CHAPTER 6

Data Structures for Concurrency

In the previous chapter, we shared how much we dislike locks. We dislike them because they tend to make our parallel programs less effective by limiting scaling. Of course, they can be a "necessary evil" when needed for correctness; however, we are well advised to structure our algorithms to minimize the need for locks. This chapter gives us some tools to help. Chapters 1–4 focused on scalable algorithms. A common characteristic is that they avoided or minimized locking. Chapter 5 introduced explicit synchronization methods, including locks, for when we need them. In the next two chapters, we offer ways to avoid using explicit synchronization by relying on features of TBB. In this chapter, we will discuss data structures with a desire to avoid locks. This chapter discusses concurrent containers to help address critical data structure considerations for concurrency. A related topic, the use of thread local storage (TLS), was already covered in Chapter 5.

This chapter and the next chapter cover the key pieces of TBB that help coordination of data between threads while avoiding the explicit synchronization found in Chapter 5. We do this to nudge ourselves toward coding in a manner that has proven ability to scale. We favor solutions where the implementations have been carefully crafted by the developers of TBB (to help motivate the importance of this for correctness, we discuss the A-B-A problem starting on page 200). We should remain mindful that the choice of algorithm can have a profound effect on parallel performance and the ease of implementation.

M. Voss, R. Asenjo, J. Reinders, *Pro TBB*, https://doi.org/10.1007/978-1-4842-4398-5_6

CHOOSE ALGORITHMS WISELY: CONCURRENT CONTAINERS ARE NOT A CURE-ALL

Parallel data access is best when it stems from a clear parallelism strategy, a key part of which is proper choice of algorithms. Controlled access, such as that offered by concurrent containers, comes at a cost: making a container "highly concurrent" is not free and is not even always possible. TBB offers concurrent containers when such support can work well in practice (queues, hash tables, and vectors). TBB does not attempt to support concurrency for containers such as "lists" and "trees," where fine-grained sharing will not scale well – the better opportunity for parallelism lies in revising algorithms and/or data structure choices.

Concurrent containers offer a thread-safe version for containers where concurrent support can work well in parallel programs. They offer a higher performance alternative to using a serial container with a coarse-grained lock around it, as discussed in the previous chapter (Chapter 5). TBB containers generally provide fine-grained locking, or lockless implementations, or sometimes both.

Key Data Structures Basics

If you are familiar with hash tables, unordered maps, unordered sets, queues, and vectors, then you may want to skip this section and resume reading with the "Concurrent Containers". To help review the key fundamentals, we provide a quick introduction to key data structures before we jump into talking about how TBB supports these for parallel programming.

Unordered Associative Containers

Unordered associative containers, in simple English, would be called a *collection*. We could also call them "sets." However, technical jargon has evolved to use the words map, set, and hash tables for various types of collections.

Associative containers are data structures which, given a *key*, can find a *value*, associated with that *key*. They can be thought of as a fancy array, we call them an "associative array." They take indices that are more complex than a simple series of numbers. Instead of `Cost[1]`, `Cost[2]`, `Cost[3]`, we can think of `Cost[Glass of Juice]`, `Cost[Loaf of Bread]`, `Cost[Puppy in the Window]`.

Our associative containers can be specialized in two ways:

1. **Map vs. Set:** Is there a *value*? Or just a *key*?
2. **Multiple values:** Can two items with the same *keys* be inserted in the same collection?

Map vs. Set

What we call a "map" is really just a "set" with a value attached. Imagine a basket of fruits (Apple, Orange, Banana, Pear, Lemon). A *set* containing fruits could tell us if we had a particular type of fruit in the basket. A simple *yes* or *no*. We could add a fruit type into the basket or remove it. A *map* adds to this a value, often a data structure itself with information. With a *map* of a fruit type into a collection (fruit basket), we could choose to keep a count, a price, and other information. Instead of a simple *yes* or *no*, we can ask about `Cost[Apple]` or `Ripeness[Banana]`. If the value is a structure with multiple fields, then we could query multiple things such as cost, ripeness, and color.

Multiple Values

Inserting something into a map/set using the same *key* as an item already in the map is not allowed (ensuring uniqueness) in the regular "map" or "set" containers but is allowed in the "multimap" and "multiset" versions. In the "multiple" versions, duplicates are allowed, but we lose the ability to look up something like `Cost[Apple] because the` *key* `Apple` is no longer unique in a map/set.

Hashing

Everything we have mentioned (associative arrays, map/set, single/multiple) is commonly implemented using *hash* functions. To understand what a *hash* function is, it is best to understand its motivation. Consider an associative array `LibraryCardNumber[Name of Patron]`. The array `LibraryCardNumber` returns the library card number for a patron given the name (specified as a string of characters) that is supplied as the index. One way to implement this associative array would be with a linked list of elements. Unfortunately, looking up an element would require searching the list one by one for a match. That might require traversing the entire list, which is highly inefficient in a parallel program because of contention over access to the share list

structure. Even without parallelism, when inserting an item verification that there is no other entry with the same *key* requires searching the entire list. If the list has thousands or millions of patrons, this can easily require excessive amounts of time. More exotic data structures, such as trees, can improve some but not all these issues.

Imagine instead, a vast array in which to place data. This array is accessed by a traditional `array[integer]` method. This is very fast. All we need, is a magical *hash* function that takes the index for the associative array (*Name of Patron*) and turns it into the `integer` we need.

