Outfitting C++ for Multi-core Processor Parallelism

Intel Threading Building Blocks

James Reinders
Foreword by Alexander Stepanov

O'REILLY®

CHAPTER 1

Why Threading Building Blocks?

Intel Threading Building Blocks offers a rich and complete approach to expressing parallelism in a C++ program. It is a library that helps you leverage multi-core processor performance without having to be a threading expert. Threading Building Blocks is not *just* a threads-replacement library; it represents a higher-level, task-based parallelism that abstracts platform details and threading mechanisms for performance and scalability.

This chapter introduces Intel Threading Building Blocks and how it stands out relative to other options for C++ programmers. Although Threading Building Blocks relies on templates and the C++ concept of generic programming, this book does not require any prior experience with these concepts or with threading.

Chapter 2 explains the challenges of parallelism and introduces key concepts that are important for using Threading Building Blocks. Together, these first two chapters set up the foundation of knowledge needed to make the best use of Threading Building Blocks.

Download and Installation

You can download Intel Threading Building Blocks, along with instructions for installation, from *http://threadingbuildingblocks.org* or *http://intel.com/software/products/tbb*.

Threading Building Blocks was initially released in August 2006 by Intel, with prebuilt binaries for Windows, Linux, and Mac OS X. Less than a year later, Intel provided more ports and is now working with the community to provide additional ports. The information on how to install Threading Building Blocks comes with the product downloads.

Overview

Multi-core processors are becoming common, yet writing even a simple `parallel_for` loop is tedious with existing threading packages. Writing an efficient *scalable* program is much harder. Scalability embodies the concept that a program should see benefits in performance as the number of processor cores increases.

Threading Building Blocks helps you create applications that reap the benefits of new processors with more and more cores as they become available.

Threading Building Blocks is a library that supports scalable parallel programming using standard C++ code. It does not require special languages or compilers. The ability to use Threading Building Blocks on virtually any processor or any operating system with any C++ compiler makes it very appealing.

Threading Building Blocks uses templates for common parallel iteration patterns, enabling programmers to attain increased speed from multiple processor cores without having to be experts in synchronization, load balancing, and cache optimization. Programs using Threading Building Blocks will run on systems with a single processor core, as well as on systems with multiple processor cores. Threading Building Blocks promotes scalable data parallel programming. Additionally, it fully supports nested parallelism, so you can build larger parallel components from smaller parallel components easily. To use the library, you specify tasks, not threads, and let the library map tasks onto threads in an efficient manner. The result is that Threading Building Blocks enables you to specify parallelism far more conveniently, and with better results, than using raw threads.

Benefits

As mentioned, the goal of a programmer in a modern computing environment is scalability: to take advantage of both cores on a dual-core processor, all four cores on a quad-core processor, and so on. Threading Building Blocks makes writing scalable applications much easier than it is with traditional threading packages.

There are a variety of approaches to parallel programming, ranging from the use of platform-dependent threading primitives to exotic new languages. The advantage of Threading Building Blocks is that it works at a higher level than raw threads, yet does not require exotic languages or compilers. You can use it with any compiler supporting ISO C++. This library differs from typical threading packages in these ways:

Threading Building Blocks enables you to specify tasks *instead of threads*
: Most threading packages require you to create, join, and manage threads. Programming directly in terms of threads can be tedious and can lead to inefficient programs because threads are low-level, heavy constructs that are close to the hardware. Direct programming with threads forces you to do the work to efficiently map logical tasks onto threads. In contrast, the Threading Building

Blocks runtime library automatically schedules tasks onto threads in a way that makes efficient use of processor resources. The runtime is very effective at load-balancing the many tasks you will be specifying.

By avoiding programming in a raw native thread model, you can expect better portability, easier programming, more understandable source code, and better performance and scalability in general.

Indeed, the alternative of using raw threads directly would amount to programming in the *assembly language of parallel programming*. It may give you maximum flexibility, but with many costs.

Threading Building Blocks targets threading for performance

Most general-purpose threading packages support many different kinds of threading, such as threading for asynchronous events in graphical user interfaces. As a result, general-purpose packages tend to be low-level tools that provide a foundation, not a solution. Instead, Threading Building Blocks focuses on the particular goal of parallelizing computationally intensive work, delivering higher-level, simpler solutions.

Threading Building Blocks is compatible *with other threading packages*

Threading Building Blocks can coexist seamlessly with other threading packages. This is very important because it does not force you to pick among Threading Building Blocks, OpenMP, or raw threads for your entire program. You are free to add Threading Building Blocks to programs that have threading in them already. You can also add an OpenMP directive, for instance, somewhere else in your program that uses Threading Building Blocks. For a particular part of your program, you will use one method, but in a large program, it is reasonable to anticipate the convenience of mixing various techniques. It is fortunate that Threading Building Blocks supports this.

Using or creating libraries is a key reason for this flexibility, particularly because libraries are often supplied by others. For instance, Intel's Math Kernel Library (MKL) and Integrated Performance Primitives (IPP) library are implemented internally using OpenMP. You can freely link a program using Threading Building Blocks with the Intel MKL or Intel IPP library.

Threading Building Blocks emphasizes scalable, data-parallel programming

Breaking a program into separate functional blocks and assigning a separate thread to each block is a solution that usually does not scale well because, typically, the number of functional blocks is fixed. In contrast, Threading Building Blocks emphasizes *data-parallel* programming, enabling multiple threads to work most efficiently together. Data-parallel programming scales well to larger numbers of processors by dividing a data set into smaller pieces. With data-parallel programming, program performance increases (scales) as you add processors. Threading Building Blocks also avoids classic bottlenecks, such as a global task queue that each processor must wait for and lock in order to get a new task.

Threading Building Blocks relies on generic programming

Traditional libraries specify interfaces in terms of specific types or base classes. Instead, Threading Building Blocks uses generic programming, which is defined in Chapter 12. The essence of generic programming is to write the best possible algorithms with the fewest constraints. The C++ Standard Template Library (STL) is a good example of generic programming in which the interfaces are specified by *requirements* on types. For example, C++ STL has a template function that sorts a sequence abstractly, defined in terms of iterators on the sequence.

Generic programming enables Threading Building Blocks to be flexible yet efficient. The generic interfaces enable you to customize components to your specific needs.

Comparison with Raw Threads and MPI

Programming using a raw thread interface, such as POSIX threads (pthreads) or Windows threads, has been an option that many programmers of shared memory parallelism have used. There are wrappers that increase portability, such as Boost Threads, which are a very portable raw threads interface. Supercomputer users, with their thousands of processors, do not generally have the luxury of shared memory, so they use message passing, most often through the popular Message Passing Interface (MPI) standard.

Raw threads and MPI expose the control of parallelism at its lowest level. They represent the *assembly languages of parallelism*. As such, they offer maximum flexibility, but at a high cost in terms of programmer effort, debugging time, and maintenance costs.

In order to program parallel machines, such as multi-core processors, we need the ability to express our parallelism without having to manage every detail. Issues such as optimal management of a thread pool, and proper distribution of tasks with load balancing and cache affinity in mind, should not be the focus of a programmer when working on expressing the parallelism in a program.

When using raw threads, programmers find basic coordination and data sharing to be difficult and tedious to write correctly and efficiently. Code often becomes very dependent on the particular threading facilities of an operating system. Raw thread-level programming is too low-level to be intuitive, and it seldom results in code designed for scalable performance. Nested parallelism expressed with raw threads creates a lot of complexities, which I will not go into here, other than to say that these complexities are handled for you with Threading Building Blocks.

Another advantage of tasks versus logical threads is that tasks are much lighter weight. On Linux systems, starting and terminating a task is about 18 times faster than starting and terminating a thread. On Windows systems, the ratio is more than 100-fold.

With threads and with MPI, you wind up mapping tasks onto processor cores explicitly. Using Threading Building Blocks to express parallelism with tasks allows developers to express more concurrency and finer-grained concurrency than would be possible with threads, leading to increased scalability.

Comparison with OpenMP

Along with Intel Threading Building Blocks, another promising abstraction for C++ programmers is OpenMP. The most successful parallel extension to date, OpenMP is a language extension consisting of pragmas, routines, and environment variables for Fortran and C programs. OpenMP helps users express a parallel program and helps the compiler generate a program reflecting the programmer's wishes. These directives are important advances that address the limitations of the Fortran and C languages, which generally prevent a compiler from automatically detecting parallelism in code.

The OpenMP standard was first released in 1997. By 2006, virtually all compilers had some level of support for OpenMP. The maturity of implementations varies, but they are widespread enough to be viewed as a natural companion of Fortran and C languages, and they can be counted upon when programming on any platform.

When considering it for C programs, OpenMP has been referred to as "excellent for Fortran-style code written in C." That is not an unreasonable description of OpenMP since it focuses on loop structures and C code. OpenMP offers nothing specific for C++. The loop structures are the same loop nests that were developed for vector supercomputers—an earlier generation of parallel processors that performed tremendous amounts of computational work in very tight nests of loops and were programmed largely in Fortran. Transforming those loop nests into parallel code could be very rewarding in terms of results.

A proposal for the 3.0 version of OpenMP includes tasking, which will liberate OpenMP from being solely focused on long, regular loop structures by adding support for irregular constructs such as `while` loops and recursive structures. Intel implemented tasking in its compilers in 2004 based on a proposal implemented by KAI in 1999 and published as "Flexible Control Structures in OpenMP" in 2000. Until these tasking extensions take root and are widely adopted, OpenMP remains reminiscent of Fortran programming with minimal support for C++.

OpenMP has the programmer choose among three scheduling approaches (static, guided, and dynamic) for scheduling loop iterations. Threading Building Blocks does not require the programmer to worry about scheduling policies. Threading Building Blocks does away with this in favor of a single, automatic, divide-and-conquer approach to scheduling. Implemented with *work stealing* (a technique for moving tasks from loaded processors to idle ones), it compares favorably to dynamic or guided scheduling, but without the problems of a centralized dealer. Static scheduling

is sometimes faster on systems undisturbed by other processes or concurrent sibling code. However, divide-and-conquer comes close enough and fits well with nested parallelism.

The generic programming embraced by Threading Building Blocks means that parallelism structures are not limited to built-in types. OpenMP allows reductions on only built-in types, whereas the Threading Building Blocks `parallel_reduce` works on any type.

Looking to address weaknesses in OpenMP, Threading Building Blocks is designed for C++, and thus to provide the simplest possible solutions for the types of programs written in C++. Hence, Threading Building Blocks is not limited to statically scoped loop nests. Far from it: Threading Building Blocks implements a subtle but critical recursive model of task-based parallelism and generic algorithms.

Recursive Splitting, Task Stealing, and Algorithms

A number of concepts are fundamental to making the parallelism model of Threading Building Blocks intuitive. Most fundamental is the reliance on breaking problems up recursively as required to get to the right level of parallel tasks. It turns out that this works much better than the more obvious static division of work. It also fits perfectly with the use of task stealing instead of a global task queue. This is a critical design decision that avoids using a global resource as important as a task queue, which would limit scalability.

As you wrestle with which algorithm structure to apply for your parallelism (`for` loop, `while` loop, pipeline, divide and conquer, etc.), you will find that you want to combine them. If you realize that a combination such as a `parallel_for` loop controlling a *parallel set of pipelines* is what you want to program, you will find that easy to implement. Not only that, the fundamental design choice of recursion and task stealing makes this work yield efficient scalable applications.

It is a pleasant surprise to new users to discover how acceptable it is to code parallelism, even inside a routine that is used concurrently itself. Because Threading Building Blocks was designed to encourage this type of nesting, it makes parallelism easy to use. In other systems, this would be the start of a headache.

With an understanding of why Threading Building Blocks matters, we are ready for the next chapter, which lays out what we need to do in general to formulate a parallel solution to a problem.

CHAPTER 3
Basic Algorithms

This is *the* key chapter in learning Intel Threading Building Blocks. Here you will come to understand the recursion, task-stealing, and algorithm templates that Threading Building Blocks uniquely combines.

The most visible contribution of Threading Building Blocks is the *algorithm templates* covered in this chapter and the next chapter. This chapter introduces the simplest loop-oriented algorithms based on recursive ranges, and the next chapter expands on that with more advanced algorithm support. Future chapters offer details that round out features needed to make your use of Threading Building Blocks complete.

Threading Building Blocks offers the following types of generic parallel algorithms, which are covered in this chapter:

Loop parallelization

`parallel_for` and `parallel_reduce`
: Load-balanced, parallel execution of a fixed number of independent loop iterations

`parallel_scan`
: A template function that computes a prefix computation (also known as a scan) in parallel (`y[i] = y[i-1]` *op* `x[i]`)

The validity of the term *Building Block* is clearer when you see how fully Threading Building Blocks supports nested parallelism to enable you to build larger parallel components from smaller parallel components. A Quicksort example shown in Chapter 11, for instance, was implemented using `parallel_for` recursively. The recursion is implicit because of Threading Building Blocks' inherent concept of *splitting*, as embodied in the parallel iterator.

When you thoroughly understand why a recursive algorithm such as Quicksort should use the `parallel_for` template with a recursive range instead of using a recursion template, you will understand a great deal about how to apply Threading Building Blocks to your applications.

Understanding some fundamental concepts can make the parallelism model of Threading Building Blocks intuitive. Most fundamental is the reliance on breaking up problems recursively as required to get to the right level of parallel tasks. The proper degree of breakdown in a problem is embodied in a concept called *grain size*. Grain size started as a mysterious manual process, which has since been facilitated with some automation (heuristics) in the latest versions of Threading Building Blocks. This chapter offers rules of thumb, based on experience with Threading Building Blocks, for picking the best grain size.

Recursively breaking down a problem turns out to be much better than the more obvious static division of work. It also fits perfectly with the use of task stealing instead of a global task queue. Reliance on task stealing is a critical design decision that avoids implementing something as important as a task queue as a global resource that becomes a bottleneck for scalability.

Furthermore, as you wrestle to decide which algorithm template to apply to parallelism (`for` loop, `while` loop, pipeline, divide and conquer, etc.), you will find that you want to mix and nest them. More often than not, you realize that in a C++ program, a combination—such as a `parallel_for` loop controlling a parallel set of *pipelines*—is what you want to program. Threading Building Blocks makes such mixtures surprisingly easy to implement. Not only that, but the fundamental design choice of recursion and task stealing makes the resulting program work very well.

Initializing and Terminating the Library

Intel Threading Building Blocks components are defined in the `tbb` namespace. For brevity's sake, the namespace is explicit in the first mention of a component in this book, but implicit afterward.

Any thread that uses an algorithm template from the library or the task scheduler must have an initialized `tbb::task_scheduler_init` object. A thread may have more than one of these objects initialized at a time. The task scheduler shuts down when all `task_scheduler_init` objects terminate. By default, the constructor for `task_scheduler_init` does the initialization and the destructor does the termination. Thus, declaring a `task_scheduler_init` in `main()`, as in Example 3-1, both starts and shuts down the scheduler.

