APPENDIX

C TBB

This appendix provides a concise introduction to Intel Threading Building Blocks (Intel TBB). It covers the subset used by this book. A good introduction is available in the O'Reilly Nutshell Book on TBB, which covers the essentials of TBB [Rei07]. The book was published in 2007 when TBB version 2.0 appeared, so some newer features are not covered. It is nevertheless a solid introduction to TBB. For a more complete guide, see the TBB Reference, Tutorial, and Design Patterns documents, which can be downloaded from `http://threadingbuildingblocks.org/`.

TBB is a collection of components that outfits C++ for parallel programming. Figure C.1 illustrates these components. At the heart of TBB is a task scheduler, which is most often used indirectly via the parallel algorithms in TBB, such as `tbb::parallel_for`. The rest of TBB provides thread-aware memory allocation, portable synchronization primitives, scalable containers, and a variety of useful utilities. Each part is important for parallelism. Indeed the non-tasking features are intended for use with other parallelism frameworks such as Cilk Plus, ArBB, and OpenMP, so that those frameworks do not have to duplicate key functionality.

C.1 UNIQUE CHARACTERISTICS

TBB shares many of the key attributes of Cilk Plus as enumerated in Section B.1, but it differs form Cilk Plus on several points:

- TBB is designed to work without any compiler changes, and thus be quickly portable to new platforms. As a result, TBB has been ported to a multitude of key operating systems and processors, and code written with TBB can likewise be easily ported.
- As a consequence of avoiding any need for compiler support, TBB does not have direct support for vector parallelism. However, TBB combined with array notation or `#pragma simd` from Cilk Plus or auto-vectorization can be an effective tool for exploiting both thread and vector parallelism.
- TBB is designed to provide comprehensive support for C++ developers in one package. It supports multiple paradigms of parallel programming. It goes beyond the strict fork–join model of Cilk Plus by supporting pipelines, dataflow, and unstructured task graphs. The additional power that these features bring is sometimes worth the additional complexity they bring to a program.
- TBB is intended to provide low-level services such as memory allocation and atomic operations that can be used by programs using other frameworks, including Cilk Plus.

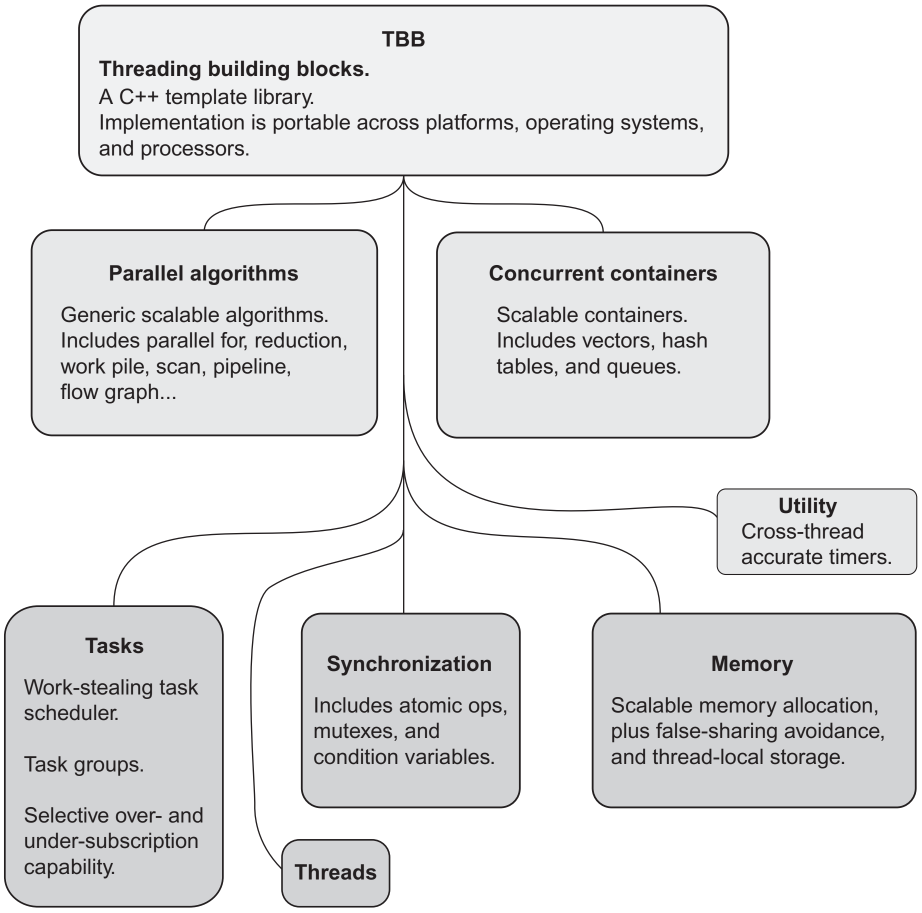
TBB
Threading building blocks.
A C++ template library.
Implementation is portable across platforms, operating systems, and processors.
Parallel algorithms
Generic scalable algorithms. Includes parallel for, reduction, work pile, scan, pipeline, flow graph...
Concurrent containers
Scalable containers. Includes vectors, hash tables, and queues.
Utility
Cross-thread accurate timers.
Tasks
Work-stealing task scheduler.
Task groups.
Selective over- and under-subscription capability.
Synchronization
Includes atomic ops, mutexes, and condition variables.
Memory
Scalable memory allocation, plus false-sharing avoidance, and thread-local storage.
Threads