Unordered

We did start with the word *unordered* as a qualifier for the type of *associative containers* that we have been discussing. We could certainly sort the keys and access these containers in a given order. Nothing prevents that. For example, the *key* might be a person's name, and we want to create a phone directory in alphabetical order.

The word *unordered* here does not mean we cannot be programming with an ordering in mind. It does mean that the data structure (container) itself does not maintain an order for us. If there is a way to "walk" the container (*iterate* in C++ jargon), the only guarantee is that we will visit each member of the container once and only once, but the order is not guaranteed and can vary run-to-run, or machine-to-machine, and so on.

Concurrent Containers

TBB provides highly concurrent container classes that are useful for all C++ threaded applications; the TBB concurrent container classes can be used with any method of threading, including TBB of course!

The C++ Standard Template Library was not originally designed with concurrency in mind. Typically, C++ STL containers do not support concurrent updates, and therefore attempts to modify them concurrently may result in corrupted containers. Of course, STL containers can be wrapped in a coarse-grained `mutex` to make them safe for concurrent access by letting only one thread operate on the container at a time. However, that approach eliminates concurrency and thereby restricts parallel speedup if done in performance critical code. Examples of protecting with mutexes were shown in Chapter 5, to protect increments of elements in a histogram. Similar protection of non-thread-safe STL routines can be done to avoid correctness issues. If not done in performance

critical sections, then performance impact may be minimal. This is an important point: conversion of containers to TBB concurrent containers should be motivated by need. Data structures that are used in parallel should be designed for concurrency to enable scaling for our applications.

The concurrent containers in TBB provide functionality similar to containers provided by the Standard Template Library (STL), but do so in a thread-safe way. For example, the `tbb::concurrent_vector` is similar to the `std::vector` class but lets us safely grow the vector in parallel. We don't need a concurrent container if we only read from it in parallel; it is only when we have parallel code that modifies a container that we need special support.

TBB offers several container classes, meant to replace corresponding STL containers in a compatible manner, that permit multiple threads to simultaneously invoke certain methods on the same container. These TBB containers offer a much higher level of concurrency, via one or both of the following methods:

- Fine-grained locking: Multiple threads operate on the container by locking only those portions they really need to lock (as the histogram examples in Chapter 5 showed us). As long as different threads access different portions, they can proceed concurrently.
- Lock-free techniques: Different threads account and correct for the effects of other interfering threads.

It is worth noting that TBB concurrent containers do come at a small cost. They typically have higher overheads than regular STL containers, and therefore operations on them may take slightly longer than on the STL containers. When the possibility of concurrent access exists, concurrent containers should be used. However, if concurrent access is not possible, the use of STL containers is advised. This is, we use concurrent containers when the speedup from the additional concurrency that they enable outweighs their slower sequential performance.

The interfaces for the containers remain the same as in STL, except where a change is required in order to support concurrency. We might jump ahead for a moment and make this a good time to consider a classic example of why some interfaces are not thread-safe – *and this is an important point to understand!* The classic example (see Figure 6-9) is the need for a new *pop-if-not-empty* capability (called `try_pop`) for queues in place of relying on a code sequence using STL *test-for-empty* followed by a *pop* if the test returned not-empty. The danger in such STL code is that another thread

might be running, empty the container (after original thread's test, but before pop) and therefore create a race condition where the *pop* may actually block. That means the STL code is not thread-safe. We could throw a lock around the whole sequence to prevent modification of the queue between our test and our pop, but such locks are known to destroy performance when used in parallel parts of an application. Understanding this simple example (Figure 6-9) will help illuminate what is required to support parallelism well.

Like STL, TBB containers are templated with respect to an allocator argument. Each container uses that allocator to allocate memory for user-visible items. The default allocator for TBB is the scalable memory allocator supplied with TBB (discussed in Chapter 7). Regardless of the allocator specified, the implementation of the container may also use a different allocator for strictly internal structures.

TBB currently offers the following concurrent containers:

- Unordered associative containers
  - Unordered map (including unordered multimap)
  - Unordered set (including unordered multiset)
  - Hash table
- Queue (including bounded queue and priority queue)
- Vector

WHY DO TBB CONTAINERS ALLOCATOR ARGUMENTS DEFAULT TO TBB?

Allocator arguments are supported with all TBB containers, and they default to the TBB scalable memory allocators (see Chapter 7).

The containers default to using a mix of `tbb::cache_aligned_allocator` and `tbb:tbb_allocator`. We document the defaults in this chapter, but Appendix B of this book and the TBB header files are resources for learning the defaults. There is no requirement to link in the TBB scalable allocator library (see Chapter 7), as the TBB containers will silently default to using `malloc` when the library is not present. However, we should link with the TBB scalable allocator because the performance will likely be better from just linking in – especially easy using it as a proxy library as explained in Chapter 7.