Example 3-1. Initializing the library

```
#include "tbb/task_scheduler_init.h"
using namespace tbb;

int main() {
    task_scheduler_init init;
     ...
    return 0;
}
```

The `using` directive in the example enables you to use the library identifiers without having to write out the namespace prefix `tbb` before each identifier. The rest of the examples assume that such a `using` directive is present.

Automatic startup/shutdown was not implemented because, based on Intel's experience in implementing OpenMP, we knew that parts are too problematic on some operating systems to do it behind the scenes. In particular, always knowing when a thread shuts down can be quite problematic.

Calling the initialization more than once will not cause the program to fail, but it is a bit wasteful and can cause a flurry of extra warnings from some debugging or analysis tools, such as the Intel Thread Checker.

The section "Mixing with Other Threading Packages" in Chapter 10 explains how to construct `task_scheduler_init` objects if your program creates threads itself using another interface.

The constructor for `task_scheduler_init` takes an optional parameter that specifies the number of desired threads, including the calling thread. The optional parameter can be one of the following:

- The value `task_scheduler_init::automatic`, which is the default when the parameter is not specified. It exists for the sake of the method `task_scheduler_init::initialize`.
- The value `task_scheduler_init::deferred`, which defers the initialization until the `task_scheduler_init::initialize(`*n*`)` method is called. The value *n* can be any legal value for the constructor's optional parameter.
- A *positive integer* specifying the number of threads to use.

The deferred form of the task scheduler allows you to create the `init` object at the right scope so that the destruction will happen when that scope is exited. The actual initialization call can then occur inside a subroutine without having the destruction implicit in the return cause a problem.

The argument should be specified only when doing scaling studies during development. Omit the parameter, or use `task_scheduler_init::automatic`, for production code. The reason for not specifying the number of threads in production code is that in a large software project, there is no way for various components to know how many threads would be optimal for other components. Hardware threads are a shared global resource. It is best to leave the decision of how many threads to use to the task scheduler.

The parameter is ignored if another `task_scheduler_init` object is active. Disagreements in the number of threads are resolved in favor of the first `task_scheduler_init` that has not been deferred.

Design your programs to try to create many more tasks than there are threads, and let the task scheduler choose the mapping from tasks to threads.

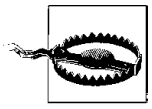
In general, you will let the library be terminated automatically by the destructor when all `task_scheduler_init` objects terminate.

For cases where you want more control, there is a method named `task_scheduler_init::terminate` for terminating the library early, before the `task_scheduler_init` is destroyed. Example 3-2 defers the decision of the number of threads to be used by the scheduler (line 3 defers, line 5 commits), and terminates it early (line 8).

Example 3-2. Early scheduler termination

```
1  int main( int argc, char* argv[] ) {
2      int nthread = strtol(argv[0],0,0);
3      task_scheduler_init init(task_scheduler_init::deferred);
4      if( nthread>=1 )
5        init.initialize(nthread);
6    ... code that uses task scheduler only if nthread>=1 ...
7      if( nthread>=1 )
8          init.terminate( );
9      return 0;
10 }
```

In Example 3-2, you can omit the call to `terminate( )` because the destructor for `task_scheduler_init` checks whether the `task_scheduler_init` was initialized and, if so, performs the termination.



The task scheduler is somewhat expensive to start up and shut down, so it's recommended that you put the `task_scheduler_init` in your main routine or when a thread is born, and do not try to create a scheduler every time you use a parallel algorithm template.

Loop Parallelization

The simplest form of scalable parallelism is a loop of iterations that can each run simultaneously without interfering with each other. The following sections demonstrate how to parallelize such simple loops.

`parallel_for` *and* `parallel_reduce`

: Load-balanced, parallel execution of a fixed number of independent loop iterations

`parallel_scan`

: A template function that computes a parallel prefix (`y[i] = y[i-1]` *op* `x[i]`)

parallel_for

Suppose you want to apply a function `Foo` to each element of an array, and it is safe to process each element concurrently. Example 3-3 shows the sequential code to do this.

Example 3-3. Original loop code

```
void SerialApplyFoo( float a[], size_t n ) {
    for( size_t i=0; i<n; ++i )
        Foo(a[i]);
}
```

The iteration space here is of type `size_t`, and it goes from `0` to `n-1`. The template function `tbb::parallel_for` breaks this iteration space into chunks and runs each chunk on a separate thread.

The first step in parallelizing this loop is to convert the loop body into a form that operates on a chunk. The form is a Standard Template Library (STL)-style function object, called the *body* object, in which `operator()` processes a chunk. Example 3-4 declares the body object.

Example 3-4. A class for use by a parallel_for

```
#include "tbb/blocked_range.h"

class ApplyFoo {
    float *const my_a;
public:
    void operator()( const blocked_range<size_t>& r ) const {
        float *a = my_a;
        for( size_t i=r.begin(); i!=r.end(); ++i )
            Foo(a[i]);
    }
    ApplyFoo( float a[] ) :
        my_a(a)
    {}
};
```

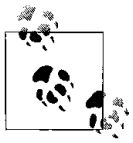
Note the iteration space argument to `operator()`. A `blocked_range<T>` is a template class provided by the library. It describes a one-dimensional iteration space over type `T`. Class `parallel_for` works with other kinds of iteration spaces, too. The library provides `blocked_range2d` for two-dimensional spaces. A little later in this chapter, in the section "Advanced Topic: Other Kinds of Iteration Spaces," I will explain how you can define your own spaces.

An instance of `ApplyFoo` needs member fields that remember all the local variables that were defined outside the original loop but were used inside it. Usually, the constructor for the body object will initialize these fields, though `parallel_for` does not

care how the body object is created. The template function `parallel_for` requires that the body object have a copy constructor, which is invoked to create a separate copy (or copies) for each worker thread. It also invokes the destructor to destroy these copies.

In most cases, the implicitly generated copy constructor and destructor work correctly. You may need an explicit destructor if you've designed it such that the destructor must perform some action, such as freeing memory. If the copy constructor and destructor are both implicit, there will be no such side effects. But, if the destructor is explicit, most likely the copy constructor will need to be explicit as well.

Because the body object might be copied, its `operator()` should not modify the body. Thus, in Example 3-4, the `operator()` function should not modify data member `my_a`. It *can* modify what `my_a` points to. This distinction is emphasized by declaring `my_a` as `const` and what it points to as (non-`const`) `float`. Otherwise, the modification might or might not become visible to the thread that invoked `parallel_for`, depending upon whether `operator()` is acting on the original or a copy. As a reminder of this nuance, `parallel_for` requires that the body object's `operator()` be declared as `const`.



Threading Building Blocks is designed to help guard against mistakes that would lead to failures: thus, `operator()` is required to be `const`-qualified as a syntactic guard against trying to accumulate side effects that would be lost by the thread-private copies.

The example `operator()` loads `my_a` into a local variable, `a`. Though it's not necessary, there are two reasons for doing this in the example:

Style

It makes the loop body look more like the original.

Performance

Sometimes, putting frequently accessed values into local variables helps the compiler optimize the loop better, because local variables are often easier for the compiler to track.

After you have written the loop body as a body object, invoke the template function `parallel_for`, as shown in Example 3-5.

Example 3-5. Use of parallel_for

```
#include "tbb/parallel_for.h"

void ParallelApplyFoo( float a[], size_t n ) {
    parallel_for(blocked_range<size_t>(0,n,YouPickAGrainSize), ApplyFoo(a) );
}
```

The blocked_range constructed here represents the entire iteration space from 0 to n-1, which parallel_for divides into subspaces for each processor. The general form of the constructor is:

```
blocked_range<T>(begin,end,grainsize)
```

The T specifies the value type. The arguments begin and end specify the iteration space in STL style as a half-open interval [begin,end). Half-open intervals are convenient because the empty set is easily represented by [X,X). If Y is less than X, a range [X,Y) is considered invalid and will raise an assertion if the debug macro TBB_DO_ASSERT is defined and is nonzero.

Half-Open Intervals

Intervals are specified using a square bracket [or] to indicate inclusion, or a rounded bracket (or) to indicate exclusion. An interval of [2,7] indicates the numbers 2, 3, 4, 5, 6, 7, whereas the interval (2,7) means 3, 4, 5, 6. The half-open interval [2,7) indicates 2, 3, 4, 5, 6.

Threading Building Blocks uses half-open intervals. This means that the interval [X,Y) effectively creates an iteration which will be covered by the loop for (i=X;i<Y;i++).

Example 3-6 defines a routine, ParallelAverage, using parallel_for, that sets output[i] to the average of input[i-1], input[i], and input[i+1], for $0 \le i < n$.

Example 3-6. Parallel average

```
#include "tbb/parallel_for.h"
#include "tbb/blocked_range.h"

using namespace tbb;

struct Average {
    float* input;
    float* output;
    void operator( )( const blocked_range<int>& range ) const {
        for( int i=range.begin(); i!=range.end( ); ++i )
            output[i] = (input[i-1]+input[i]+input[i+1])*(1/3.0f);
    }
};

// Note: The input must be padded such that input[-1] and input[n]
// can be used to calculate the first and last output values.
void ParallelAverage( float* output, float* input, size_t n ) {
    Average avg;
    avg.input = input;
    avg.output = output;
    parallel_for( blocked_range<int>( 0, n, 1000 ), avg );
}
```

Grain size

The third argument, `grainsize`, specifies the number of iterations for a *reasonable size* chunk to deal out to a processor. If the iteration space has more than `grainsize` iterations, `parallel_for` splits it into separate subranges that are scheduled separately.

The `grainsize` amortizes parallel scheduling overhead. Having a `grainsize` independent of the number of processors tends to keep, in common cases, the parallel scheduling overhead in constant proportion to real work. This is because the packaging-and-handling overhead is relatively constant per grain and therefore independent of the number of processors.

The `grainsize` enables you to avoid excessive parallel overhead. A parallel loop construct incurs overhead cost for every subrange. If the subranges are too small, the overhead may exceed the useful work. By specifying a grain size, you can limit the overhead. The `grainsize` effectively sets a minimum threshold for parallelization.

Figure 3-1 illustrates the impact of overhead by showing the useful work as lettered squares surrounded by the overhead of a grain of work (the darker surrounding areas). On the left, the problem is broken into four pieces (4X), and on the right, with a finer grain size, the problem is broken into 36 pieces (36X).

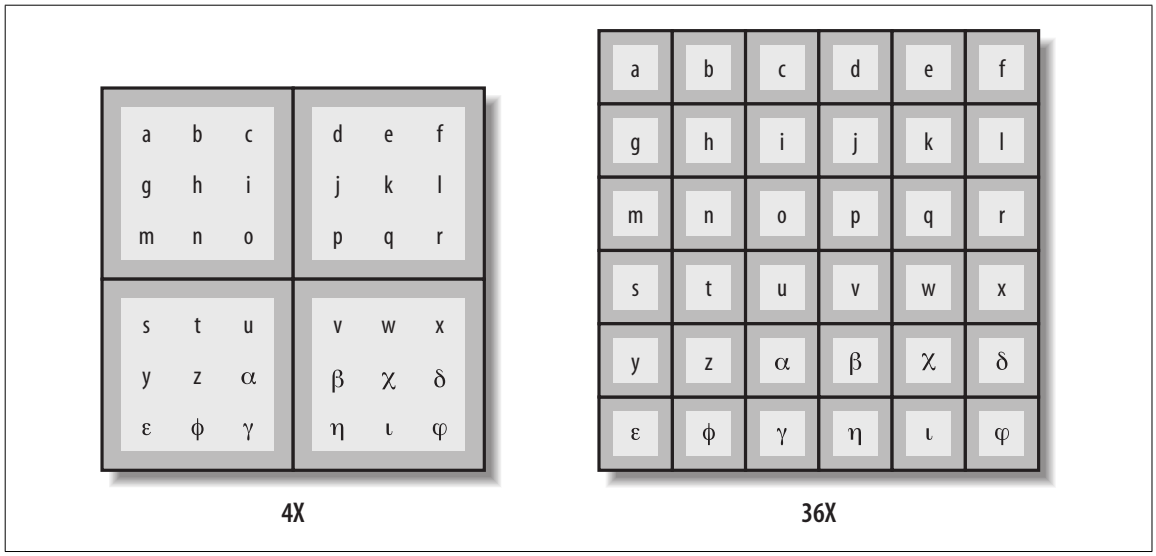


Figure 3-1. Packaging versus grain size, same workload

The total work to be done on the system is represented by the light and dark gray regions combined—the overall box. The 36X case shows how too small a grain size leads to a relatively high proportion of overhead. The 4X case shows how a large grain size reduces this proportion, at the cost of reducing potential parallelism. The

overhead as a fraction of useful work depends on the grain size, not on the number of grains. Consider this relationship and not the total number of iterations or number of processors when setting a grain size.

A recommended rule of thumb is that `grainsize` iterations of `operator()` should take at least 10,000 to 100,000 instructions to execute, which typically means more than a few thousand mathematic calculations. When in doubt, do the following:

1. Set the `grainsize` parameter higher than necessary. Setting it to 10,000 is usually a good starting point.
2. Run your algorithm on one processor core.
3. Start halving the `grainsize` parameter and see how much the algorithm slows down as the value decreases.

A slowdown of about 5 to 10 percent when running with a single thread is a good setting for most purposes. The drawback of setting a grain size too high is that it can reduce parallelism. For example, if your `grainsize` value is 10,000 and the loop has 20,000 iterations, the `parallel_for` distributes the loop across only two processors, even if more are available. However, if you are unsure, err on the side of being a little too high instead of a little too low because too low a value hurts serial performance, which in turns hurts parallel performance if other parallelism is available higher in your program.

Grain size is not an exact science; you do not have to set it very precisely.

To illustrate the inexact nature of setting the best value, Figure 3-2 shows a typical "bathtub curve" for execution time versus grain size, based on the floating-point `a[i]=b[i]*c` computation over 1 million indices. There is very little work per iteration.

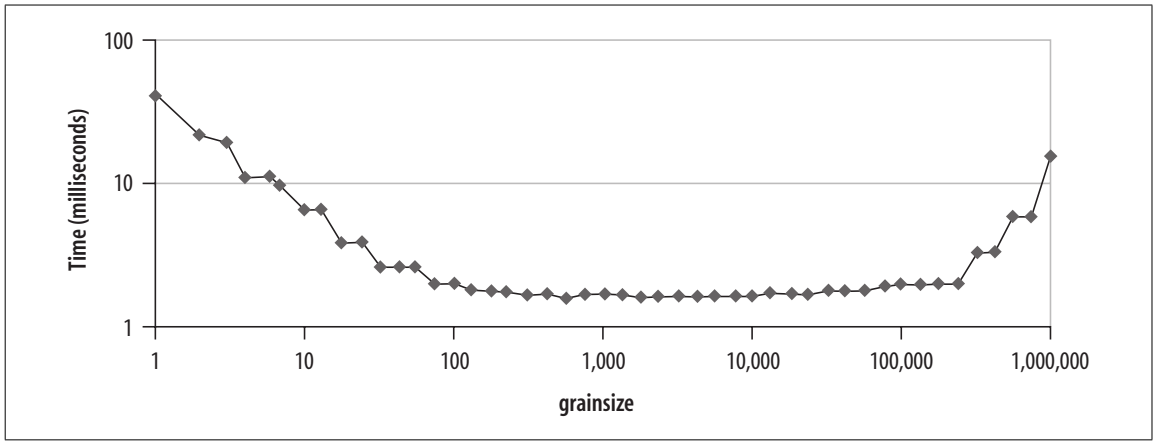
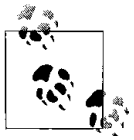


Figure 3-2. Wall clock time versus grainsize

The scale is logarithmic. The downward slope on the left side indicates that with a grain size of 1, most of the time is spent on packaging (dark gray in Figure 3-1). An increase in grain size brings a proportional decrease in parallel overhead. Then the curve flattens out because the packaging (overhead) becomes insignificant for a sufficiently large grain size. At the extreme right, the curve turns up because the chunks are so large that there are fewer chunks than available hardware threads. Notice that any grain size over the wide range of 100 to 100,000 works quite well.