FIGURE C.1

Overview of Threading Building Blocks.

TBB is an active open source project. It is widely adopted and often cited in articles about parallelism in C++. It continues to grow as the parallel ecosystem evolves.

C.2 USING TBB

Include the header `<tbb/tbb.h>` to use TBB in a source file. All public identifiers are in namespace `tbb` or `tbb::flow`. In the following descriptions, the phrase “in parallel” indicates that parallelism is permitted if resources allow, but is not mandated. As with Cilk Plus, the license to ignore unnecessary parallelism allows the TBB task scheduler to use parallelism efficiently.

C.3 parallel_for

The function template `parallel_for` maps a functor across range of values. The template takes several forms. The simplest is:

```
tbb::parallel_for(first,last,f)
```

where f is a functor. It evaluates the expression $f(i)$ in parallel for all i in the half-open interval [*first*,*last*), Both *first* and *last* must be of the same integral type. It is a parallel equivalent of:

```
for (auto i=first; i<last; ++i) f(i);
```

A slight variation specifies a stride:

```
tbb::parallel_for(first,last,stride,f)
```

It is like the previous version, except that the possible values of *i* step by *stride*, starting with *first*. This form is a parallel equivalent of:

```
for (auto i=first; i<last; i+=stride ) f(i);
```

Another form of `parallel_for` takes two arguments:

```
tbb::parallel_for(range,f)
```

It decomposes *range* into subranges and applies f to each subrange, in parallel. Hence, the programmer has the opportunity to optimize f to operate on an entire subrange instead of a single index. This version in effect exposes the tiled implementation of the map pattern used by TBB.

This form of parallel for also generalizes the parallel map pattern beyond one-dimensional ranges. The argument *range* can be any *recursively splittable range* type. A type `R` is such a type if it has the following methods:

`R::R(const R&)`	Copy constructor.
`R:: R()`	Destructor.
`bool R::is_divisible() const`	True if splitting constructor can be called, false otherwise.
`bool R::empty() const`	True if range is empty, false otherwise.
`R::R(R& r, split)`	Splitting constructor. It splits range `r` into two subranges. One of the subranges is the newly constructed range. The other subrange is overwritten onto `r`.

The implementation of `parallel_for` uses these methods to implement a generic recursive map in the spirit of Listing 8.1.

C.3.1 blocked_range

The most commonly used recursive range is `tbb::blocked_range`. It is typically used with integral types or random-access iterator types. For example, `blocked_range<int>(0,8)` represents the index range $\{0,1,2,3,4,5,6,7\}$. An optional third argument called the *grainsize* specifies the maximum size for splitting. It defaults to 1. For example, the following snippet splits a range of size 30 with grainsize 20 into two indivisible subranges of size 15:

```
// Construct half-open interval [0,30) with grainsize of 20
blocked_range<int> r(0,30,20);
assert(r.is_divisible());
// Call splitting constructor
blocked_range<int> s(r);
// Now r=[0,15) and s=[15,30) and both have a grainsize 20
// Inherited from the original value of r
assert(!r.is_divisible());
assert(!s.is_divisible());
```

Listing 4.2 on page 126 shows an example that uses `blocked_range` with `parallel_for`.

A two-dimensional variant is called `tbb::blocked_range2d`. It permits using a single `parallel_for` to iterate over two dimensions at once, which sometimes yields better cache behavior than nesting two one-dimensional instances of `parallel_for`.