Class name *and C++11 connection notes*	Concurrent traversal and insertion.	Keys have a value associated with them.	Support concurrent erasure	Built-in locking.	No visible locking (lock-free interface).	Identical items allowed to be inserted.	[ ] and at accessors
`concurrent_hash_map` *Predates C++11.*	✓	✓	✓	✓	✗	✗	✗
`concurrent_unordered_map` *Closely resembles the C++11 `unordered_map`.*	✓	✓	✗	✗	✓	✗	✓
`concurrent_unordered_multimap` *Closely resembles the C++11 `unordered_multimap`.*	✓	✓	✗	✗	✓	✓	✗
`concurrent_unordered_set` *Closely resembles the C++11 `unordered_set`.*	✓	✗	✗	✗	✓	✗	✗
`concurrent_unordered_multiset` *Closely resembles the C++11 `unordered_multiset`.*	✓	✗	✗	✗	✓	✓	✗

Figure 6-1. *Comparison of concurrent unordered associative containers*

Concurrent Unordered Associative Containers

Unordered associative containers are a group of class templates that implement hash table variants. Figure 6-1 lists these containers and their key differentiating features. Concurrent unordered associative containers can be used to store arbitrary elements, such as integers or custom classes, because they are templates. TBB offers implementations of unordered associative containers that can perform well concurrently.

A hash map (also commonly called a hash table) is a data structure that maps keys to values using a hash function. A hash function computes an index from a key, and the index is used to access the "bucket" in which value(s) associated with the key are stored.

Choosing a good hash function is very important! A perfect hash function would assign each key to a unique bucket so there will be no *collisions* for different keys. In practice, however, hash functions are not perfect and will occasionally generate the same index for more than one key. These collisions require some form of

accommodation by the hash table implementation, and this will introduce some overhead – hash functions should be designed to minimize collisions by hashing inputs into a nearly even distribution across the buckets.

The advantage of a hash map comes from the ability to, in the average case, provide O(1) time for searches, insertions, and keys. The advantage of a TBB hash map is support for concurrent usage both for correctness and performance. This assumes that a good hash function is being used – one that does not cause many collisions for the keys that are used. The theoretical worst case of O(n) remains whenever an imperfect hash function exists, or if the hash table is not well-dimensioned.

Often hash maps are, in actual usage, more efficient than other table lookup data structures including search trees. This makes hash maps the data structure of choice for many purposes including associative arrays, database indexing, caches, and sets.

---

concurrent_hash_map

TBB supplies `concurrent_hash_map`, which maps keys to values in a way that permits multiple threads to concurrently access values via `find`, `insert`, and `erase` methods. As we will discuss later, `tbb:: concurrent_hash_map` was designed for parallelism, and therefore its interfaces are thread-safe unlike the STL `map/set` interfaces we will cover later in this chapter.

The keys are unordered. There is at most one element in a `concurrent_hash_map` for each key. The key may have other elements in flight but not in the map. Type `HashCompare` specifies how keys are hashed *and* how they are compared for equality. As is generally expected for hash tables, if two keys are equal, then they must hash to the same hash code. This is why `HashCompare` ties the concept of comparison and hashing into a single object instead of treating them separately. Another consequence of this is that we need to not change the hash code of a key while the hash table is nonempty.

A `concurrent_hash_map` acts as a container of elements of type `std::pair<const Key,T>`. Typically, when accessing a container element, we are interested in either updating it or reading it. The template class `concurrent_hash_map` supports these two purposes respectively with the classes `accessor` and `const_accessor` that act as smart pointers. An accessor represents update (write) access. As long as it points to an element, all other attempts to look up that key in the table block until the accessor is done. A `const_accessor` is similar, except that it represents read-only access. Multiple accessors can point to the same element at the same time. This feature can greatly improve concurrency in situations where elements are frequently read and infrequently updated.

We share a simple example of code using the `concurrent_hash_map` container in Figures 6-2 and 6-3. We can improve the performance of this example by reducing the lifetime of the element access. The methods `find` and `insert` take an `accessor` or `const_accessor` as an argument. The choice tells `concurrent_hash_map` whether we are asking for update or read-only access. Once the method returns, the access lasts until the `accessor` or `const_accessor` is destroyed. Because having access to an element can block other threads, try to shorten the lifetime of the `accessor` or `const_accessor`. To do so, declare it in the innermost block possible. To release access even sooner than the end of the block, use method `release`. Figure 6-5 shows a rework of the loop body from Figure 6-2 that uses `release` instead of depending upon destruction to end thread lifetime. The method `remove(key)` can also operate concurrently. It implicitly requests write access. Therefore, before removing the key, it waits on any other extant accesses on key.

```
#include <tbb/concurrent_hash_map.h>
#include <tbb/blocked_range.h>
#include <tbb/parallel_for.h>
#include <string>

// Structure that defines hashing and comparison operations for
user's type.
struct MyHashCompare {
  static size_t hash( const std::string& x ) {
    size_t h = 0;
    for( const char* s = x.c_str(); *s; ++s )
      h = (h*17)^*s;
    return h;
  }
  //! True if strings are equal
  static bool equal( const std::string& x, const std::string& y
) {
    return x==y;
  }
};

// A concurrent hash table that maps strings to ints.
typedef tbb::concurrent_hash_map<std::string,int,MyHashCompare>
StringTable;

// Function object for counting occurrences of strings.
struct Tally {
  StringTable& table;
  Tally( StringTable& table_ ) : table(table_) {}
  void operator()(
    const tbb::blocked_range<std::string*> range ) const {
    for( std::string* p=range.begin(); p!=range.end(); ++p ) {
      StringTable::accessor a;
      table.insert( a, *p );
      a->second += 1;
    }
  }
};
```