A general rule of thumb for parallelizing loop nests is to parallelize the outermost one possible. The reason is that each iteration of an outer loop is likely to provide a bigger grain of work than an iteration of an inner loop.

For a sufficiently simple function Foo, the examples might not show significant speedup when written as parallel loops. The cause could be insufficient system bandwidth between the processors and memory. In that case, you may have to rethink your algorithm to take better advantage of cache. Restructuring to better utilize the cache usually benefits the parallel program as well as the serial program.

Automatic grain size

The parallel loop templates in the original release of Threading Building Blocks required a grainsize parameter. We have been looking into ways to automatically determine the right value, but it's not easy.

Feedback from users is that they want automatic grain size determination, even if it is not always optimal, so the grainsize parameter is now optional in creating the iterator. When grainsize is not specified, a *partitioner* should be supplied to the algorithm template.

If both the partitioner and the grainsize are omitted, it's the same as specifying a grainsize of 1. If there are more than 10,000 instructions per iteration, it will work okay. With fewer than a thousand or so, there will be a serious performance hit.

A partitioner is an object that guides the chunking of a range. Currently, only auto_partitioner makes sense without a grainsize.

The auto_partitioner provides an alternative that heuristically chooses the grain size so that you do not have to specify one. The heuristic attempts to limit overhead while still providing ample opportunities for load balancing. Guessing the grain size with the heuristic is not easy, but it does have a connection with the task scheduler that allows it to get dynamic guidance, which can make it better than a static choice of grain size.

Example 3-7 shows how to use an auto_partitioner instead of a grainsize. Notice that the grainsize parameter is omitted when constructing the blocked_range and that an auto_partitioner object is passed as a third argument to the parallel_for.

Example 3-7. Use of auto_partitioner

```
#include "tbb/parallel_for.h"

void ParallelApplyFoo( float a[], size_t n ) {
    parallel_for(blocked_range<size_t>(0,n), ApplyFoo(a),
                 auto_partitioner( ) );
}
```

As with most heuristics, there are situations in which `auto_partitioner` might not guess optimally and `simple_partitioner` would yield better performance. We recommend using `auto_partitioner` unless you have the time to experiment and tune the grain size for machines of interest.

Support for partitioners is a new feature in Threading Building Blocks and will almost certainly have some additions in the future. You should check the documentation with the latest release to see whether there are new features.

Notes on automatic grain size

The optimal grain size depends upon implementation. This issue is not limited to parallelism. Try writing a large file using a single-character write-and-flush operation on each character. Picking appropriate chunk sizes is common in programming. Thus, automatic determination of an optimal grain size is still a research problem.

parallel_for with partitioner

`parallel_for` takes an optional third argument to specify a partitioner. See the earlier section "Automatic grain size" for more information.

This example shows a simple use of the partitioner concept with a `parallel_for`. The code shown in Example 3-8 is an extension of Example 3-6. An `auto_partitioner` is used to guide the splitting of the range.

Example 3-8. Parallel average with partitioner

```
#include "tbb/parallel_for.h"
#include "tbb/blocked_range.h"

using namespace tbb;

struct Average {
    float* input;
    float* output;
    void operator( )( const blocked_range<int>& range ) const {
        for( int i=range.begin(); i!=range.end( ); ++i )
            output[i] = (input[i-1]+input[i]+input[i+1])*(1/3.0f);
    }
};
```

Example 3-8. Parallel average with partitioner (continued)

```
// Note: The input must be padded such that input[-1] and input[n]
// can be used to calculate the first and last output values.
void ParallelAverage( float* output, float* input, size_t n ) {
    Average avg;
    avg.input = input;
    avg.output = output;
    parallel_for( blocked_range<int>( 0, n ), avg, auto_partitioner( ) );
```

Two important changes from Example 3-6 should be noted:

- The call to `parallel_for` takes a third argument, an `auto_partitioner` object.
- The `blocked_range` constructor is not provided with a `grainsize` parameter.

parallel_reduce

Applying a function such as `sum`, `max`, `min`, or logical `AND` across all the members of a group is called a *reduction* operation. Doing a reduction in parallel can yield a different answer from a serial reduction because of rounding. For instance, `A+B+C+D+E+F` may be evaluated in serial as `(((((A+B)+C)+D)+E)+F)`, whereas the parallel version may compute `((A+B)+((C+D)+(E+F)))`. Ideally, the results would be the same, but if rounding can occur, the answers will differ. Traditional C++ programs perform reductions in loops, as in the summation shown in Example 3-9.

Example 3-9. Original reduction code

```
float SerialSumFoo( float a[], size_t n ) {
    float sum = 0;
    for( size_t i=0; i!=n; ++i )
        sum += Foo(a[i]);
    return sum;
}
```

If the iterations are independent, you can parallelize this loop using the template class `parallel_reduce`, as shown in Example 3-10.

Example 3-10. A class for use by a parallel_reduce

```
class SumFoo {
    float* my_a;
public:
    float sum;
    void operator( )( const blocked_range<size_t>& r ) {
        float *a = my_a;
        for( size_t i=r.begin(); i!=r.end( ); ++i )
            sum += Foo(a[i]);
    }

    SumFoo( SumFoo& x, split ) : my_a(x.my_a), sum(0) {}

    void join( const SumFoo& y ) {sum+=y.sum;}
```

Example 3-10. A class for use by a parallel_reduce (continued)

```
    SumFoo(float a[] ) :
        my_a(a), sum(0)
    {}
};
```

Threading Building Blocks defines `parallel_reduce` similar to `parallel_for`. The principle difference is that thread-private copies of the *body* must be merged at the end, and therefore the `operator()` is *not* `const`. Note the differences with class `ApplyFoo` from Example 3-4. The `operator()` is not `const` because it must update `SumFoo::sum`. Another difference is that `SumFoo` has a *splitting constructor* and a method named `join` that must be present for `parallel_reduce` to work.

The splitting constructor takes as arguments a reference to the original object, and has a dummy argument of type `split`. This dummy argument serves, simply by its presence, to distinguish the splitting constructor from a copy constructor (which would not have this argument). More information appears later in this chapter in a section titled "Advanced Topic: Other Kinds of Iteration Spaces."

The `join` method is invoked whenever a task finishes its work and needs to merge the result back with the main body of work. The parameter passed to the method is the result of the work, so the method will just repeat the same operation that was performed in each task on each element (in this case, a sum).

When a worker thread is available, as decided by the task scheduler, `parallel_reduce` hands off work to it by invoking the splitting constructor to create a subtask for the processor. When the task completes, `parallel_reduce` uses the `join` method to accumulate the result of the subtask. The diagram in Figure 3-3 shows the split-join sequence.

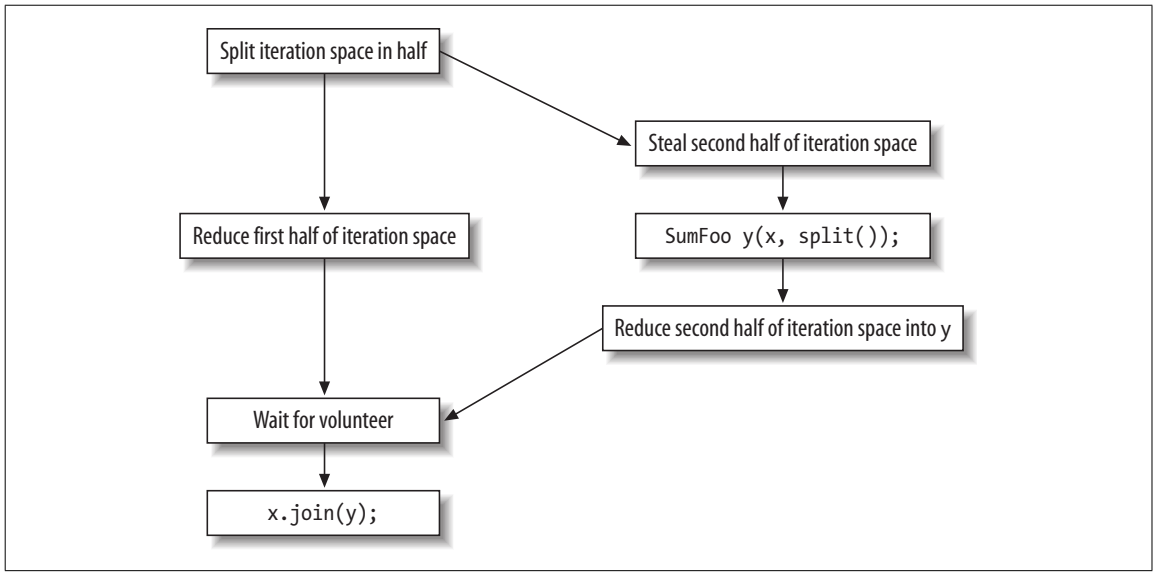


Figure 3-3. Split-join sequence

A line in Figure 3-3 indicates order in time. Notice that the splitting constructor might run concurrently while object `x` is being used for the first half of the reduction. Therefore, all actions of the splitting constructor that create `y` must be made thread-safe with respect to `x`. If the splitting constructor needs to increment a reference count shared with other objects, it should use an atomic increment (described in Chapter 8).

Define `join` to update `this` to represent the accumulated result for `this` and the right-hand side. The reduction operation should be associative, but it does not have to be commutative. For a noncommutative operation `op`, `left.join(right)` should update `left` to be the result of `left op right`.

A body is split only if the range is split, but the converse is not necessarily true. Figure 3-4 diagrams a sample execution of `parallel_reduce`. The root represents the original body b_0 being applied to the half-open interval `[0,20)`. The range is recursively split at each level into two subranges. The grain size for the example is 5, which yields four leaf ranges. The slash marks (`/`) denote where copies (b_1 and b_2) of the body were created by the body splitting constructor. Bodies b_0 and b_1 each evaluate one leaf. Body b_2 evaluates leaf `[10,15)` and leaf `[15,20)`, in that order. On the way back up the tree, `parallel_reduce` invokes b_0`.join(`b_1`)` and b_0`.join(`b_2`)` to merge the results of the leaves.

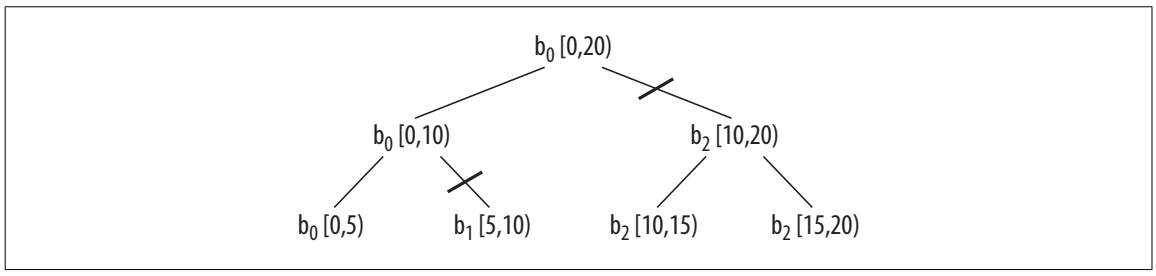


Figure 3-4. parallel_reduce over blocked_range<int>(0,20,5)

Figure 3-4 shows only one of the possible executions. Other valid executions include splitting b_2 into b_2 and b_3, or doing no splitting at all. With no splitting, b_0 evaluates each leaf in left to right order, with no calls to `join`.

A given body always evaluates one or more consecutive subranges in left to right order. For example, in Figure 3-4, body b_2 is guaranteed to evaluate `[10,15)` before `[15,20)`. You may rely on the consecutive left to right property for a given instance of a body, but you must not rely on a particular choice of body splitting. `parallel_reduce` makes the choice of body splitting nondeterministically. The left-to-right property allows commutative operations to work, such as finding the first minimum number in a sequence along with its position, as shown later in Example 3-13.

When no worker threads are available, `parallel_reduce` executes sequentially from left to right in the same sense as for `parallel_for`. Sequential execution never invokes the splitting constructor or method `join`.

Example 3-11 uses the class defined in Example 3-10 to perform the reduction.

Example 3-11. Parallel reduction code

```
float ParallelSumFoo( const float a[], size_t n ) {
    SumFoo sf(a);
    parallel_reduce(blocked_range<size_t>(0,n,YouPickAGrainSize), sf );
    return sf.sum;
}
```

As with `parallel_for`, you must provide a reasonable grain size, with enough iterations to take at least 10,000 instructions. If you are not sure, it is best to err on the side of too large a grain size. You can also use a partitioner object to allow the runtime library to guide the chunking of the range.

`parallel_reduce` generalizes to any associative operation. In general, the splitting constructor does two things:

- Copies read-only information necessary to run the loop body
- Initializes the reduction variables to the identity element of the operations

The `join` method should do the corresponding merges. You can do more than one reduction at the same time: for instance, you can gather the `min` and `max` with a single `parallel_reduce`.

Advanced example

An example of a more advanced associative operation is to find the index containing the smallest element of an array. A serial version might look like Example 3-12.

Example 3-12. Original minimization code

```
long SerialMinIndexFoo( const float a[], size_t n ) {
    float value_of_min = FLT_MAX;        // FLT_MAX from <climits>
    long index_of_min = -1;
    for( size_t i=0; i<n; ++i ) {
        float value = Foo(a[i]);
        if( value<value_of_min ) {
              value_of_min = value;
            index_of_min = i;
        }
    }
    return index_of_min;
}
```

The loop works by keeping track of the minimum value found so far, and the index of this value. This is the only information carried between loop iterations. To convert the loop to use `parallel_reduce`, the function object must keep track of the information carried, and must be able to merge this information when iterations are spread across multiple threads. Also, the function object must record a pointer to `a` to provide context.

Example 3-13 shows the complete function object.