C.3.2 Partitioners

The range form of `parallel_for` takes an optional *partitioner* argument, which lets the programmer specify performance-related tactics [RVK08]. The argument can have one of three types:

- `auto_partitioner`: The runtime will try to subdivide the range sufficiently to balance load, but no further. This behavior is the same as when no partitioner is specified.
- `simple_partitioner`: The runtime must subdivide the range into subranges as finely as possible; that is, method `is_divisible` will be false for the final subranges.
- `affinity_partitioner`: Request that the assignment of subranges to underlying threads be similar to a previous invocation of `parallel_for` or `parallel_reduce` with the same `affinity_partitioner` object.

These partitioners also work with `parallel_reduce`.

An invocation of `parallel_for` with a `simple_partitioner` looks like:

```
parallel_for(r,f,simple_partitioner());
```

This partitioner is useful in two scenarios:

- The functor f uses a fixed amount of memory for temporary storage, and hence cannot deal with subranges of arbitrary size. For example, if r is a `blocked_range`, the partitioner guarantees that f is invoked on subranges not exceeding the grainsize of r.
- The work for $f(r)$ is highly unbalanced in a way that fools the `auto_partitioner` heuristic into not dividing work finely enough to balance load.

An `affinity_partitioner` can be used for **cache fusion** (Section 4.4). Unlike the other two partitioners, it carries state. The state holds information for replaying the assignment of subranges to threads. Listing C.1 shows an example of its use in a common pattern: serially iterating a map. In the listing, variable `ap` enables cache fusion of each map to the next map. Because it is carrying information between serial iterations, it must be declared outside the serial loop.

```
void relax(
   double* a, // Pointer to array of data
   double* b, // Pointer to temporary storage
   size_t n,   // Number of data elements
   int iterations //Number of serial iterations
) {
   assert(iterations%2==0);
   // Partitioner should be declared outside the loop
   tbb::affinity_partitioner ap;
   // Serial loop around a parallel loop
   for( size_t t=0; t<iterations; ++t ) {
      tbb::parallel_for(
         tbb::blocked_range<size_t>(1,n-1),
         [=]( tbb::blocked_range<size_t> r ) {
            size_t e = r.end();
#pragma simd
            for( size_t i=r.begin(); i<e; ++i )
               b[i] = (a[i-1]+a[i]+a[i+1])*(1/3.0);
         },
         ap);
      std::swap(a,b);
   }
}
```

LISTING C.1

Example of `affinity_partitioner`. TBB uses the variable `ap` to remember which threads ran which subranges of the previous invocation of `parallel_for` and biases execution toward replaying that assignment. The `pragma simd` is for showmanship. It makes the impact of the partitioner more dramatic by raising arithmetic performance so that memory bandwidth becomes the limiting resource.

C.4 parallel_reduce

Function template `parallel_reduce` performs a reduction over a recursive range. It has several forms. The form used in this book is:

```
T result = tbb::parallel_reduce(
   range,
   identity,
   subrange_reduction,
   combine);
```

where:

- `range` is a recursive range as for `parallel_for`, such as `blocked_range`.
- `identity` is the identity element of type *T*. The type of this argument determines the type used to accumulate the reduction value, so be careful about what type it has.

- `subrange_reduction` is a functor such that *subrange_reduction*(*subrange*,*init*) returns a reduction value over *init* and *subrange*. The type of *subrange* is the type of the *range* argument to `parallel_reduce`. The type of *init* is *T*, and the returned reduction value must be convertible to type *T*. Do not forget to include the contribution of *init* to the reduction value.
- `combine` is a functor such that *combine*(*x*, *y*) takes two arguments of type *T* and returns a reduction value for them. This function must be associative but does not need to be commutative.

Listings 5.5 and 5.6 in Section 5.3.4 show example invocations. The latter listing demonstrates how to do accumulation at higher precision than the values being reduced.

An alternative way to do reduction is via class `tbb::enumerable_thread_specific`, as demonstrated in Section 11.3. General advice on which to use:

- If type *T* takes little space and is cheap to copy, or the combiner operation is non-commutative, use `parallel_reduce`.
- If type *T* is large and expensive to copy *and* the combiner operation is commutative, use `enumerable_thread_specific`.

C.5 parallel_deterministic_reduce

Template function `parallel_deterministic_reduce` is a variant of `parallel_reduce` that is **deterministic** even when the combiner operation is non-associative. The result is *not* necessarily the same as left-to-right serial reduction, even when executed with a single worker, because the template uses a fixed tree-like reduction order for a given input.