Figure 6-2. *Hash Table example, part 1 of 2*

```
const size_t N = 10;

std::string Data[N] = { "Hello", "World", "TBB", "Hello",
      "So Long", "Thanks for all the fish", "So Long",
      "Three", "Three", "Three" };

void main() {
  // Construct empty table.
  StringTable table;

  // Put occurrences into the table
  tbb::parallel_for(
      tbb::blocked_range<std::string*>( Data, Data+N, 1000 ),
      Tally(table) );

  // Display the occurrences using a simple walk
  // (note: concurrent_hash_map does not offer const_iterator)
  // see a problem with this code???
  // read "Iterating thorough these structures is
  // asking for trouble" coming up in a few pages
  for( StringTable::iterator i=table.begin();
       i!=table.end();
       ++i )
    printf("%s %d\n",i->first.c_str(),i->second);
}
```

Figure 6-3. *Hash Table example, part 2 of 2*

```
Three 3
So Long 2
Hello 2
TBB 1
World 1
Thanks for all the fish 1
```

Figure 6-4. *Output of the example program in Figures 6-2 and 6-3*

```
for( std::string* p=range.begin(); p!=range.end(); ++p ) {
  StringTable::accessor a;
  table.insert( a, *p );
  a->second += 1;
  a.release();
}
```

Figure 6-5. *Revision to Figure 6-2 to reduce accessor lifetime hoping to improve scaling*

PERFORMANCE TIPS FOR HASH MAPS

- Always specify an initial size for the hash table. The default of one will scale horribly! Good sizes definitely start in the hundreds. If a smaller size seems correct, then using a lock on a small table will have an advantage in speed due to cache locality.
- Check your hash function – and be sure that there is good pseudo-randomness in the low-order bits of the hash value. In particular, you should not use pointers as keys because generally a pointer will have a set number of zero bits in the low-order bits due to object alignment. If this is the case, it is strongly recommended that the pointer be divided by the size of the type it points too, thereby shifting out the always zero bits in favor of bits that vary. Multiplication by a prime number, and shifting out some low order bits, is a strategy to consider. As with any form of hash table, keys that are equal must have the same hash code, and the ideal hash function distributes keys uniformly across the hash code space. Tuning for an optimal hash function is definitely application specific, but using the default supplied by TBB tends to work well.
- Do not use accessors if they can be avoided and limit their lifetime as much as possible when accessors are needed (see example of this in Figure 6-5). They are effectively fine-grained locks, inhibit other threads while they exist, and therefore potentially limit scaling.
- Use the TBB memory allocator (see Chapter 7). Use `scalable_allocator` as the template argument for the container if you want to enforce its usage (not allow a fallback to malloc) – at least a good sanity check during development when testing performance.

Concurrent Support for map/multimap and set/multiset Interfaces

Standard C++ STL defines unordered_set, unordered_map, unordered_multiset, and unordered_multimap. Each of these containers differs only by the constraints which are placed on their elements. Figure 6-1 is a handy reference to compare the five choices we have for concurrent map/set support including the tbb::concurrent_hash_map which we used in our code examples (Figures 6-2 through 6-5).

STL does not define anything called "hash" because C++ did not originally define a hash table. Interest in adding hash table support to STL was widespread, so there were widely used versions of STL that were extended to include hash table support, including those by SGI, gcc, and Microsoft. Without a standard, there ended up being variation in what "hash table" or "hash maps" came to mean to C++ programmers in terms of capabilities and performance. Starting with C++11, a hash table implementation was added to the STL, and the name unordered_map was chosen for the class to prevent confusion and collisions with pre-standard implementations. It could be said that the name unordered_map is more descriptive as it hints at the interface to the class and the unordered nature of its elements.

The original TBB hash table support predates C++11, called tbb::concurrent_hash_map. This hash function remains quite valuable and did not need to change to match the standard. TBB now includes support for unordered_map and unordered_set support to mirror the C++11 additions, with the interfaces augmented or adjusted only as needed to support concurrent access. Avoiding a few parallel-unfriendly interfaces is part of the "nudging us" to effective parallel programming. Appendix B has an exhaustive coverage of the details, but the three noteworthy adjustments for better parallel scaling are as follows:

- Methods requiring C++11 language features (e.g., rvalue references) are omitted.
- The erase methods for C++ standard functions are prefixed with unsafe_ to indicate that they are not concurrency safe (because concurrent erasure is only supported for concurrent_hash_map). This does not apply to concurrent_hash_map because it *does* support concurrent erasure.

- The bucket methods (count of buckets, max count of buckets, size of buckets, and support to iterate through the buckets) are prefixed with `unsafe_` as a reminder that they are not concurrency safe with respect to insertion. They are supported for compatibility with STL but should be avoided if possible. If used, they should be protected from being used concurrently with insertions occurring. These interfaces do not apply to `concurrent_hash_map` because the TBB designers avoided such functions.

Built-In Locking vs. No Visible Locking

The containers `concurrent_hash_map` and `concurrent_unordered_*` have some differences concerning the locking of accessed elements. Therefore, they may behave very differently under contention. The accessors of `concurrent_hash_map` are essentially locks: `accessor` is an exclusive lock, and `const_accessor` is a shared lock. Lock-based synchronization is built into the usage model for the container, protecting not only container integrity but to some degree data integrity as well. Code in Figure 6-2 uses an `accessor` when performing an insert into the table.