Example 3-13. Function object for minimization

```
class MinIndexFoo {
    const float *const my_a;
public:
    float value_of_min;
    long index_of_min;
    void operator( )( const blocked_range<size_t>& r ) {
        const float *a = my_a;
        for( size_t i=r.begin(); i!=r.end( ); ++i ) {
        float value = Foo(a[i]);
        if( value<value_of_min ) {
            value_of_min = value;
              index_of_min = i;
        }
        }
    }

    MinIndexFoo( MinIndexFoo& x, split ) :
        my_a(x.my_a),
        value_of_min(FLT_MAX),     // FLT_MAX from <climits>
        index_of_min(-1)
   {}

    void join( const SumFoo& y ) {
        if( y.value_of_min<value_of_min ) {
            value_of_min = y.value_of_min;
            index_of_min = y.index_of_min;
        }
    }

    MinIndexFoo( const float a[] ) :
        my_a(a),
        value_of_min(FLT_MAX),     // FLT_MAX from <climits>
        index_of_min(-1),
    {}
};
```

Now `SerialMinIndex` can be rewritten using `parallel_reduce` (see Example 3-14).

Example 3-14. Parallel minimization

```
long ParallelMinIndexFoo( float a[], size_t n ) {
    MinIndexFoo mif(a);
    parallel_reduce(blocked_range<size_t>(0,n,YouPickAGrainSize), mif );
    return mif.index_of_min;
}
```

Chapter 11 contains a prime number finder based on `parallel_reduce`.

Parallel_reduce with partitioner

`Parallel_reduce` has an optional third argument to specify a partitioner. See the section "Automatic grain size" for more information.

Example 3-15 extends Examples 3-10 and 3-11 by using an `auto_partitioner`.

Example 3-15. Parallel sum with partitioner

```
#include "tbb/parallel_reduce.h"
#include "tbb/blocked_range.h"

using namespace tbb;

struct Sum {
    float value;
    Sum( ) : value(0) {}
    Sum( Sum& s, split ) {value = 0;}
    void operator( )( const blocked_range<float*>& range ) {
        float temp = value;
        for( float* a=range.begin(); a!=range.end( ); ++a ) {
            temp += *a;
        }
        value = temp;
    }
    void join( Sum& rhs ) {value += rhs.value;}
};

float ParallelSum( float array[], size_t n ) {
    Sum total;
    parallel_reduce( blocked_range<float*>( array, array+n ),
                     total, auto_partitioner( ) );
    return total.value;
}
```

Two important changes from Example 3-11 should be noted:

- The call to `parallel_reduce` takes a third argument, an `auto_partitioner` object.
- The `blocked_range` constructor is not provided with a `grainsize` parameter.

Advanced Topic: Other Kinds of Iteration Spaces

The examples so far have used the class `blocked_range<T>` to specify ranges. This class is useful in many situations, but it does not fit every one. You can define your own iteration space objects to use with Intel Threading Building Blocks. The object must specify how it can be split into subspaces by providing two methods and a *splitting constructor*. You can see these simple definitions in the class `blocked_range<T>`.

If your class is called `R`, the methods and constructor could be as shown in Example 3-16.

Example 3-16. Define your own iteration space object

```
class R {
    // True if range is empty
    bool empty( ) const;
    // True if range can be split into nonempty subranges
    bool is_divisible( ) const;
    // Split r into subranges r and *this
    R( R& r, split );
    ...
};
```

The method `empty` must return `true` if the range is empty. The method `is_divisible` needs to return `true` if the range can be split into two nonempty subspaces, and such a split is worth the overhead. The splitting constructor needs to take two arguments:

- The first of type `R`
- The second of type `tbb::split`

The second argument is not used; it serves only to distinguish the constructor from an ordinary copy constructor.

The splitting constructor should attempt to split `r` into two halves of roughly the same size, update `r` to be the first half, and let the constructed object be the second half. The two halves should be nonempty. The parallel algorithm templates call the splitting constructor on `r` only if `r.is_divisible` is `true`.

The code in Example 3-17 defines a type `TrivialIntegerRange` that models the Range Concept. It represents a half-open interval `[lower,upper)` that is divisible down to a single integer.

Example 3-17. Trivial integer range

```
struct TrivialIntegerRange {
    int lower;
    int upper;
    bool empty( ) const {return lower==upper;}
    bool is_divisible( ) const {return upper>lower+1;}
    TrivialIntegerRange( TrivialIntegerRange& r, split ) {
        int m = (r.lower+r.upper)/2;
        lower = m;
        upper = r.upper;
        r.upper = m;
    }
};
```

`TrivialIntegerRange` is for demonstration and is not very practical because it lacks a `grainsize` parameter. Use the library class `blocked_range` instead. Example 3-18 shows the implementation of `blocked_range` in the Threading Building Blocks library. A full discussion of the class is beyond the scope of this book, but the code serves to show the relationships among the various methods of an iteration space and the use of the grain size.

Example 3-18. Implementation of blocked_range

```
class blocked_range {
public:
  typedef Value const_iterator;
  typedef size_t size_type;

  blocked_range() : my_begin(), my_end( ) {}

  blocked_range( Value begin, Value end, size_type grainsize ) :
    my_end(end), my_begin(begin), my_grainsize(grainsize)
  {
    __TBB_ASSERT( my_grainsize>0, "grainsize must be positive" );
  }

  const_iterator begin( ) const {return my_begin;}
  const_iterator end( ) const {return my_end;}

  size_type size( ) const {
    __TBB_ASSERT( !(end()<begin()), "size() unspecified if end()<begin( )" );
    return size_type(my_end-my_begin);
  }

  size_type grainsize( ) const {return my_grainsize;}

  //------------------------------------------------------------------------
  // Methods that implement Range Concept
  //------------------------------------------------------------------------
  bool empty( ) const {return !(my_begin<my_end);}
  bool is_divisible() const {return my_grainsize<size( );}

  blocked_range( blocked_range& r, split ) :
    my_end(r.my_end),
    my_begin(do_split(r)),
    my_grainsize(r.my_grainsize)
  {}

private:
  Value my_end;
  Value my_begin;
  size_type my_grainsize;

  static Value do_split( blocked_range& r ) {
    __TBB_ASSERT( r.is_divisible( ), "cannot split blocked_range that is not divisible" );
    Value middle = r.my_begin + (r.my_end-r.my_begin)/2u;
    r.my_end = middle;
    return middle;
  }

  template<typename RowValue, typename ColValue>
  friend class blocked_range2d;
};
```

The iteration space does not have to be linear. Look at Example 3-19 (code from the header file *tbb/blocked_range2d.h*) for an example of a two-dimensional range. Its splitting constructor attempts to split the range along its longest axis. When used with `parallel_for`, it causes the loop to be *recursively blocked* in a way that improves cache usage. This nice cache behavior means that using `parallel_for` over a `blocked_range2d<T>` can make a loop run faster than the sequential equivalent, even on a single processor.

Example 3-19. A two-dimensional range

```
class blocked_range2d {
public:
    typedef blocked_range<RowValue> row_range_type;
    typedef blocked_range<ColValue> col_range_type;

private:
    row_range_type my_rows;
    col_range_type my_cols;

public:

    blocked_range2d( RowValue row_begin, RowValue row_end, typename row_range_type::size_
type row_grainsize,
                     ColValue col_begin, ColValue col_end, typename col_range_type::size_
type col_grainsize ) :
        my_rows(row_begin,row_end,row_grainsize),
        my_cols(col_begin,col_end,col_grainsize)
    {
    }

    bool empty( ) const {
        // Yes, it is a logical OR here, not AND.
        return my_rows.empty() || my_cols.empty( );
    }

    bool is_divisible( ) const {
        return my_rows.is_divisible() || my_cols.is_divisible( );
    }

    blocked_range2d( blocked_range2d& r, split ) :
        my_rows(r.my_rows),
        my_cols(r.my_cols)
    {
        if( my_rows.size()*double(my_cols.grainsize()) < my_cols.size( )*double(my_rows.
grainsize( )) ) {
            my_cols.my_begin = col_range_type::do_split(r.my_cols);
        } else {
            my_rows.my_begin = row_range_type::do_split(r.my_rows);
        }
    }
```

Example 3-19. A two-dimensional range (continued)

```
    const row_range_type& rows( ) const {return my_rows;}

    const col_range_type& cols( ) const {return my_cols;}
};
```

Notes on blocked_range2d

The template class `blocked_range2d` is included in Threading Building Blocks because beneficial uses for it showed up in many common applications. Parallelizing over two dimensions instead of one often yields more parallelism and better cache behavior than parallelizing over only one dimension.

The idea of extending to three or more dimensions, and the idea of making the number of dimensions a parameter, were both considered but rejected because they added too much complexity with few practically motivating cases.

A constructor for `blocked_range2d` that takes two `blocked_range` arguments was considered as an alternative to the six-argument constructor, but so far, practice has shown that such a constructor just adds extra clutter.

parallel_scan

A `parallel_scan` computes a parallel prefix, also known as a parallel scan. This computation is an advanced concept in parallel computing that is sometimes useful in scenarios that appear to have inherently serial dependencies.

A mathematical definition of the parallel prefix is as follows. Let $\oplus$ be an associative operation $\oplus$ with left-identity element $\mathtt{id}_{\oplus}$. The parallel prefix of $\oplus$ over a sequence $x_0, x_1, \ldots x_{n-1}$ is a sequence $y_0, y_1, y_2, \ldots y_{n-1}$ where:

- $y_0 = id_{\oplus} \oplus x_0$
- $y_i = y_{i-1} \oplus i$

For example, if $\oplus$ is addition, the parallel prefix corresponds to a running sum and the identity element is 0. A serial implementation of parallel prefix is:

```
T temp = id⊕;
for( int i=1; i<=n; ++i ) {
    temp = temp ⊕ x[i];
    y[i] = temp;
}
```

Parallel prefix performs this in parallel by reassociating the application of $\oplus$ and using two passes. It may invoke $\oplus$ up to twice as many times as the serial prefix algorithm. But given the right grain size and sufficient hardware threads, it can outperform the serial prefix because—even though it does more work—it can distribute the work across more than one hardware thread.

Because parallel_scan needs two passes, systems with only two hardware threads tend to exhibit only a small speedup. parallel_scan is better suited for future systems with more than two cores. It shows how a problem that appears inherently sequential can be parallelized.

Example 3-20 demonstrates how to use parallel_scan in a way similar to the sequential example.

Example 3-20. parallel_scan

```
using namespace tbb;

class Body {
    T reduced_result;
    T* const y;
    const T* const x;
public:
    Body( T y_[], const T x_[] ) : reduced_result(0), x(x_), y(y_) {}
    T get_reduced_result( ) const {return reduced_result;}

    template<typename Tag>
    void operator( )( const blocked_range<int>& r, Tag ) {
        T temp = reduced_result;
        for( int i=r.begin(); i<r.end( ); ++i ) {
            temp = temp ⊕ x[i];
            if( Tag::is_final_scan( ) )
                y[i] = temp;
        }
        reduced_result = temp;
    }
    Body( Body& b, split ) : x(b.x), y(b.y), reduced_result(id⊕) {}
    void reverse_join( Body& a ) {
      reduced_result = a.reduced_result ⊕ reduced_result;
    }
    void assign( Body& b ) {reduced_result = b.reduced_result;}
};

float DoParallelScan( T y[], const T x[], int n) {
    Body body(y,x);
    parallel_scan( blocked_range<int>(0,n,1000), body );
    return body.get_reduced_result( );
}
```

The definition of operator( ) demonstrates typical patterns when using parallel_scan:

- A single template defines both the first pass and the second pass versions. Doing so is not required, but it usually saves coding effort because the two versions are usually similar. The library defines static method is_final_scan( ) to enable differentiation among the versions.

- The prescan variant computes the ⊕ reduction, but does not update y. The prescan is used by parallel_scan to generate look-ahead partial reductions.
- The final scan variant computes the ⊕ reduction and updates y.

The operation reverse_join is similar to the operation join used by parallel_reduce, except that the arguments are reversed. In other words, this is the *right* argument of ⊕.

The template function parallel_scan decides whether and when to generate parallel work. It is thus crucial that ⊕ is associative and that the methods of Body faithfully represent it. Operations such as floating-point addition that are somewhat associative can be used, with the understanding that the results may be rounded differently depending upon the association used by parallel_scan. The reassociation may differ between runs, even on the same machine. However, if no worker threads are available, execution associates identically to the serial form shown at the beginning of this section.

Parallel_scan with partitioner

Parallel_scan has an optional third argument to specify a partitioner (Example 3-21). See the section "Automatic grain size" for more information.

Example 3-21. parallel_scan with partitioner argument

```
using namespace tbb;

class Body {
    T sum;
    T* const y;
    const T* const x;
public:
    Body( T y_[], const T x_[] ) : sum(0), x(x_), y(y_) {}
    T get_sum( ) const {return sum;}

    template<typename Tag>
    void operator( )( const blocked_range<int>& r, Tag ) {
        T temp = sum;
        for( int i=r.begin(); i<r.end( ); ++i ) {
            temp = temp ⊕ x[i];
            if( Tag::is_final_scan( ) )
                y[i] = temp;
        }
        sum = temp;
    }
    Body( Body& b, split ) : x(b.x), y(b.y), sum(id⊕) {}
    void reverse_join( Body& a ) { sum = a.sum ⊕ sum;}
    void assign( Body& b ) {sum = b.sum;}
};

float DoParallelScan( T y[], const T x[], int n) {
    Body body(y,x);
    parallel_scan( blocked_range<int>(0,n), body, auto_partitioner( ) );
    return body.get_sum( );
}
```

Two important changes from Example 3-20 should be noted:

- The call to `parallel_scan` takes a third argument, an `auto_partitioner` object.
- The `blocked_range` constructor is not provided with a `grainsize` parameter.

Recursive Range Specifications

Most algorithms provided by the Threading Building Blocks library are generic and operate on all types that model the necessary concepts. Recursive ranges define the space for the algorithm to operate upon and therefore are important to understand.

Splittable Concept

Requirements for a type whose instances can be split into two pieces. Table 3-1 lists the requirements for a splittable type `X` with instance `x`.

Table 3-1. Splittable Concept

Pseudosignature	Semantics
`X::X(X& x, split)`	Split `x` into two parts, one reassigned to `x` and the other to the newly constructed object.

Description

A type is splittable if it has a *splitting constructor* that allows an instance to be split into two pieces. The splitting constructor takes as arguments a reference to the original object, and a dummy argument of type `split`, which is defined by the library. The dummy argument distinguishes the splitting constructor from a copy constructor. After the constructor runs, `x` and the newly constructed object should represent the two pieces of the original `x`. The library uses splitting constructors in two contexts:

- *Partitioning* a range into two subranges that can be processed concurrently
- *Forking* a body (function object) into two bodies that can run concurrently

The following model types provide examples.

Model Types: Splittable Ranges

`blocked_range` and `blocked_range2d` represent splittable ranges. For each of these, splitting partitions the range into two subranges.

The bodies for `parallel_reduce`, `parallel_scan`, `simple_partitioner`, and `auto_partitioner` must be splittable. For each of these, splitting results in two bodies that can run concurrently.

split Class

Type for dummy argument of a splitting constructor.

```
#include "tbb/tbb_stddef.h"

class split;
```

Description

An argument of type `split` is used to distinguish a splitting constructor from a copy constructor.