As of this writing, `parallel_deterministic_reduce` is a "preview feature" that must be enabled by setting the preprocessory symbol `TBB_PREVIEW_DETERMINISTIC_REDUCE=1` either on the compiler command line or *before* including TBB headers in a source file.

C.6 parallel_pipeline

Template function `parallel_pipeline` is used for building a **pipeline** of serial and parallel stages. See Section 9.4.1 for details and Listing 12.2 for an example.

C.7 parallel_invoke

Template function `parallel_invoke` evaluates a fixed set of functors in parallel. For example,

```
tbb::parallel_invoke(f,g,h);
```

evaluates the expressions `f()`, `g()`, and `h()` in parallel and waits until they all complete. Anywhere from 2 to 10 functors are currently supported. Listing 13.3 (page 302) and Listing 15.4 (page 322) show uses of `parallel_invoke`. Both listings cross-reference similar code in Cilk Plus, so you can compare the syntactic difference.

C.8 task_group

Class task_group runs an arbitrary number of functors in parallel. Listing C.2 shows an example.

In general, a single task_group should not be used to run a large number of tasks, because it can become a sequential bottleneck. Consider using parallel_for for a large number of tasks.

If one of the functors throws an exception, the task group is cancelled. This means that any tasks in the group that have not yet started will not start at all, but all currently running tasks will keep going. After all running tasks in the group complete, one of the exceptions thrown by the tasks will be thrown to the caller of wait. Hence, if nested parallelism is created by nesting task_group, the exception propagates up the task tree until it is caught.

Listing 8.12 on page 235 shows a use of task_group.

C.9 task

Class tbb::task is the lowest-level representation of a task in TBB. It is designed primarily for efficient execution, not convenience, because it serves as a foundation, and thus should impose minimal performance penalty. Higher level templates such as parallel_for and task_group provide

```
// Item in a linked list
class morsel {
public:
    void munch();
    morsel* next;
};

// Apply method munch to each item in a linked
// list rooted at p
void munch_list( morsel* p ) {
    tbb::task_group g;
    while( p ) {
        // Call munch on an item
        g.run( [=]{p->munch();} );
        // Advance to the next item
        p = p->next;
    }
    // Wait for all tasks to complete
    g.wait();
}
```

LISTING C.2

Using task_group.

convenient interfaces. Tasks can be spawned explicitly, or implicitly when all of their predecessor tasks complete. See the discussion of Listing 8.13 on pages 236–237 for how to use it.

C.9.1 empty_task

A `tbb::empty_task` is a `task` that does nothing. It is sometimes used for synchronization purposes, as in Listing 8.13 on pages 236–237.

C.10 atomic

Atomic objects have update operations that appear to happen instantaneously, as a single indivisible task. They are often used for lock-free synchronization. Atomic objects can be declared as instances of the class template `tbb::atomic<`*T*`>`, where *T* is an integral, enum, or pointer type. Listing C.3 shows an example use case.

```
float array[N];
tbb::atomic<int> count;

void append( float value ) {
   array[count++] = value;
}
```

LISTING C.3

Example of using atomic<int> as a counter.

If *m* threads execute `count++` at the same time, and its initial value is *k*, each thread will get a distinct result value drawn from the set $k, k+1, \ldots, k+m-1$, and afterward `count` will have value $k+m$. This, is true even if the threads do this simultaneously. Thus, despite the lack of mutexes, the code correctly appends items to the array.

In the example it is critical to use the value returned by `count++` and not reread `count`, because another thread might intervene and cause the reread value to be different than the result of `count++`.

Here is a description of the atomic operations supported by a variable `x` declared as a `tbb::atomic <X>`:

- *read, write*: Reads and writes on `x` are atomic. This property is not always true of non-atomic types. For example, on hardware with a natural word size of 32 bits, often reads and writes of 64-bit values are not atomic, even if executed by a single instruction.
- *fetch-and-add*: The operations `x+=k`, `x−=k`, `++x`, `x++`, `−−x`, and `x−−` have the usual meaning, but atomically update `x`. The expression `x.fetch_and_add(k)` is equivalent to `(x+=k)−k`.
- *exchange*: The operation `x.fetch_and_store(y)` atomically performs `x=y` and returns the previous value of `x`.

```
// Node in a linked list
struct node {
   float data;
   node* next;
};

// Root of a linked list
tbb::atomic<node*> root;

// Atomically prepend node a to the list
void add_to_list( node* a ) {
   for(;;) {
      // Take snapshot of root
      node* b = root;
      // Use the snapshot as link for a
      a->next = b;
      // Update root with a if root is still equal to b
      if( root.compare_and_swap(a,b)==b ) break;
      // Otherwise start over and try again
   }
}

// Atomically grab pointer to the entire list and reset root to NULL
node* grab_list() {
   return root.fetch_and_store((node*)NULL);
}
```

LISTING C.4

Using atomic operations on a list.