Iterating Through These Structures Is Asking for Trouble

We snuck in some concurrency unsafe code at the end of Figure 6-3 when we iterated through the hash table to dump it out. If insertions or deletions were made while we walked the table, this could be problematic. In our defense, we will just say "it is debug code – we do not care!" But, experience has taught us that it is all too easy for code like this to creep into non-debug code. Beware!

The TBB designers left the iterators available for `concurrent_hash_map` for debug purposes, but they purposefully did not tempt us with iterators as return values from other members.

Unfortunately, STL tempts us in ways we should learn to resist. The `concurrent_unordered_*` containers are different than `concurrent_hash_map` – the API follows the C++ standard for associative containers (keep in mind, the original TBB `concurrent_hash_map` predates any standardization by C++ for concurrent containers). The operations to add or find data return an iterator, so this tempts us to iterate with it. In a parallel program, we risk this being simultaneously with other operations on the map/set. If we give into temptation, protecting data integrity is completely left to us

as programmers, the API of the container does not help. One could say that the C++ standard containers offer additional flexibility but lack the built-in protection that `concurrent_hash_map` offers. The STL interfaces are easy enough to use concurrently, if we avoid the temptation to use the iterators returned from an add or find operation for anything other than referencing the item we looked up. If we give into the temptation (we should not!), then we have a lot of thinking to do about concurrent updates in our application. Of course, if there are no updates happening – only lookups – then there are no parallel programming issues with using the iterators.

Concurrent Queues: Regular, Bounded, and Priority

Queues are useful data structures where items are added or removed from the queue with operations known as push (add) and pop (remove). The unbounded queue interfaces provide a "try pop" which tells us if the queue was empty and no value was popped from the queue. This steers us away from writing our own logic to avoid a blocking pop by testing empty – an operation that is not thread-safe (see Figure 6-9). Sharing a queue between multiple threads can be an effective way to pass work items from thread to thread – a queue holding "work" to do could have work items added to request future processing and removed by tasks that want to do the processing.

Normally, a queue operates in a first-in-first-out (FIFO) fashion. If I start with an empty queue, do a `push(10)` and then a `push(25)`, then the first pop operation will return `10`, and the second pop will return a `25`. This is much different than the behavior of a stack, which would usually be last-in-first-out. But, we are not talking about stacks here!

We show a simple example in Figure 6-6 which clearly shows that the pop operations return the values in the same order as the push operations added them to the queue.

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

int myarray[10] = { 16, 64, 32, 512, 1, 2, 512, 8, 4, 128 };

void pval(int test, int val) {
  if (test) {
    std::cout << " " << val;
  } else {
    std::cout << " ***";
  }
}

void simpleQ() {
  tbb::concurrent_queue<int> queue;
  int val;

  for( int i=0; i<10; ++i )
    queue.push(myarray[i]);

  std::cout << "Simple  Q   pops are";

  for( int i=0; i<10; ++i )
    pval( queue.try_pop(val), val );

  std::cout << std::endl;
}

int main() {
  simpleQ();
//  boundedQ();
//  prioQ();
//  prioQgt();
  return 0;
}

Output is:
Simple  Q   pops are 16 64 32 512 1 2 512 8 4 128
```

Figure 6-6. *Example of using the simple (FIFO) queue*

There are two twists offered for queues: *bounding* and *priorities*. Bounding adds the concept of enforcing a limit on the size of a queue. This means that a push might not be possible if the queue is full. To handle this, the bounded queue interfaces offer us ways to have a push wait until it can add to the queue, or have a "try to push" operation that does the push if it can or lets us know the queue was full. A bounded queue is by default unbounded! If we want a bounded queue, we need to use `concurrent_bounded_queue` and call method `set_capacity` to set the size for the queue. We show in Figure 6-7 a simple usage of bounded queue in which only the first six items pushed made it into the queue. We could add a test on `try_push` and do something. In this case, we have the program print ******* when the pop operation finds that the queue was empty.

```
void boundedQ() {
  tbb::concurrent_bounded_queue<int> queue;
  int val;

  queue.set_capacity(6);

  for( int i=0; i<10; ++i )
    queue.try_push(myarray[i]);

  std::cout << "Bounded Q   pops are";

  for( int i=0; i<10; ++i )
    pval( queue.try_pop(val), val );

  std::cout << std::endl;
}

Output of the expanded program is:
Simple  Q   pops are 16 64 32 512 1 2 512 8 4 128
Bounded Q   pops are 16 64 32 512 1 2 *** *** *** ***
```

Figure 6-7. *This routine expands our program to show bounded queue usage*

A priority adds a twist to first-in-first-out by effectively sorting items in the queue. The default priority, if we do not specify one in our code, is `std::less<T>`. This means that a pop operation will return the highest valued item in the queue.

Figure 6-8 shows two examples of priority usage, one defaulting to `std:: less<int>` while the other specifying `std::greater<int>` explicitly.