Members

```
namespace tbb {
    class split {
    };
}
```

Range Concept

Requirements for a type representing a recursively divisible set of values. Table 3-2 lists the requirements for a `Range` type `R`.

Table 3-2. Range Concept

Pseudosignature	Semantics
`R::R( const R& )`	Copy constructor
`R::~R()`	Destructor
`bool R::empty() const`	`True` if range is empty
`bool R::is_divisible() const`	`True` if range can be partitioned into two subranges
`R::R( R& r, split )`	Split `r` into two subranges

Description

A `Range` can be recursively subdivided into two parts. It is recommended that the division be into nearly equal parts, but it is not required. Splitting as evenly as possible typically yields the best parallelism. Ideally, a range is recursively splittable *until the parts represent portions of work that are more efficient to execute serially rather than split further*. This key limit to splitting is called the grain size. The amount of work represented by a `Range` typically depends upon higher-level context; hence, a typical type that models a `Range` should provide a way to control the degree of splitting. For example, the template class `blocked_range` has a `grainsize` parameter that specifies the biggest range considered indivisible.

The constructor that implements splitting is called a *splitting constructor*. If the set of values has a sense of direction, by convention the splitting constructor should construct the second part of the range and update the argument to refer to the first part of the range. Following this convention causes the `parallel_for`, `parallel_reduce`, and `parallel_scan` algorithms, when running sequentially, to work across a range in the increasing order typical of an ordinary sequential loop.

Model Types

`blocked_range` models a one-dimensional range.

`blocked_range2d` models a two-dimensional range.

blocked_range<Value> Template Class

Template class for a recursively divisible half-open interval.

```
#include "tbb/blocked_range.h"
template<typename Value> class blocked_range;
```

Description

A `blocked_range<Value>` represents a half-open range `[i,j)` that can be recursively split. The types `i` and `j` must model the requirements in Table 3-3. Because the requirements are pseudosignatures, signatures that differ in ways that can be implicitly converted are allowed. For example, a `blocked_range<int>` is allowed because the difference of two `int` values can be implicitly converted to a `size_t`. Examples that model the `Value` requirements are integral types, pointers, and STL random-access iterators whose difference can be implicitly converted to a `size_t`.

A `blocked_range` models the Range Concept.

Table 3-3. Value Concept for block_range

Pseudosignature	Semantics
`Value::Value( const Value& )`	Copy constructor
`Value::~Value( )`	Destructor
`bool operator<( const Value& i, const Value& j )`	Value `i` precedes value `j`
`size_t operator-( const Value& i, const Value& j )`	Number of values in range `[i,j)`
`Value operator+( const Value& i, size_t k )`	*k*th value after `i`

A `blocked_range<Value>` specifies a `grainsize` of type `size_t`. A `blocked_range` is splittable into two subranges if the size of the range exceeds `grainsize`. The ideal grain size depends upon the context of the `blocked_range<Value>`, which is typically the range argument to the loop templates `parallel_for`, `parallel_reduce`, and `parallel_scan`. Too small a grain size may cause scheduling overhead within the loop templates to swamp speedup gained from parallelism. Too large a grain size may unnecessarily limit parallelism. For example, if the grain size is so large that the range can be split only once, the maximum possible parallelism is two.

For a blocked_range [i,j) where j<i, not all methods have specified behavior. However, enough methods do have specified behavior that parallel_for, parallel_reduce, and parallel_scan iterate over the same iteration space as the serial loop for(Value index=i; index<j; ++index)..., even when j<i. If the debug macro TBB_DO_ASSERT is nonzero, methods with unspecified behavior raise an assertion failure. You should *not* use iteration spaces [i,j) with j<i.

Members

```
namespace tbb {
    template<typename Value>
    class blocked_range {
    public:
        // types
        typedef size_t size_type;
        typedef Value const_iterator;

        // constructors
        blocked_range( Value begin, Value end, size_type grainsize=1);
        blocked_range( blocked_range& r, split );

        // capacity
        size_type size( ) const;
        bool empty( ) const;

        // access
        size_type grainsize( ) const;
        bool is_divisible( ) const;

        // iterators
        const_iterator begin( ) const;
        const_iterator end( ) const;
    };
}
```

size_type
: Description: the type for measuring the size of a blocked_range. The type is always a size_t.

const_iterator
: Description: the type of a value in the range. Despite its name, the type const_iterator is not necessarily an STL iterator; it merely needs to meet the Value requirements in Table 3-3. However, it is convenient to call it const_iterator so that if it is a const_iterator, the blocked_range behaves like a read-only STL container.

blocked_range(Value begin, Value end, size_t grainsize=1)
: Effects: constructs a blocked_range representing the half-open interval [begin,end) with the given grainsize.

  Example: the statement blocked_range<int> r(5, 14, 2); constructs a range of int that contains the values 5 through 13 inclusive, with a grainsize of 2. (The begin parameter 5 is taken to be inclusive, and the end parameter 14 to be exclusive.) Afterward, r.begin( )==5 and r.end( )==14.

`blocked_range( blocked_range& range, split )`

Requirements: `is_divisible( )` is `true`.

Effects: partitions range into two subranges. The newly constructed `blocked_range` is approximately the second half of the original range, and `range` is updated to be the remainder. Each subrange has the same `grainsize` as the original range.

Example: let `i` and `j` be integers that define a half-open interval `[i,j)` and let `g` specify a `grainsize`. The statement `blocked_range<int> r(i,j,g)` constructs a `blocked_range<int>` that represents `[i,j)` with `grainsize g`. Running the statement `blocked_range<int> s(r,split);` subsequently causes `r` to represent `[i, i +(j -i)/2)` and `s` to represent `[i +(j -i)/2, j)`, both with `grainsize g`.

`size_type size( ) const`

Requirements: `end()<begin( )` is `false`.

Effects: determines size of range.

Returns: `end()-begin( )`.

`bool empty( ) const`

Effects: determines whether the range is empty.

Returns: `!(begin()<end( ))`.

`size_type grainsize( ) const`

Returns: grain size of range.

`bool is_divisible( ) const`

Requirements: `!(end()<begin( ))`.

Effects: determines whether the range can be split into subranges.

Returns: `true` if `size()>grainsize( )`; `false` otherwise.

`const_iterator begin( ) const`

Returns: inclusive lower bound on the range.

`const_iterator end( ) const`

Returns: exclusive upper bound on the range.

blocked_range2d Template Class

Template class that represents a recursively divisible, two-dimensional, half-open interval.

```
#include "tbb/blocked_range2d.h"

template<typename RowValue, typename ColValue>

  class blocked_range2d;
```

Description

A `blocked_range2d<RowValue,ColValue>` represents a half-open, two-dimensional range `[i0,j0)x[i1,j1)`. Each axis of the range has its own splitting threshold. The `RowValue` and `ColValue` must meet the requirements in Table 3-3. A `blocked_range` is splittable if either axis is splittable. A `blocked_range` models the Range Concept.

Members

```
namespace tbb {
template<typename RowValue, typename ColValue=RowValue>
    class blocked_range2d {
    public:
        // Types
        typedef blocked_range<RowValue> row_range_type;
        typedef blocked_range<ColValue> col_range_type;

        // Constructors
        blocked_range2d( RowValue row_begin, RowValue row_end,
                         typename row_range_type::size_type row_grainsize,
                         ColValue col_begin, ColValue col_end,
                         typename col_range_type::size_type col_grainsize);
        blocked_range2d( blocked_range2d& r, split );

        // Capacity
        bool empty( ) const;

        // Access
        bool is_divisible( ) const;
        const row_range_type& rows( ) const;
        const col_range_type& cols( ) const;
    };
}
```

`row_range_type`

Description: a `blocked_range<RowValue>`. That is, the type of the row values.

`col_range_type`

Description: a `blocked_range<ColValue>`. That is, the type of the column values.

`blocked_range2d<RowValue,ColValue>( RowValue row_begin, RowValue row_end, typename row_range_type::size_type row_grainsize, ColValue col_begin, ColValue col_end, typename col_range_type::size_type col_grainsize )`

Effects: constructs a `blocked_range2d` representing a two-dimensional space of values. The space is the half-open Cartesian product `[row_begin,row_end)x[col_begin,col_end)`, with the given grain sizes for the rows and columns.

Example: the statement `blocked_range2d<char,int> r('a', 'z'+1, 3, 0, 10, 2 );` constructs a two-dimensional space that contains all value pairs of the form `(i, j)`, where `i` ranges from `'a'` to `'z'` with a grain size of `3`, and `j` ranges from `0` to `9` with a grain size of `2`.

`blocked_range2d<RowValue,ColValue> ( blocked_range2d& range, split )`

Effects: partitions a range into two subranges. The newly constructed `blocked_range2d` is approximately the second half of the original range, and `range` is updated to be the remainder. Each subrange has the same grain size as the original range. The split is either by rows or by columns. The choice of which axis to split is intended to cause, after repeated splitting, the subranges to approach the aspect ratio of the respective row and column grain sizes. For example, if the `row_grainsize` is twice the `col_grainsize`, the subranges will tend toward having twice as many rows as columns.

`bool empty( ) const`
: Effects: determines whether the range is empty.

Returns: `rows().empty()||cols().empty( )`.

`bool is_divisible( ) const`
: Effects: determines whether the range can be split into subranges.

Returns: `rows().is_divisible()||cols().is_divisible( )`.

`const row_range_type& rows( ) const`
: Returns: range containing the rows of the value space.

`const col_range_type& cols( ) const`
: Returns: range containing the columns of the value space.

Partitioner Concept

Requirements for a type that decides whether a range should be operated on by a task body or further split. Table 3-4 lists the requirements for a partitioner type `P`.

Table 3-4. Partitioner Concept

Pseudosignature	Semantics
`P::~P( )`	Destructor.
`template <typename Range> bool P::should_execute_range( const Range &r, const task &t)`	`True` if `r` should be passed to the body of `t`. `False` if `r` should instead be split.
`P::P( P& p, split )`	Split `p` into two partitioners.

Description

The partitioner implements rules for deciding when a given range should no longer be subdivided, but should be operated over as a whole by a task's body.

The default behavior of the algorithms `parallel_for`, `parallel_reduce`, and `parallel_scan` is to recursively split a range until no subrange remains that is divisible, as decided by the function `is_divisible` of the Range Concept. The Partitioner Concept models rules for the early termination of the recursive splitting of a range, providing the ability to change the default behavior. A `Partitioner` object's decision making is implemented using two functions: a splitting constructor and the function `should_execute_range`.

Within the parallel algorithms, each `Range` object is associated with a `Partitioner` object. Whenever a `Range` object is split using its splitting constructor to create two subranges, the associated `Partitioner` object is likewise split to create two matching `Partitioner` objects.

When a `parallel_for`, `parallel_reduce`, or `parallel_scan` algorithm needs to decide whether to further subdivide a range, it invokes the function `should_execute_range` for the `Partitioner` object associated with the range. If the function returns `true` for the given range and task, no further splits are performed on the range and the current task applies its body over the entire range.

Model Types: Partitioners

simple_partitioner models the default behavior of splitting a range until it cannot be further subdivided.

auto_partitioner models an adaptive behavior that monitors the work-stealing actions of the task_scheduler to reduce the number of splits performed.

simple_partitioner Class

A class that models the default range-splitting behavior of the parallel_for, parallel_reduce, and parallel_scan algorithms, where a range is recursively split until it cannot be further subdivided.

```
#include "tbb/partitioner.h"

class simple_partitioner;
```

Description

The class simple_partitioner models the default range-splitting behavior of the parallel_for, parallel_reduce, and parallel_scan algorithms.

simple_partitioner( )
: An empty default constructor.

simple_partitioner(simple_partitioner &partitioner, split)
: An empty splitting constructor.

template<typename Range> bool should_execute_range (const Range &r, const task &t)
: A function that returns true when the provided range should be executed to completion by the given task. It returns !range.is_divisible( ).

auto_partitioner Class

A class that models an adaptive partitioner that monitors the work-stealing actions of the task_scheduler to manage the number of splits performed.

```
#include "tbb/partitioner.h"

class auto_partitioner;
```

Description

The class auto_partitioner models an adaptive partitioner that limits the number of splits needed for load balancing by reacting to work-stealing events.

The range is first divided into S_I subranges, where S_I is proportional to the number of threads created by the task scheduler. These subranges are executed to completion by tasks unless they are stolen. If a subrange is stolen by an idle thread, the auto_partitioner further subdivides the range to create additional subranges.

The auto_partitioner creates additional subranges only if threads are actively stealing work. If the load is well balanced, the use of only a few large initial subranges reduces the overheads incurred when splitting and joining ranges. However, if there is a load imbalance that results in work stealing, the auto_partitioner creates additional subranges that can be stolen to more finely balance the load.

The auto_partitioner therefore attempts to minimize the number of range splits, while providing ample opportunities for work stealing.

`auto_partitioner( )`
: An empty default constructor.

`auto_partitioner(auto_partitioner &partitioner, split )`
: A splitting constructor that divides the auto_partitioner partitioner into two partitioners.

`template<typename Range> bool should_execute_range (const Range &r, const task &t)`
: A function that returns true when the provided range should be operated on as a whole by the given task's body. This function may return true even if range.is_divisible( ) == true and always returns true if range.is_divisible( ) == false. That is, this function may decide that t should process an r that can be further subdivided, but it always decides that t should process an r that cannot be further subdivided.

Table 3-5 provides guidance for selecting between the simple_partitioner and auto_partitioner classes.

Table 3-5. Guidance for selecting a partitioner

Partitioner type	Discussion
simple_partitioner	Recursively splits a range until it is no longer divisible. The Range::is_divisible function is wholly responsible for deciding when recursive splitting halts. When used with classes such as blocked_range and blocked_range2d, the selection of an appropriate grain size is therefore critical to allow concurrency while limiting overhead.
auto_partitioner	Guides splitting decisions based on the work-stealing behavior of the task scheduler. When used with classes such as blocked_range and blocked_range2d, the selection of an appropriate grain size is less important. Subranges that are larger than the grain size are used unless load imbalances are detected. Therefore, acceptable performance may often be achieved by simply using the default grain size of 1.

Ranges larger than the grain size may be passed to the body when using an auto_partitioner. The body should therefore not use the value of grainsize as an upper bound on the size of the range (for allocating temporary storage, for example).

parallel_for<Range,Body> Template Function

Template for a function that performs parallel iteration over a range of values.

```
#include "tbb/parallel_for.h"
template<typename Range, typename Body>
void parallel_for( const Range& range, const Body& body );
template<typename Range, typename Body, typename Partitioner>
  void parallel_for( const Range& range, const Body& body,
                     Partitioner &partitioner );
```

Description

A `parallel_for<Range,Body>` represents parallel execution of `Body` over each value in `Range`. Type `Range` must model the Range Concept. The body must model the requirements in Table 3-6.