- *compare-and-swap*: The operation `x.compare_and_swap(y,z)` atomically performs `if(x==z) x=y`, and returns the original value of `x`. The operation is said to succeed if the assignment happens. Code can check for success by testing whether the return value equals `z`.

Listing C.4 shows uses of compare-and-swap and exchange to manipulate a linked list.

Doing more complicated list operations atomically is beyond the scope of this appendix.

In particular, implementing *pop* with a compare-and-swap loop scheme similar to the one in `add_to_list` requires special care to avoid a hazard called the *ABA problem* [Mic04]. The code shown has a benign form of the ABA problem, which happens when:

1. A thread executes `node* b=a`, and `a` was NULL.
2. Another threads executes `add_to_list` and `grab_list`.
3. The thread in step 1 executes `root.compare_and_swap(a,b)`. The compare-and-swap sees that $a == NULL$ and succeeds, just as if no other thread intervened.

The point is that a successful compare-and-swap does *not* mean that no thread intervened. Here, there is no harm done because as long as `root==a->next` when the compare-and-swap succeeds, the resulting list is correct. But, in other operations on linked structures, the effects can corrupt the structure or even cause invalid memory operations on freed memory.

Compare-and-swap loops also require care if there might be heavy contention. If P threads execute a compare-and-swap loop to update a location, $P-1$ threads will fail and have to try again. Then $P-2$ threads will fail, and so forth. The net burden is $\Theta(P^2)$ attempts and corresponding memory traffic, which can saturate the memory interconnect. One way to avoid the problem is *exponential backoff*—wait after each compare-and-swap fails, and double the wait after each subsequent failure.

C.11 `enumerable_thread_specific`

An object *e* of type `enumerable_thread_specific<T>` has a separate instance (or "local view") of `T` for each thread that accesses it. The expression *e*`.local()` returns a reference to the local view for the calling thread. Thus, multiple threads can operate on a `enumerable_thread_specific` without locking. The expression *e*`.combine(`*combine*`)` returns a reduction over the local view. See Section 11.3 for more details on how to use `enumerable_thread_specific`.

C.12 NOTES ON C++11

Though TBB works fine with C++98, it is simpler to use with C++11. In particular, C++11 introduces **lambda expressions** (Section D.2) and `auto` declarations (Section D.1) that simplify use of TBB and other template libraries. Lambda expressions are already implemented in the latest versions of major C++ compilers. We strongly recommend using them to teach, learn, and use TBB, because once you get past the novelty, they make TBB code easier to write and easier to read.

Additionally, TBB implements most of some C++11 features related to threading, thus providing an immediate migration path for taking advantage of these features even before they are implemented by C++ compilers. This path is further simplified by the way that TBB's injection of these features into namespace `std` is optional.

These features are:

- `std::mutex`: A mutex with a superset of the C++11 interface. The superset includes TBB's interface for mutexes.
- `std::lock_guard`: C++11 support for exception-safe scoped locking.
- `std::thread`: A way to create a thread and wait for it to complete. Sometimes threads really are a better solution than tasks, particularly if the "work" must be preemptively scheduled or mostly involves waiting for something to happen. Also, note that threads provide mandatory parallelism, which may be important when interacting with the outside world or in a user interface. Tasks provide optional parallelism, which is better for efficient computation.
- `std::condition_variable`: A way to wait until the state protected by a mutex meets a condition.

The parts of the C++11 interface not implemented in TBB are those that involve time intervals, since those would have involved implementing the C++11 time facilities. However, TBB does have equivalents to this functionality, based on TBB's existing `tick_count` interface for time.

A condition variable solves the problem of letting a thread wait until a state protected by a mutex meets a condition. It is used when threads need to wait for some other thread to update some state protected by a mutex. The waiting thread(s) acquire the mutex, check the state, and decide whether to wait. They wait on an associated condition variable. The `wait` member function atomically releases the mutex and starts the wait. Another thread acquires mutex associated with the condition, modifies the state protected by the mutex, and then signals one or all of the waiter(s) when it is done. Once the mutex is released, the waiters reacquire the mutex and can recheck the state to see if they can proceed or need to continue waiting.