```
void prioQ() {
  tbb::concurrent_priority_queue<int> queue;
  int val;

  for( int i=0; i<10; ++i )
    queue.push(myarray[i]);

  std::cout << "Prio    Q   pops are";

  for( int i=0; i<10; ++i )
    pval( queue.try_pop(val), val );

  std::cout << std::endl;
}

void prioQgt() {
  tbb::concurrent_priority_queue<int,std::greater<int>> queue;
  int val;

  for( int i=0; i<10; ++i )
    queue.push(myarray[i]);

  std::cout << "Prio    Qgt pops are";

  for( int i=0; i<10; ++i )
    pval( queue.try_pop(val), val );

  std::cout << std::endl;
}

Output of the expanded program is:
Simple  Q   pops are 16 64 32 512 1 2 512 8 4 128
Bounded Q   pops are 16 64 32 512 1 2 *** *** *** ***
Prio    Q   pops are 512 512 128 64 32 16 8 4 2 1
Prio    Qgt pops are 1 2 4 8 16 32 64 128 512 512
```

Figure 6-8. *These routines expand our program to show priority queueing*

As our examples in the prior three figures show, to implement these three variations on queues, TBB offers three container classes: `concurrent_queue`, `concurrent_bounded_queue`, and `concurrent_priority_queue`. All concurrent queues permit multiple threads to concurrently push and pop items. The interfaces are similar to STL `std::queue` or `std::priority_queue` except where it must differ to make concurrent modification of a queue safe.

The fundamental methods on a queue are `push` and `try_pop`. The `push` method works as it would with a `std::queue`. It is important to note that there is not support for `front` or `back` methods because they would not be safe in a concurrent environment since these methods return a reference to an item in the queue. In a parallel program, the front or back of a queue could be changed by another thread in parallel making the use of `front` or `back` meaningless.

Similarly, pop and testing for empty are not supported for unbounded queues – instead the method `try_pop` is defined to pop an item if it is available and return a `true` status; otherwise, it returns no item and a status of `false`. The test-for-empty and pop methods are combined into a single method to encourage thread-safe coding. For bounded queues, there is a non-blocking `try_push` method in addition to the potentially blocking `push` method. These help us avoid the `size` methods to inquire about the size of the queue. Generally, the `size` methods should be avoided, especially if they are holdovers from a sequential program. Since the size of a queue can change concurrently in a parallel program, the `size` method needs careful thought if it is used. For one thing, TBB can return a negative value for `size` methods when the queue empty and there are pending pop methods. The `empty` method is true when `size` is zero or less.

Bounding Size

For `concurrent_queue` and `concurrent_priority_queue`, capacity is unbounded, subject to memory limitations on the target machine. The `concurrent_bounded_queue` offers controls over bounds – a key feature being that a `push` method will block until the queue has room. A bounded queue is useful in slowing a supplier to match the rate of consumption instead of allowing a queue to grow unconstrained.

`concurrent_bounded_queue` is the only `concurrent_queue_*` container that offers a `pop` method. The `pop` method will block until an item becomes available. A `push` method can be blocking only with a `concurrent_bounded_queue` so this container type also offers a non-blocking method called `try_push`.

This concept of bounding to rate match, to avoid overflowing memory or overcommitting cores, also exists in Flow Graph (see Chapter 3) through the use of a `limiter_node`.

Priority Ordering

A priority queue maintains an ordering in the queue based on the priorities of individual queued items. As we mentioned earlier, a normal queue has a first-in-first-out policy, whereas a priority queue sorts its items. We can provide our own Compare to change the ordering from the default of `std::less<T>`. For instance, using `std::greater<T>` causes the smallest element to be the next to retrieved for a `pop` method. We did exactly that in our example code in Figure 6-8.

Staying Thread-Safe: Try to Forget About Top, Size, Empty, Front, Back

It is important to note that there is no `top` method, and we probably should avoid using `size` and `empty` methods. Concurrent usage means that the values from all three can change due to push/pop methods in other threads. Also, the `clear` and `swap` methods, while supported, are not thread-safe. TBB forces us to rewrite code using `top` when converting a `std::priority_queue` usage to `tbb::concurrent_priority_queue` because the element that would be returned could be invalidated by a concurrent pop. Because the return values are not endangered by concurrency, TBB does support `std::priority_queue` methods of `size`, `empty`, and `swap`. However, we recommend carefully reviewing the wisdom of using either function in a concurrent application, since a reliance on either is likely to be a hint that the code that needs rewriting for concurrency.

`std::` code, not thread safe

```
#include <iostream>
#include <queue>



void main() {
  int sum (0);
  int item;

  std::priority_queue<int> myPQ;

  for(int i=0; i<10001; i+=1) {
    myPQ.push(i);
  }

  while( !myPQ.empty() ) {
    sum += myPQ.top();
    myPQ.pop();
  }

  // prints "total: 50005000"
  std::cout << "total: "
            << sum << '\n';
}
```

`tbb::` code, thread safe

```
#include <iostream>

#include <tbb/concurrent_priority_queue.h>
#include <tbb/parallel_for.h>

void main() {
  int sum (0);
  int item;

  tbb::concurrent_priority_queue<int> myPQ;

  tbb::parallel_for(0,10001,1,
    [&](size_t i){myPQ.push(i);} );


  while( myPQ.try_pop(item) )
    sum += item;