Table 3-6. Requirements for parallel_for body

Pseudosignature	Semantics
`Body::Body( const Body& )`	Copy constructor
`Body::~Body( )`	Destructor
`void Body::operator( )( Range& range ) const`	Apply body to range

A `parallel_for` recursively splits the range into subranges to the point where `is_divisible( )` returns `false` for each subrange, and makes copies of the body for each of these subranges. For each such body/subrange pair, it invokes `Body::operator( )`. The invocations are interleaved with the recursive splitting in order to minimize space overhead and efficiently use the cache.

Some of the copies of the range and body may be destroyed after `parallel_for` returns. This late destruction is not an issue in typical usage, but it is something to be aware of when looking at execution traces or writing range or body objects with complex side effects.

When worker threads are available, `parallel_for` executes iterations in nondeterministic order. Do not rely upon any particular execution order for correctness. However, due to efficiency concerns, `parallel_for` tends to operate on consecutive runs of values.

When no worker threads are available, `parallel_for` executes iterations from left to right in the following sense. Imagine drawing a binary tree that represents the recursive splitting. Each nonleaf node represents splitting a subrange `r` by invoking the splitting constructor `Range(r,split( ))`. The left child represents the updated value of `r`. The right child represents the newly constructed object. Each leaf in the tree represents an indivisible subrange. The method `Body::operator( )` is invoked on each leaf subrange, from left to right.

Complexity

If the range and body take O(1) space, and the range splits into nearly equal pieces, the space complexity is O(*p* log *n*), where *p* is the number of threads and *n* is the size of the range.

parallel_reduce<Range,Body> Template Function

Computes reduction over a range of values.

```
#include "tbb/parallel_reduce.h"

template<typename Range, typename Body>

void parallel_reduce( const Range& range, Body& body );

template<typename Range, typename Body, typename Partitioner>

  void parallel_reduce( const Range& range, Body& body,

                        Partitioner &partitioner );
```

Description

A parallel_reduce<Range,Body> performs parallel reduction of Body over each value in Range. Type Range must model the Range Concept. The body must model the requirements in Table 3-7.

Table 3-7. Requirements for parallel_reduce body

Pseudosignature	Semantics
Body::Body(Body&, split);	Splitting constructor. Must be able to run the operator( ) and join methods concurrently.
Body::~Body( )	Destructor.
void Body::operator( )(Range& range);	Accumulate results for the subrange.
void Body::join(Body& rhs);	Join results. The result in rhs should be merged into the result of this.

A parallel_reduce recursively splits the range into subranges to the point where is_divisible( ) returns false for each subrange. A parallel_reduce uses the splitting constructor to make one or more copies of the body for each thread. It may copy a body while the body's operator( ) or join method runs concurrently. You are responsible for ensuring the safety of such concurrency. In typical usage, the safety requires no extra effort.

When worker threads are available, parallel_reduce invokes the splitting constructor for the body. For each such split, it invokes the join method after processing in order to merge the results from the bodies.

Complexity

If the range and body take O(1) space, and the range splits into nearly equal pieces, the space complexity is O(*p* log *n*), where *p* is the number of threads and *n* is the size of the range.

parallel_scan<Range,Body> Template Function

Template function that computes parallel prefix.

```
#include "tbb/parallel_scan.h"
template<typename Range, typename Body>
  void parallel_scan( const Range& range, Body& body );
template<typename Range, typename Body, typename Partitioner>
    void parallel_scan( const Range& range, Body& body,
                        Partitioner &partitioner );
```

Description

A `parallel_scan<Range,Body>` computes a parallel prefix, also known as a parallel scan. This can be useful in scenarios that appear to have inherently serial dependencies. Given an associative operation $\oplus$ with left-identity element $\mathtt{id}_{\oplus}$, the parallel prefix of $\oplus$ over a sequence x_0, x_1, ... x_{n-1} is a sequence y_0, y_1, y_2, ... y_{n-1}, where $y_0 = id_{\oplus} \oplus x_0$ and $y_i = y_{i-1} \oplus x_i$.

The template `parallel_scan<Range,Body>` implements a parallel prefix generically. The body must model the requirements in Table 3-8.

Table 3-8. parallel_scan requirements

Pseudosignature	Semantics
`void Body::operator()(const Range& r, pre_scan_tag)`	Preprocess iterations for range `r`.
`void Body::operator()(const Range& r, final_scan_tag)`	Do final processing for iterations of range `r`.
`Body::Body(Body& b, split)`	Split `b` so that this and `b` can accumulate separately.
`Void Body::reverse_join(Body& a)`	Merge preprocessing state of `a` into this, where `a` was created earlier from `b` by `b`'s splitting constructor.
`Void Body::assign(Body& b)`	Assign state of `b` to this.

pre_scan_tag and final_scan_tag Classes

Types that distinguish the phases of `parallel_scan`.

```
#include "tbb/parallel_scan.h"
struct pre_scan_tag;
struct final_scan_tag;
```

Description

The types `pre_scan_tag` and `final_scan_tag` are dummy types used in conjunction with `parallel_scan`.

Members

```
namespace tbb {

    struct pre_scan_tag {
        static bool is_final_scan();
    };

    struct final_scan_tag {
        static bool is_final_scan();
    };

}
```

`bool is_final_scan()`

Returns: `true` for a `final_scan_tag`; `false` otherwise.

Summary of Loops

The high-level loop templates in Intel Threading Building Blocks give you efficient, scalable ways to exploit the power of multi-core chips without having to start from scratch. They let you design your software at a concurrent task level and not worry about low-level manipulation of threads. Because they are generic, you can customize them to your specific needs. Although algorithms in this chapter can unlock the power of multi-core processing for many applications, sometimes you will require more complex algorithms. The next chapter takes the models shown in this chapter to a higher level.

CHAPTER 4

Advanced Algorithms

Algorithm templates are the keys to using Intel Threading Building Blocks. This chapter presents some relatively complex algorithms that build on the foundation laid in Chapter 3, so you should understand Chapter 3 before jumping into this chapter. This chapter covers Threading Building Blocks' support for the following types of generic parallel algorithms.

Parallel algorithms for streams:

`parallel_while`
: Use for an unstructured stream or pile of work. Offers the ability to add additional work to the pile while running.

`pipeline`
: Use when you have a linear sequence of stages. Specify the maximum number of items that can be in transit. Each stage can be serial or parallel. This uses the cache efficiently because each worker thread takes an item through as many stages as possible, and the algorithm is biased toward finishing old items before tackling new ones.

Parallel sort:

`parallel_sort`
: A comparison sort with an average time complexity not to exceed $O(n \log n)$ on a single processor and approaching $O(N)$ as more processors are used. When worker threads are available, `parallel_sort` creates subtasks that may be executed concurrently.

Parallel Algorithms for Streams

You can successfully parallelize many applications using only the constructs discussed thus far. However, some situations call for other parallel patterns. This section describes the support for some of these alternative patterns:

`parallel_while`
: Use for an unstructured stream or pile of work. Offers the ability to add additional work to the pile while running.

`pipeline`
: Use when you have a linear pipeline of stages. Specify the maximum number of items that can be in flight. Each stage can be serial or parallel. This uses the cache efficiently because each worker thread handles an item through as many stages as possible, and the algorithm is biased toward finishing old items before tackling new ones.

Cook Until Done: parallel_while

For some loops, the end of the iteration space is not known in advance, or the loop body may add more iterations to do before the loop exits. You can deal with both situations using the template class `tbb::parallel_while`.

A linked list is an example of an iteration space that is not known in advance. In parallel programming, it is usually better to use dynamic arrays instead of linked lists because accessing items in a linked list is inherently serial. But if you are limited to linked lists, if the items can be safely processed in parallel, and if processing each item takes at least a few thousand instructions, you can use `parallel_while` in a situation where the serial form is as shown in Example 4-1.

Example 4-1. Original list processing code

```
void SerialApplyFooToList( Item*root ) {
    for( Item* ptr=root; ptr!=NULL; ptr=ptr->next )
        Foo(pointer->data);
}
```

If `Foo` takes at least a few thousand instructions to run, you can get parallel speedup by converting the loop to use `parallel_while`. Unlike the templates described earlier, `parallel_while` is a class, not a function, and it requires *two* user-defined objects. The first object defines the stream of items. The object must have a method, `pop_if_present`, such that when `bool b = pop_if_present(v)` is invoked, it sets `v` to the next iteration value if there is one and returns `true`. If there are no more iterations, it returns `false`. Example 4-2 shows a typical implementation of `pop_if_present`.

Example 4-2. pop_if_present for a parallel_while

```
class ItemStream {
    Item* my_ptr;
public:
    bool pop_if_present( Item*& item ) {
        if( my_ptr ) {
            item = my_ptr;
            my_ptr = my_ptr->next;
            return true;
```

Example 4-2. pop_if_present for a parallel_while (continued)

```
        } else {
            return false;
        }
    };
    ItemStream( Item* root ) : my_ptr(root) {}
}
```

The second object defines the loop body, and must define an `operator( ) const` and an `argument_type` member type. This is similar to a C++ function object from the C++ standard header, `<functional>`, except that it must be `const` (see Example 4-3).

Example 4-3. Use of parallel_while

```
class ApplyFoo {
public:
    void operator( )( Item* item ) const {
        Foo(item->data);
    }
    typedef Item* argument_type;
};
```

Given the stream and body classes, the new code is as shown in Example 4-4.

Example 4-4. ParallelApplyFooToList

```
void ParallelApplyFooToList( Item*root ) {
    parallel_while<ApplyFoo> w;
    ItemStream stream;
    ApplyFoo body;
    w.run( stream, body );
}
```

The `pop_if_present` method does not have to be thread-safe for a given stream because `parallel_while` never calls it concurrently for the same stream. Notice that this convenience makes `parallel_while` nonscalable because the fetching is serialized. But in many situations, you still get useful speedup over doing things sequentially.

`parallel_while` may concurrently invoke `pop_if_present` on the same object, but only if the object is in different streams.

There is a second way that `parallel_while` can acquire work, and this is the way it can become scalable. The body of a `parallel_while` `w`, if given a reference to `w` when it is constructed, can add more work by calling `w.add(item)`, where `item` is of type `Body::argument_type`.

For example, perhaps processing a node in a tree is a prerequisite to processing its descendants. With `parallel_while`, after processing a node, you could use `parallel_while::add` to add the descendant nodes. The instance of `parallel_while` does not terminate until all items have been processed.

Notes on parallel_while scaling

Use of `parallel_while` usually does not provide scalable parallelism if the `add` method is not used because the input stream typically acts as a bottleneck. However, this bottleneck is broken if the stream is used to get things started and further items come from prior items invoking the `add` method.

Even in the nonscalable case, `parallel_while` covers a commonly desired idiom of walking a sequential structure (e.g., a linked list) and dispatching concurrent work for each item in the structure.

parallel_while Template Class

Template class that processes work items.

```
#include "tbb/parallel_while.h"

template<typename Body> class parallel_while;
```

Description

A `parallel_while<Body>` performs parallel iteration over items. The processing to be performed on each item is defined by a function object of type `Body`. The items are specified in two ways:

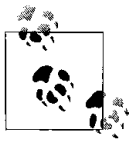
- An initial stream of items
- Additional items that are added while the stream is being processed

Table 4-1 shows the requirements on the stream and body.

Table 4-1. Requirements for parallel_while stream and body

Pseudosignature	Semantics
`bool S::pop_if_present( B::argument_type& item )`	Get next stream item. `parallel_while` does not concurrently invoke the method on the same object.
`B::operator( )( B::argument_type& item ) const`	Process item. `parallel_while` may concurrently invoke the operator for the same object but a different item.
`B::argument_type( )`	Default constructor.
`B::argument_type( const B::argument_type& )`	Copy constructor.
`~B::argument_type( )`	Destructor.

For example, a unary function object, as defined in Section 20.3 of the C++ standard, models the requirements for `B`. A `concurrent_queue` models the requirements for `S`.



To achieve speedup, the grain size of `B::operator()` needs to be on the order of 10,000 instructions to execute. Otherwise, the internal overheads of `parallel_while` swamp the useful work. The parallelism in `parallel_while` is not scalable if all the items come from the input stream. To achieve scaling, design your algorithm such that the `add` method often adds more than one piece of work.

Members

```
namespace tbb {
    template<typename Body>
    class parallel_while {
    public:
        parallel_while();
        ~parallel_while();

        typedef typename Body::argument_type value_type;

        template<typename Stream>
        void run( Stream& stream, const Body& body );

        void add( const value_type& item );
    };
}
```

`parallel_while<Body>()`

Effects: constructs a `parallel_while` that is not yet running.

`~parallel_while<Body>()`

Effects: destroys a `parallel_while`.

`Template <typename Stream> void run( Stream& stream, const Body& body )`

Effects: applies `body` to each item in `stream` and any other items that are added by the method `add`. Terminates when both of the following conditions become true:

- `stream.pop_if_present` returns `false`
- `body(x)` has returned for all items `x` generated from the stream or the `add` method

`void add( const value_type& item )`

Requirements: must be called from a call to `body.operator()` by `parallel_while`. Otherwise, the termination semantics of the `run` method are undefined.

Effects: adds item to collection of items to be processed.

Working on the Assembly Line: Pipeline

Pipelining is a common parallel pattern that mimics a traditional manufacturing assembly line. Data flows through a series of pipeline stages and each stage processes the data in some way. Given an incoming stream of data, some of these stages can operate in parallel, and others cannot. For example, in video processing, some operations on frames do not depend on other frames and so can be done on multiple frames at the same time. On the other hand, some operations on frames require processing prior frames first.

Pipelined processing is common in multimedia and signal processing applications. A classical thread-per-stage implementation suffers two problems:

- The speedup is limited to the number of stages.
- When a thread finishes a stage, it must pass its data to *another* thread.

Eliminating these problems is desirable. To do so, you specify whether a stage is serial or parallel. A serial stage processes items one at a time, in order. A parallel stage may process items out of order or concurrently. By allowing some stages to be run concurrently, you make available more opportunities for load balancing and concurrency. Given sufficient processor cores and concurrent opportunities, the throughput of the pipeline is limited to the throughput of the slowest serial filter.

The `pipeline` and `filter` classes implement the pipeline pattern. Here we'll look at a simple text-processing problem to demonstrate the usage of `pipeline` and `filter`. The problem is to read a text file, capitalize the first letter of each word, and write the modified text to a new file. Figure 4-1 illustrates the pipeline.

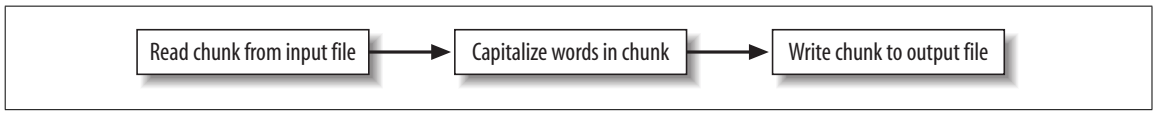


Figure 4-1. Pipeline

Assume that the file I/O is sequential. However, the capitalization stage can be done in parallel. That is, if you can serially read *n* chunks very quickly, you can capitalize the *n* chunks in parallel, as long as they are written in the proper order to the output file.