Condition variables should be the method of choice to have a thread wait until a condition changes. TBB makes this method of choice portable to more operating systems.

C.13 HISTORY

The development of TBB was done at Intel and with the involvement of one of the authors of this book, Arch Robison. We can therefore recount the history of TBB from a personal perspective.

TBB was first available as a commercial library from Intel in the summer of 2006, not long after Intel shipped its first dual-core processors. It provided a much needed comprehensive answer to the question, "What must be fixed or added to C++ for parallel programming?" TBB's key programming abstractions for parallelism focused on logical specification of parallelism via algorithm templates. By also including a task-stealing scheduler, a thread-aware memory allocator, portable mutexes, global timestamps, and concurrent containers, TBB provided what was needed to program for parallelism in C++. The first release was primarily focused on strict **fork–join** or loop-type **data parallelism**.

The success of Intel TBB would, however, have been limited if it had remained a proprietary solution. Even during the release of version 1.0, Intel was in discussions with early customers on the future direction of TBB in both features and licensing.

Watching and listening to early adopters, such as Autodesk Maya, highlighted that much of the value of TBB was not only for data parallelism but also for more general parallelism using tasks, pipelines, scalable memory allocation, and lower-level constructs like synchronization primitives. Intel also received encouragement to make TBB portable by creating and supporting it via an open source project.

This customer feedback and encouragement led, only a year later, to version 2.0, which included a GPL v2 with the runtime exception version of both the source and binaries, as well as maintaining the availability of non-GPL binaries. Intel's customers had said that this would maximize adoption, and the results have definitely shown they were right.

Intel increased the staffing on TBB, worked proactively to build a community to support the project, and continued to innovate with new usage models and features over the next few years. We have been amazed and humbled by the response of such users as Adobe Systems, Avid, Epic Games, Dream-Works, and many others, along with that of other community members. TBB now has a very large user community and has had contributions that have led to Intel TBB being ported to many operating systems, platforms, and processors. We appreciate Intel's willingness to let us prove that an open source project initiated by Intel, yet supporting non-x86 processors, not only made sense—but would be very popular with developers. We've definitely proven that!

Through the involvement of customers and community, TBB has grown to be the most feature-rich and comprehensive solution for parallel application development available today. It has also become the most popular!

The TBB project was grown by a steady addition of ports to a wide variety of machines and operating systems and the addition of numerous new features that have added to the applicability and power of TBB.

TBB was one of the inspirations for Microsoft's Task Parallel Library (TPL) for .NET and Microsoft's Parallel Patterns Library (PPL) for C++. Intel and Microsoft have worked jointly to specify and implement a common subset of functionality shared by TBB and Microsoft's Parallel Patterns Library (PPL). In some cases, Intel and Microsoft have exchanged implementations and tests to ensure compatibility. An appendix of *The TBB Reference Manual* summarizes the common subset.

The most recent version of TBB, version 4.0, adds a powerful capability for expressing parallelism as data flowing through a graph. Use of TBB continues to grow, and the open source project enjoys serious support from Intel and others.

The Intel Cilk Plus project complements TBB by supplying C interfaces, simpler syntax, better opportunity for compiler optimization, and data parallel operations that lead to effective vectorization. None of these would be possible without direct compiler support. Intel briefly considered calling Cilk Plus simply "compiled TBB." While this conveyed the desire to extend TBB for the objectives mentioned, it proved complicated to explain the name so the name Cilk Plus was introduced. The full interoperability between TBB and Cilk Plus increases the number of options for software developers without adding complications. Like TBB, Intel has open sourced Cilk Plus to help encourage adoption and contribution to the project. TBB and Cilk Plus are sister projects at Intel.

C.14 SUMMARY

Intel Threading Building Blocks is a widely used and highly portable template library that provides a comprehensive set of solutions to programs using tasks in C++. It also provides a set of supporting functionality that can be used with or without the tasking infrastructure, such as concurrency-safe STL-compatible data structures, memory allocation, and portable atomics. Although we focus on tasks in this book due to their increased machine independence, safety, and scalability over threads, TBB also implements a significant subset of the C++11 standard's thread support, including platform-independent mutexes and condition variables. Much more information is available at `http://threadingbuildingblocks.org`.