  // prints "total: 50005000"
  std::cout << "total: "
            << sum << '\n';
}
```

Figure 6-9. *Motivation for* `try_pop` *instead of* `top` *and* `pop` *shown in a side-by-side comparison of STL and TBB priority queue code. Both will total 50005000 in this example without parallelism, but the TBB scales and is thread-safe.*

Iterators

For debugging purposes alone, all three concurrent queues provide limited iterator support (`iterator` and `const_iterator` types). This support is intended solely to allow us to inspect a queue during debugging. Both `iterator` and `const_iterator` types follow the usual STL conventions for forward iterators. The iteration order is from least recently pushed to most recently pushed. Modifying a queue invalidates any iterators that reference it. The iterators are relatively slow. They should be used only for debugging. An example of usage is shown in Figure 6-10.

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

int main() {
  tbb::concurrent_queue<int> queue;
  for( int i=0; i<10; ++i )
    queue.push(i);
  for( tbb::concurrent_queue<int>::const_iterator
       i(queue.unsafe_begin());
       i!=queue.unsafe_end();
       ++i )
    std::cout << *i << " ";
  std::cout << std::endl;
  return 0;
}
Output of this program is:
0 1 2 3 4 5 6 7 8 9
```

Figure 6-10. *Sample debugging code for iterating through a concurrent queue – note the* `unsafe_` *prefix on* `begin` *and* `end` *to emphasize the debug-only non-thread-safe nature of these methods.*

Why to Use This Concurrent Queue: The A-B-A Problem

We mentioned at the outset of this chapter that there is significant value in having containers that have been written by parallelism experts for us to "just use." None of us should want to reinvent good scalable implementations for each application. As motivation, we diverge to mention the A-B-A problem – a classic computer science example of parallelism gone wrong! At first glance, a concurrent queue might seem easy enough to simply write our own. It is not. Using the `concurrent_queue` from TBB, or any other well-researched and well-implemented concurrent queue, is a good idea. Humbling as the experience can be, we would not be the first to learn it is not as easy as we could naively believe. The update idiom (`compare_and_swap`) from Chapter 5 is inappropriate if the `A-B-A` problem (see sidebar) thwarts our intent. This is a frequent problem when trying to design a non-blocking algorithm for linked data structures, including a concurrent queue. The TBB designers have a solution to the A-B-A problem already packaged in the solutions for concurrent queues. We can just rely upon it. Of course, it is open source code so you can hunt around in the code to see the solution if you are feeling curious. If you do look in the source code, you'll see that arena management (subject of Chapter 12) has to deal with the ABA problem as well. Of course, you can just use TBB without needing to know any of this. We just wanted to

emphasize that working out concurrent data structures is not as easy as it might appear – hence the love we have for using the concurrent data structures supported by TBB.

THE A-B-A PROBLEM

Understanding the A-B-A problem is a key way to train ourselves to think through the implications of concurrency when designing our own algorithms. While TBB avoids the A-B-A problems while implementing concurrent queues and other TBB structures, it is a reminder that we need to "Think Parallel."

The A-B-A problem occurs when a thread checks a location to be sure the value is *A* and proceeds with an update only if the value was *A*. The question arises whether it is a problem if other tasks change the same location in a way that the first task does not detect:

1. A task reads a value *A* from *globalx*.
2. Other tasks change *globalx* from *A* to *B* and then back to *A*.
3. The task in step 1 does its `compare_and_swap`, reading *A* and thus not detecting the intervening change to *B*.

If the task erroneously proceeds under an assumption that the location has not changed since the task first read it, the task may proceed to corrupt the object or otherwise get the wrong result.

Consider an example with linked lists. Assume a linked list *W(1)→X(9)→Y(7)→Z(4)*, where the letters are the node locations and the numbers are the values in the nodes. Assume that some task transverses the list to find a node *X* to dequeue. The task fetches the next pointer, *X.next* (which is *Y*) with the intent to put it in *W.next*. However, before the swap is done, the task is suspended for some time.

During the suspension, other tasks are busy. They dequeue *X* and then happen to reuse that same memory and queue a new version of node *X* as well as dequeueing Y and adding Q at some point in time. Now, the list is *W(1)→X(2)→Q(3)→Z(4)*.

Once the original task finally wakes up, it finds that *W.next* still points to *X*, so it swaps out *W.next* to become *Y*, thereby making a complete mess out of the linked list.

Atomic operations are the way to go if they embody enough protection for our algorithm. If the A-B-A problem can ruin our day, we need to find a more complex solution. `tbb::concurrent_queue` has the necessary additional complexity to get this right!

When to NOT Use Queues: Think Algorithms!

Queues are widely used in parallel programs to buffer consumers from producers. Before using an explicit queue, we need to consider using `parallel_do` or `pipeline` instead (see Chapter 2). These options are often more efficient than queues for the following reasons:

- Queues are inherently bottlenecks because they must maintain an order.
- A thread that is popping a value will stall if the queue is empty until a value is pushed.
- A queue is a passive data structure. If a thread pushes a value, it could take time until it pops the value, and in the meantime the value (and whatever it references) becomes *cold* in cache. Or worse yet, another thread pops the value, and the value (and whatever it references) must be moved to the other processor core.

In contrast, `parallel_do` and `pipeline` avoid these bottlenecks. Because their threading is implicit, they optimize use of worker threads so that they do other work until a value shows up. They also try to keep items *hot* in cache. For example, when another work item is added to a `parallel_do`, it is kept local to the thread that added it unless another idle thread can steal it before the *hot* thread processes it. This way, items are more often processed by the *hot* thread thereby reducing delays in fetching data.

Concurrent Vector

TBB offers a class called `concurrent_vector`. A `concurrent_vector<T>` is a dynamically growable array of `T`. It is safe to grow a `concurrent_vector` even while other threads are also operating on elements of it, or even growing it themselves. For safe concurrent growing, `concurrent_vector` has three methods that support common uses of dynamic arrays: `push_back`, `grow_by`, and `grow_to_at_least`.