To decide whether to capitalize a letter, inspect whether the *preceding* character is a blank. For the first letter in each chunk, you must inspect the last letter of the preceding chunk. But doing so would introduce a complicated dependency in the middle stage.

The solution is to have each chunk also store the last character of the preceding chunk. The chunks overlap by one character. This *overlapping window* strategy is quite common to pipeline-processing problems. In the example, the window is represented by an instance of the `MyBuffer` class. It looks like a typical Standard Template Library (STL) container for characters, except that `begin()[-1]` is legal and holds the last character of the preceding chunk (see Example 4-5).

Example 4-5. Use of pipeline class

```
// Buffer that holds block of characters and last character of preceding buffer.
class MyBuffer {
    static const size_t buffer_size = 10000;
    char* my_end;
    // storage[0] holds the last character of the preceding buffer.
    char storage[1+buffer_size];
public:
    // Pointer to first character in the buffer
```

Example 4-5. Use of pipeline class (continued)

```
    char* begin( ) {return storage+1;}
    const char* begin( ) const {return storage+1;}
    // Pointer to one past last character in the buffer
    char* end( ) const {return my_end;}
    // Set end of buffer.
    void set_end( char* new_ptr ) {my_end=new_ptr;}
    // Number of bytes a buffer can hold
    size_t max_size( ) const {return buffer_size;}
    // Number of bytes in buffer.
    size_t size() const {return my_end-begin( );}
};
// Below is the top-level code for building and running the pipeline
    // Create the pipeline
    tbb::pipeline pipeline;

    // Create file-reading stage and add it to the pipeline
    MyInputFilter input_filter( input_file );
    pipeline.add_filter( input_filter );

    // Create capitalization stage and add it to the pipeline
    MyTransformFilter transform_filter;
    pipeline.add_filter( transform_filter );

    // Create file-writing stage and add it to the pipeline
    MyOutputFilter output_filter( output_file );
    pipeline.add_filter( output_filter );

    // Run the pipeline
    pipeline.run( MyInputFilter::n_buffer );

    // Remove filters from pipeline before they are implicitly destroyed.
    pipeline.clear( );
```

The parameter passed to the `pipeline::run` method controls the level of parallelism. Conceptually, *tokens* flow through the pipeline. A serial stage must process each token one at a time, in order. A parallel stage can process multiple tokens in parallel.

If the number of tokens were unlimited, there might be a problem where the unordered stage in the middle keeps gaining tokens because the output stage cannot keep up. This situation typically leads to undesirable resource consumption by the middle stage. The parameter to the `pipeline::run` method specifies the maximum number of tokens that can be in flight. Once this limit is reached, the `pipeline` class doesn't create a new token at the input stage until another token is destroyed at the output stage.

This top-level code also shows the `clear` method that removes all stages from the pipeline. This call is required if the filters have to be destroyed before the pipeline. The pipeline is a container that holds filters, and as with most containers in C++, it is illegal to destroy an item while it is in the container.

Now look in detail at how the stages are defined. Each stage is derived from the `filter` class. First let's consider the output stage because it is the simplest (see Example 4-6).

Example 4-6. Output stage for pipeline

```
// Filter that writes each buffer to a file.
class MyOutputFilter: public tbb::filter {
    FILE* my_output_file;
public:
    MyOutputFilter( FILE* output_file );
    /*override*/void* operator()( void* item );
};

MyOutputFilter::MyOutputFilter( FILE* output_file ) :
    tbb::filter(/*is_serial=*/true),
    my_output_file(output_file)
{
}

void* MyOutputFilter::operator()( void* item ) {
    MyBuffer& b = *static_cast<MyBuffer*>(item);
    fwrite( b.begin(), 1, b.size(), my_output_file );
    return NULL;
}
```

The class is derived from the `filter` class. When its constructor calls the base class constructor for `filter`, it specifies that this is a serial filter. The class overrides the virtual method `filter::operator()`, which is the method invoked by the pipeline to process an item. The parameter `item` points to the item to be processed. The value returned points to the item to be processed by the next filter. Because this is the last filter, the return value is ignored, and thus can be `NULL`.

The middle stage is similar. Its `operator()` returns a pointer to the item to be sent to the next stage (see Example 4-7).

Example 4-7. Middle stage for pipeline

```
// Filter that changes the first letter of each word
// from lowercase to uppercase.
class MyTransformFilter: public tbb::filter {
public:
    MyTransformFilter();
    /*override*/void* operator()( void* item );
};

MyTransformFilter::MyTransformFilter() :
    tbb::filter(/*serial=*/false)
{}

/*override*/void* MyTransformFilter::operator()( void* item ) {
    MyBuffer& b = *static_cast<MyBuffer*>(item);
    bool prev_char_is_space = b.begin()[-1]==' ';
```

Example 4-7. Middle stage for pipeline (continued)

```
    for( char* s=b.begin(); s!=b.end(); ++s ) {
        if( prev_char_is_space && islower(*s) )
            *s = toupper(*s);
        prev_char_is_space = isspace(*s);
    }
    return &b;
}
```

The middle stage operates on purely local data. Thus, any number of invocations on `operator()` can run concurrently on the same instance of `MyTransformFilter`. The class communicates this fact to the pipeline by constructing its base class, `filter`, with the `<serial>` parameter set to `false`.

The input filter is the most complicated because it has to decide when the end of the input is reached and it must allocate buffers (see Example 4-8).

Example 4-8. Input stage for pipeline

```
class MyInputFilter: public tbb::filter {
public:
    static const size_t n_buffer = 4;
    MyInputFilter( FILE* input_file_ );
private:
    FILE* input_file;
    size_t next_buffer;
    char last_char_of_previous_buffer;
    MyBuffer buffer[n_buffer];
    /*override*/ void* operator()(void*);
};

MyInputFilter::MyInputFilter( FILE* input_file_ ) :
    filter(/*is_serial=*/true),
    next_buffer(0),
    input_file(input_file_),
    last_char_of_previous_buffer(' ')
{
}

void* MyInputFilter::operator()(void*) {
    MyBuffer& b = buffer[next_buffer];
    next_buffer = (next_buffer+1) % n_buffer;
    size_t n = fread( b.begin(), 1, b.max_size(), input_file );
    if( !n ) {
        // end of file
        return NULL;
    } else {
        b.begin()[-1] = last_char_of_previous_buffer;
        last_char_of_previous_buffer = b.begin()[n-1];
        b.set_end( b.begin()+n );
        return &b;
    }
}
```

The input filter is serial because it is reading from a sequential file. The override of `operator()` ignores its parameter because it is generating a stream, not transforming it. It remembers the last character of the preceding chunk so that it can properly overlap windows.

The buffers are allocated from a circular queue of size `n_buffer`. This might seem risky because after the initial `n_buffer` input operations, buffers are recycled without any obvious checks as to whether they are still in use. But the recycling is indeed safe because of two constraints:

- The pipeline received `n_buffer` tokens when `pipeline::run` was called. Therefore, no more than `n_buffer` buffers are ever in flight simultaneously.
- The last stage is serial. Therefore, the buffers are retired by the last stage in the order they were allocated by the first stage.

Notice that if the last stage were *not* serial, you would have to keep track of which buffers are currently in use because buffers might be retired out of order.

The directory *examples/pipeline/text_filter* that comes with Threading Building Blocks contains the complete code for the text filter.

Throughput of pipeline

The throughput of a pipeline is the rate at which tokens flow through it, and it is limited by two constraints. First, if a pipeline is run with *n* tokens, there obviously cannot be more than *n* operations running in parallel. Selecting the right value of *n* may involve some experimentation. Too low a value limits parallelism; too high a value may demand too many resources (for example, more buffers).

Second, the throughput of a pipeline is limited by the throughput of the slowest sequential stage. This is true even for a pipeline with no parallel stages. No matter how fast the other stages are, the slowest sequential stage is the bottleneck. So in general, you should try to keep the sequential stages fast and, when possible, shift work to the parallel stages.

The text-processing example has relatively poor speedup because the serial stages are limited by the I/O speed of the system. Indeed, even when files are on a local disk, you are unlikely to see a speedup of much more than 2X. To really benefit from a pipeline, the parallel stages need to be doing more substantial work compared to the serial stages.

The window size, or subproblem size for each token, can also limit throughput. Making windows too small may cause overheads to dominate the useful work. Making windows too large may cause them to spill out of cache. A good guideline is to try for a large window size that still fits in cache. You may have to experiment a bit to find a good window size.

Nonlinear pipelines

The `pipeline` template supports only linear pipelines. It does not directly handle more baroque plumbing, such as in Figure 4-2.

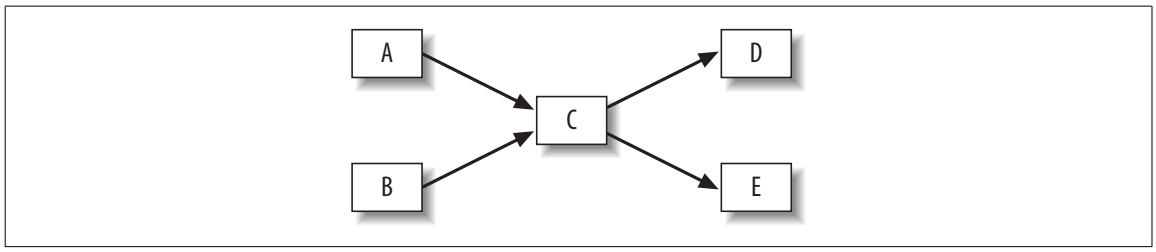


Figure 4-2. Nonlinear pipeline

However, you can still use `pipeline` for this. One solution is to topologically sort the stages into a linear order, as in Figure 4-3. Another solution, which injects dummy stages to get lower latency, is provided in Chapter 11 in the section titled "Two Mouths: Feeding Two from the Same Task in a Pipeline."

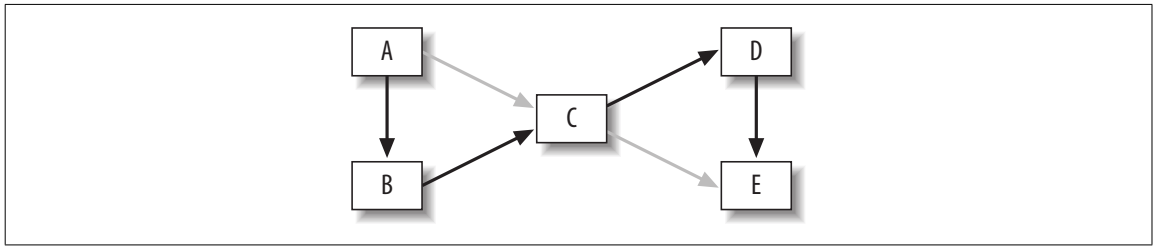


Figure 4-3. Topologically sorted pipeline

In the topological sorting of the stages (Figure 4-3), the light gray arrows are the original arrows that are now implied by transitive closure of the other arrows. It might seem that a lot of parallelism is lost by forcing a linear order on the stages, but in fact, the only loss is in the *latency* of the pipeline, not the throughput. The latency is the time it takes a token to flow from the beginning to the end of the pipeline. Given a sufficient number of processors, the latency of the original nonlinear pipeline is three stages. This is because stages A and B could process the token concurrently, and likewise, stages D and E could process the token concurrently. In the linear pipeline, the latency is five stages. The behavior of stages A, B, D, and E may need to be modified to properly handle objects that don't need to be acted upon by the stage, other than to be passed along to the next stage in the pipeline.

The throughput remains the same because, regardless of the topology, the throughput is still limited by the throughput of the slowest serial stage. If `pipeline` supported nonlinear pipelines, it would add a lot of programming complexity, and would not improve throughput. The linear limitation of `pipeline` is a good trade-off of gain versus pain.

pipeline Class

Abstract base class that performs pipelined execution.

```
#include "tbb/pipeline.h"

class pipeline;
```

Description

A `pipeline` represents the pipelined application of a series of filters to a stream of items. Each `filter` is `parallel` or `serial`. See the `filter` class for details.

A pipeline contains one or more filters, denoted here as f_i, where i denotes the position of the filter in the pipeline. The pipeline starts with filter f_0, followed by f_1, f_2, and so on. The following steps describe how to use the class:

1. Derive f_i classes from `filter`. The constructor for f_i specifies whether it is serial via the Boolean parameter to the constructor for the base class `filter`.
2. Override the virtual method `filter::operator()` to perform the filter's action on the item, and return a pointer to the item to be processed by the next filter. The first filter, f_0, generates the stream. It should return `NULL` if there are no more items in the stream. The return value for the last filter is ignored.
3. Create an instance of class `pipeline`.
4. Create instances of the f_i filters and add them to the pipeline in order from first to last. An instance of a filter can be added once, at most, to a pipeline. A filter should never be a member of more than one pipeline at a time.
5. Call the method `pipeline::run`. The parameter `max_number_of_live_tokens` puts an upper bound on the number of stages that will be run concurrently. Higher values may increase concurrency at the expense of more memory consumption from having more items in flight.

Given sufficient processors and tokens, the throughput of the pipeline is limited to the throughput of the slowest serial filter.

A filter must be removed from the pipeline before destroying it. You can accomplish this by destroying the pipeline first, or by calling `pipeline::clear()`.

Members

```
namespace tbb {
    class pipeline {
    public:
        pipeline();
        virtual ~pipeline();
        void add_filter( filter& f );
        void run( size_t max_number_of_live_tokens );
        void clear();
    };
}
```

`pipeline()`
: Effects: constructs a pipeline with no filters.

`~pipeline()`
: Effects: removes all filters from the pipeline and destroys the pipeline.

`void add_filter( filter& f )`
: Effects: appends filter `f` to the sequence of filters in the pipeline. The filter `f` must not already be in a pipeline.

`void run( size_t max_number_of_live_tokens )`
: Effects: runs the pipeline until the first filter returns `NULL` and each subsequent filter has processed all items from its predecessor. The number of items processed in parallel depends upon the structure of the pipeline and the number of available threads. At most, `max_number_of_live_tokens` are in flight at any given time.

`void clear( )`
: Effects: removes all filters from the pipeline.

filter Class

Abstract base class that represents a filter in a pipeline.

```
#include "tbb/pipeline.h"

class filter;
```

Description

A `filter` represents a filter in a `pipeline`. A filter is `parallel` or `serial`. A parallel filter can process multiple items in parallel and possibly out of order. A serial filter processes items one at a time in the original stream order. Parallel filters are preferred when viable because they permit parallel speedup. Whether the filter is serial or parallel is specified by an argument to the constructor.

The `filter` class should be used only in conjunction with `pipeline`.

Members

```
namespace tbb {
    class filter {
    protected:
        filter( bool is_serial );
    public:
        bool is_serial( ) const;
        virtual void* operator( )( void* item ) = 0;
        virtual ~filter( );
    };
}
```

`filter( bool is_serial )`
: Effects: constructs a serial filter if `is_serial` is `true`, or a parallel filter if `is_serial` is `false`.

