb/tbb.h>
#include <pstl/algorithm>
#include <pstl/execution>
#include "ch01.h"

using ImagePtr = std::shared_ptr<ch01::Image>;
void writeImage(ImagePtr image_ptr);

ImagePtr applyGamma(ImagePtr image_ptr, double gamma) {
  auto output_image_ptr =
    std::make_shared<ch01::Image>(image_ptr->name() + "_gamma",
      ch01::IMAGE_WIDTH, ch01::IMAGE_HEIGHT);
  auto in_rows = image_ptr->rows();
  auto out_rows = output_image_ptr->rows();
  const int height = in_rows.size();
  const int width = in_rows[1] - in_rows[0];

  tbb::parallel_for( 0, height,
    [&in_rows, &out_rows, width, gamma](int i) {
      auto in_row = in_rows[i];
      auto out_row = out_rows[i];
      std::transform(pstl::execution::unseq, in_row, in_row+width,
        out_row, [gamma](const ch01::Image::Pixel &p) {
          double v = 0.3*p.bgra[2] + 0.59*p.bgra[1] + 0.11*p.bgra[0];
          assert(v > 0);
          double res = pow(v, gamma);
          if(res > ch01::MAX_BGR_VALUE) res = ch01::MAX_BGR_VALUE;
          return ch01::Image::Pixel(res, res, res);
      });
    }
  );
  return output_image_ptr;
}
```

Figure 1-12. *Using `std::transform` to add SIMD parallelism to the inner loops*

```
ImagePtr applyTint(ImagePtr image_ptr, const double *tints) {
  auto output_image_ptr =
    std::make_shared<ch01::Image>(image_ptr->name() + "_tinted",
      ch01::IMAGE_WIDTH, ch01::IMAGE_HEIGHT);
  auto in_rows = image_ptr->rows();
  auto out_rows = output_image_ptr->rows();
  const int height = in_rows.size();
  const int width = in_rows[1] - in_rows[0];

  tbb::parallel_for( 0, height,
    [&in_rows, &out_rows, width, tints](int i) {
      auto in_row = in_rows[i];
      auto out_row = out_rows[i];
      std::transform(pstl::execution::unseq, in_row, in_row+width,
        out_row, [tints](const ch01::Image::Pixel &p) {
          std::uint8_t b = (double)p.bgra[0] +
                           (ch01::MAX_BGR_VALUE-p.bgra[0])*tints[0];
          std::uint8_t g = (double)p.bgra[1] +
                           (ch01::MAX_BGR_VALUE-p.bgra[1])*tints[1];
          std::uint8_t r = (double)p.bgra[2] +
                           (ch01::MAX_BGR_VALUE-p.bgra[2])*tints[2];
          return ch01::Image::Pixel(
            (b > ch01::MAX_BGR_VALUE) ? ch01::MAX_BGR_VALUE : b,
            (g > ch01::MAX_BGR_VALUE) ? ch01::MAX_BGR_VALUE : g,
            (r > ch01::MAX_BGR_VALUE) ? ch01::MAX_BGR_VALUE : r
          );
      });
    }
  );
  return output_image_ptr;
}
```

Figure 1-12. (*continued*)

In Figure 1-12, each `Image::Pixel` object contains an array with four single byte elements, representing the blue, green, red, and alpha values for that pixel. By using the `unseq` execution policy, a vectorized loop is used to apply the function across the row of elements. This level of parallelization corresponds to the SIMD layer in Figure 1-3 and takes advantage of the vector units in the CPU core that the code executes on but does not spread the computation across different cores.

Note Passing an execution policy to a Parallel STL algorithm does not guarantee parallel execution. It is legal for the library to choose a more restrictive execution policy than the one requested. It is therefore important to check the impact of using an execution policy – especially one that depends on compiler implementations!

While the examples we created in Figure 1-7 through Figure 1-12 are a bit contrived, they demonstrate the breadth and power of the TBB library's parallel execution interfaces. Using a single library, we expressed message-driven, fork-join, and SIMD parallelism, composing them together into a single application.

Summary

In this chapter, we started by explaining why a library such as TBB is even more relevant today than it was when it was first introduced over 10 years ago. We then briefly looked at the major features in the library, including the parallel execution interfaces and the other features that are independent of the execution interfaces. We saw that the high-level execution interfaces map to the common message-driven, fork-join, and SIMD layers that are found in many parallel applications. We then discussed how to get a copy of TBB and verify that our environment is correctly set up by writing, compiling, and executing very simple examples. We concluded the chapter by building a more complete example that uses all three high-level execution interfaces.

We are now ready to walk through the key support for parallel programming in the next few chapters: Generic Parallel Algorithms (Chapter 2), Flow Graphs (Chapter 3), Parallel STL (Chapter 4), Synchronization (Chapter 5), Concurrent Containers (Chapter 6), and Scalable Memory Allocation (Chapter 7).