Figure 6-11 shows a simple usage of `concurrent_vector`, and Figure 6-12 shows, in the dump of the vector contents, the effects of parallel threads having added concurrently. The outputs from the same program would prove identical if sorted into numerical order.

When to Use tbb::concurrent_vector Instead of std::vector

The key value of `concurrent_vector<T>` is its ability to grow a vector concurrently and its ability to guarantee that elements do not move around in memory.

concurrent_vector does have more overhead than std::vector. So, we should use concurrent_vector when we need the ability to dynamically resize it while other accesses are (or might be) in flight or require that an element never move.

```
#include <iostream>

#include <tbb/concurrent_vector.h>
#include <tbb/parallel_for.h>

void oneway() {
//  Create a vector containing integers
    tbb::concurrent_vector<int> v = {3, 14, 15, 92};

    // Add more integers to vector IN PARALLEL
    for( int i = 100; i < 1000; ++i ) {
      v.push_back(i*100+11);
      v.push_back(i*100+22);
      v.push_back(i*100+33);
      v.push_back(i*100+44);
    }

    // Iterate and print values of vector (debug use only)
    for(int n : v) {
      std::cout << n << std::endl;
    }
}

void allways() {
//  Create a vector containing integers
    tbb::concurrent_vector<int> v = {3, 14, 15, 92};

    // Add more integers to vector IN PARALLEL
    tbb::parallel_for( 100, 999, [&](int i){
      v.push_back(i*100+11);
      v.push_back(i*100+22);
      v.push_back(i*100+33);
      v.push_back(i*100+44);
      });

    // Iterate and print values of vector (debug use only)
    for(int n : v) {
      std::cout << n << std::endl;
    }
}
```

Figure 6-11. *Concurrent vector small example*

```
3                    3
14                   14
15                   15
92                   92
10011                10011
. . .                . . .
84911                72611
84922                91211
84933                87111
84944                72622
85011                91222
85022                87122
85033                72633
85044                91233
. . .                . . .
99933                99833
99944                99844
```

Figure 6-12. *The left side is output generated while using* `for` *(not parallel), and the right side shows output when using* `parallel_for` *(concurrent pushing into the vector).*

Elements Never Move

A `concurrent_vector` never moves an element until the array is cleared, which can be an advantage over the STL `std::vector` even for single-threaded code. Unlike a `std::vector`, a `concurrent_vector` never moves existing elements when it grows. The container allocates a series of contiguous arrays. The first reservation, growth, or assignment operation determines the size of the first array. Using a small number of elements as initial size incurs fragmentation across cache lines that may increase element access time. `shrink_to_fit()` merges several smaller arrays into a single contiguous array, which may improve access time.

Concurrent Growth of concurrent_vectors

While concurrent growing is fundamentally incompatible with ideal exception safety, `concurrent_vector` does offer a practical level of exception safety. The element type must have a destructor that never throws an exception, and if the constructor can throw an exception, then the destructor must be nonvirtual and work correctly on zero-filled memory.

The `push_back(x)` method safely appends `x` to the vector. The `grow_by(n)` method safely appends `n` consecutive elements initialized with `T()`. Both methods return an iterator pointing to the first appended element. Each element is initialized with `T()`. The following routine safely appends a C string to a shared vector:

```
void Append( concurrent_vector<char>& vector,
             const char* string ) {
    size_t n = strlen(string)+1;
    std::copy( string, string+n, vector.grow_by(n) );
}
```

`grow_to_at_least(n)` grows a vector to size `n` if it is shorter. Concurrent calls to the growth methods do not necessarily return in the order that elements are appended to the vector.

`size()` returns the number of elements in the vector, which may include elements that are still undergoing concurrent construction by methods `push_back`, `grow_by`, or `grow_to_at_least`. The previous example uses `std::copy` and iterators, not `strcpy` and pointers, because elements in a `concurrent_vector` might not be at consecutive addresses. It is safe to use the iterators while the `concurrent_vector` is being grown, as long as the iterators never go past the current value of `end()`. However, the iterator may reference an element undergoing concurrent construction. Therefore, we are required to synchronize construction and access.

Operations on `concurrent_vector` are concurrency safe with respect to growing, not for clearing or destroying a vector. Never invoke `clear()` if there are other operations in flight on the `concurrent_vector`.

Summary

In this chapter, we discussed three key data structures (hash/map/set, queues, and vectors) that have support in TBB. This support from TBB offers thread-safety (okay to run concurrently) as well as an implementation that scales well. We offered advice on things to avoid, because they tend to cause trouble in parallel programs – including using the iterators returned by map/set for anything other than the one item that was looked up. We reviewed the A-B-A problem both as a motivation for using TBB instead of writing our own and as an excellent example of the thinking we need to do when parallel programs share data.

As with other chapters, the complete APIs are detailed in Appendix B, and the code shown in figures is all downloadable.

Despite all the wonderful support for parallel use of containers, we cannot emphasize enough the concept that thinking through algorithms to minimize synchronization of any kind is critical to high performance parallel programming. If you can avoid sharing data structures, by using `parallel_do`, `pipeline`, `parallel_reduce`, and so on, as we mentioned in the section "When to NOT Use Queues: Think Algorithms!" – you may find your programs scale better. We mention this in multiple ways throughout this book, because thinking this through is important for the most effective parallel programming.