`~filter( )`
: Effects: destroys the filter. The filter must not be in a pipeline; otherwise, memory might be corrupted. The debug version of the library raises an assertion failure if the filter is in a pipeline. Always clear or destroy the containing pipeline first. A way to remember this is that a pipeline acts like a container of filters, and a C++ container usually does not allow one to destroy an item while it is in the container.

`bool is_serial( ) const`

Returns: `true` if `filter` is `serial`; `false` if `filter` is `parallel`.

`virtual void* operator( )( void * item )`

Effects: the derived filter should override this method to process an item and return a pointer to `item` to be processed by the next filter. The `item` parameter is `NULL` for the first filter in the pipeline.

Returns: the first filter in a pipeline should return `NULL` if there are no more items to process. The result of the last filter in a pipeline is ignored.

parallel_sort

`parallel_sort` is a comparison sort with an average time complexity $O(n \log n)$. When worker threads are available, `parallel_sort` creates subtasks that may be executed concurrently. This sort provides an *unstable* sort of the sequence `[begin1, end1)`. Being an unstable sort means that it might not preserve the relative ordering of elements with equal keys.

The sort is deterministic; sorting the same sequence will produce the same result each time. The requirements on the iterator and sequence are the same as for `std::sort`.

A call to `parallel_sort(i,j,comp)` sorts the sequence `[i,j)` using the third argument `comp` to determine relative orderings. If `comp(x,y)` returns `true`, `x` appears before `y` in the sorted sequence. A call to `parallel_sort(i,j)` is equivalent to `parallel_sort(i,j,std::less<T>)`.

Example 4-9 shows two sorts. The sort of array `a` uses the default comparison, which sorts in ascending order. The sort of array `b` sorts in descending order by using `std::greater<float>` for comparison.

Example 4-9. Two sorts

```
#include "tbb/parallel_sort.h"
#include <math.h>

using namespace tbb;

const int N = 100000;
float a[N];
float b[N];

void SortExample( ) {
    for( int i = 0; i < N; i++ ) {
       a[i] = sin((double)i);
       b[i] = cos((double)i);
    }
    parallel_sort(a, a + N);
    parallel_sort(b, b + N, std::greater<float>( ));
}
```

parallel_sort<RandomAccessIterator, Compare> Template Function

Sorts a sequence.

```
#include "tbb/parallel_sort.h"
template<typename RandomAccessIterator>
  void parallel_sort(RandomAccessIterator begin,
                     RandomAccessIterator end);
template<typename RandomAccessIterator, typename Compare>
  void parallel_sort(RandomAccessIterator begin,
                     RandomAccessIterator end,
                     const Compare& comp );
```

Description

Performs an *unstable* sort of the sequence [`begin1`, `end1`). The requirements on the iterator and sequence are the same as for `std::sort`. Specifically, `RandomAccessIterator` must be a random access iterator, and its value type `T` must model the requirements in Table 4-2.

Table 4-2. Requirements on value type T of RandomAccessIterator for parallel_sort

Pseudosignature	Semantics
`void swap( T& x, T& y )`	Swaps `x` and `y`.
`bool Compare::operator( )( const T& x, const T& y )`	`True` if `x` comes before `y`; `false` otherwise.

Complexity

`parallel_sort` is a comparison sort with an average time complexity of $O(n \log n)$ on a single-processor core, where n is the number of elements in the sequence. Complexity reduces to $O(N)$ as the number of processors increases. When worker threads are available, `parallel_sort` creates subtasks that may be executed concurrently, leading to improved execution times.

CHAPTER 5
Containers

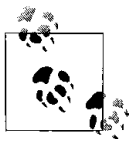
Intel Threading Building Blocks provides *highly concurrent* containers that permit multiple threads to invoke a method simultaneously on the same container. At this time, a concurrent queue, vector, and hash map are provided. All of these highly concurrent containers can be used with this library, OpenMP, or raw threads.

Highly concurrent containers are very important because Standard Template Library (STL) containers generally are not concurrency-friendly, and attempts to modify them concurrently can easily corrupt the containers. As a result, it is standard practice to wrap a lock (mutex) around STL containers to make them safe for concurrent access, by letting only one thread operate on the container at a time. But that approach eliminates concurrency, and thus is not conducive to multi-core parallelism.

As much as possible, the interfaces of the Threading Building Blocks containers are similar to STL, but they do not match completely because some STL interfaces are inherently not thread-safe. The Threading Building Blocks containers provide fine-grained locking or lock-free implementations, and sometimes both.

Fine-grained locking
: Multiple threads operate on the container by locking only those portions they really need to lock. As long as different threads access different portions, they can proceed concurrently.

Lock-free algorithms
: Different threads proceed straight through the operation without locks, accounting for and correcting the effects of other interfering threads. There is inevitably some waiting at the end, but the contention over locks can be avoided during the operations.



Locks are worth avoiding because they limit concurrency, and mistakes create problems that are difficult to debug. Threading Building Blocks avoids the need for locks, but does not guarantee you freedom from locks.

Highly concurrent containers come at a cost. They typically have higher overhead than regular STL containers, and so operations on highly concurrent containers may take longer than for STL containers. Therefore, you should use highly concurrent containers when the speedup from the additional concurrency that they enable outweighs their slower sequential performance.

Unlike STL, the Intel Threading Building Blocks containers are not templated with an allocator argument. The library retains control over memory allocation.

Why Do Containers Not Use an Allocator Argument?

An allocator argument is not supported for two reasons. First, it would lead to code bloat because much more of the container class code would be in the headers. Second, there is an advantage in having detailed control over memory allocation, so trade-offs can be made between cache-aligned memory and packed memory to avoid always paying for alignment.

The containers use a mix of `cache_aligned_allocator` and the default operator, `new`. The `cache_aligned_allocator` uses the scalable allocator (Chapter 6) if it is present. There are places for improvement; for instance, the `concurrent_hash_map` should use the `scalable_allocator` in a future version to enhance its scalability.

There is no requirement to link in the scalable allocator, as it will default to using `malloc`. Performance will likely be better if you link with the scalable allocator.

If you have your own scalable memory allocator which you prefer, you will have to work a little to force Threading Building Blocks containers to use it. The only recourse currently is to have your allocator connected using the same interface as the Threading Building Blocks `scalable_allocator` (the four C routines prototyped in *tbb/scalable_allocator.h*). Perhaps a future version will address this area.

concurrent_queue

The template class `concurrent_queue<T>` implements a concurrent queue with values of type `T`. Multiple threads may simultaneously push and pop elements from the queue.

In a single-threaded program, a queue is a first-in first-out structure. But if multiple threads are pushing and popping concurrently, the definition of *first* is uncertain. The only guarantee of ordering offered by `concurrent_queue` is that if a thread pushes multiple values, and another thread pops those same values, they will be popped in the same order that they were pushed.

Pushing is provided by the `push` method. There are blocking and nonblocking flavors of pop:

`pop_if_present`
: This method is nonblocking: it attempts to pop a value, and if it cannot because the queue is empty, it returns anyway.

`pop`
: This method blocks until it pops a value. If a thread must wait for an item to become available and it has nothing else to do, it should use `pop(item)` and not `while(!pop_if_present(item)) continue;` because `pop` uses processor resources more efficiently than the loop.

Unlike most STL containers, `concurrent_queue::size_type` is a *signed* integral type, not unsigned. This is because `concurrent_queue::size( )` is defined as the number of push operations started minus the number of pop operations started. If pops outnumber pushes, `size( )` becomes negative. For example, if a `concurrent_queue` is empty and there are `n` pending pop operations, `size( )` returns `-n`. This provides an easy way for producers to know how many consumers are waiting on the queue. In particular, consumers may find the `empty( )` method useful; it is defined to be true if and only if `size( )` is not positive.

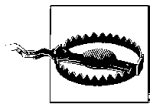
By default, a `concurrent_queue<T>` is unbounded. It may hold any number of values until memory runs out. It can be bounded by setting the queue capacity with the `set_capacity` method. Setting the capacity causes `push` to block until there is room in the queue. Bounded queues are slower than unbounded queues, so if there is a constraint elsewhere in your program that prevents the queue from becoming too large, it is better not to set the capacity.

Iterating over a concurrent_queue for Debugging

The template class `concurrent_queue` supports STL-style iteration. This support is intended only for debugging, when you need to dump a queue. It provides iterator and `const_iterator` types. Both follow the usual STL conventions for forward iterators. The iterators go in the forward direction only and are too slow to be very useful in production code. The iteration order is from least recently pushed item to most recently pushed item. If a queue is modified, all iterators pointing to it become invalid and unsafe to use. The snippet of code in Example 5-1 dumps a queue. The operator `<<` is defined for a `Foo`.

Example 5-1. Concurrent queue listing dump program

```
concurrent_queue<Foo> q;
...
for(concurrent_queue<Foo>::const_iterator i(q.begin()); i!=q.end( ); ++i ) {
    cout << *i;
}
```



The iterators are relatively slow. You should use them *only* for debugging.

When Not to Use Queues

Queues are widely used in parallel programs to buffer consumers from producers. Before using an explicit queue, however, consider using `parallel_while` or `pipeline` instead. These options are often more efficient than queues for the following reasons:

- A queue is inherently a bottleneck because it must maintain first-in first-out order.
- A thread that is popping a value may have to wait idly until the value is pushed.
- A queue is a passive data structure. If a thread pushes an item and another thread pops it, the item must be moved to the other processor. Even if the original thread pops the item, enough time could elapse between the push and the pop for the item (and whatever it references) to be discarded from the cache.

In contrast, `parallel_while` and `pipeline` avoid these bottlenecks. Because their threading is implicit, they optimize the use of worker threads so that they do other work until a value shows up. They also try to keep items hot in the cache. For example, when another work item is added to a `parallel_while`, it is kept local to the thread that added it, unless another idle thread can steal it before the "hot" thread processes it. By applying this selectivity in pulling from another queue, items are more often processed by the hot thread.

---

concurrent_queue Template Class

Template class for queue with concurrent operations.

```
#include "tbb/concurrent_queue.h"

template<typename T> class concurrent_queue;
```

Description

A `concurrent_queue` is a bounded data structure that permits multiple threads to concurrently push and pop items. Its behavior is first-in first-out in reference to any items pushed by a single thread and popped by a single thread. The default bounds are large enough to make the queue practically unbounded, subject to memory limitations on the target machine.

The interface is different from that of an STL `std::queue` because `concurrent_queue` is designed for concurrent operations. See Table 5-1 for the differences.

Table 5-1. std::queue versus tbb::concurrent_queue

Feature	STL std::queue	tbb::concurrent_queue
Access to front and back	`front` and `back` methods	Not present. They would be unsafe while concurrent operations are in progress.
`size_type`	Unsigned integral type	Signed integral type.
`size( )`	Returns the number of items in the queue	Returns the number of pushes minus the number of pops. Waiting push or pop operations are included in the difference. The size is negative if there are pops waiting for corresponding pushes.
Copy and pop item from queue `q`	`x=q.front( )` `q.pop( )`	`q.pop(x)` waits for the object indefinitely. For an immediate return, use `pop_if_present`.

Copy and pop item unless queue `q` is empty	`bool b=!q.empty( );`
	`if(b) {`
	`  x=q.front( );`
	`  q.pop( );`
	`}`
		`q.pop_if_present(x)` returns `true`, with the object in `x`, if an object is available; otherwise returns `false` immediately (no waiting).

Pop of empty queue	Not allowed	Waits until an item becomes available.

If the push or pop operation blocks, it blocks using a user-space lock, which can waste processor resources when the blocking time is long. The `concurrent_queue` class is designed for situations where the blocking time is typically short relative to the rest of the application time.

Members

```
namespace tbb {
    template<typename T>
    class concurrent_queue {
    public:
        // types
        typedef T value_type;
        typedef T& reference;
        typedef const T& const_reference;
        typedef std::ptrdiff_t size_type;
        typedef std::ptrdiff_t difference_type;

        concurrent_queue( ) {}
        ~concurrent_queue( );

        void push( const T& source );
        void pop( T& destination );
        bool pop_if_present( T& destination );
        size_type size() const {return internal_size( );}
        bool empty( ) const;
        size_t capacity( );
        void set_capacity( size_type capacity );

        typedef implementation-defined iterator;
        typedef implementation-defined const_iterator;

        // iterators (these are slow and intended only for debugging)
```

```
            iterator begin( );
            iterator end( );
            const_iterator begin( ) const;
            const_iterator end( ) const;
        };
    }
```

`concurrent_queue( )`
: Effects: constructs an empty queue.

`~concurrent_queue( )`
: Effects: destroys all items in the queue.

`void push( const T& source )`
: Effects: waits until `size( )<capacity`, then pushes a copy of `source` onto the back of the queue.

`void pop( T& destination )`
: Effects: waits until a value becomes available and pops it from the queue. Assigns it to `destination`. Destroys the original value.

`bool pop_if_present( T& destination )`
: Effects: if a value is available, pops it from the queue, assigns it to `destination`, and destroys the original value. Otherwise, does nothing.

  Returns: `true` if value was popped; `false` otherwise.

`size_type size( ) const`
: Returns: number of pushes minus number of pops. The result is negative if there are pop operations waiting for corresponding pushes.

`bool empty( ) const`
: Returns: `size( )== 0`.

  This does not mean the queue is really empty. Because `size` is the difference between pushes and pops, `empty( )` can return `true` when there is work in flight.

`size_type capacity( ) const`
: Returns: maximum number of values that the queue can hold.

`void set_capacity( size_type capacity )`
: Effects: sets the maximum number of values that the queue can hold.

Example

Example 5-2 builds a queue with the integers `0...9`, and then dumps the queue to standard output. Its overall effect is to print `0 1 2 3 4 5 6 7 8 9`.

Example 5-2. Concurrent queue count

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

using namespace std;
using namespace tbb;

int main( ) {
    concurrent_queue<int> queue;
    for( int i=0; i<10; ++i )
```

Example 5-2. Concurrent queue count (continued)

```
        queue.push(i);
    for( concurrent_queue<int>::const_iterator i(queue.begin()); i!=queue.end(); ++i )
        cout << *i << " ";
    cout << endl;
    return 0;

}
```

`iterator begin()`
: Returns: iterator pointing to the beginning of the queue.

`iterator end()`
: Returns: iterator pointing to the end of the queue.

`const_iterator begin() const`
: Returns: `const_iterator` pointing to the beginning of the queue.

`const_iterator end() const`
: Returns: `const_iterator` pointing to the end of the queue.

concurrent_vector

A `concurrent_vector<T>` is a dynamically growable array of items of type `T` for which it is safe to simultaneously access elements in the vector while growing it. However, be careful not to let another task access an element that is under construction or is otherwise being modified. For safe concurrent growing, `concurrent_vector` has two methods for resizing that support common uses of dynamic arrays: `grow_by` and `grow_to_at_least`. The index of the first element is 0. The method `grow_by(n)` enables you to safely append `n` consecutive elements to a vector, and returns the index of the first appended element. Each element is initialized with `T()`. So for instance, Example 5-3 safely appends a C string to a shared vector.

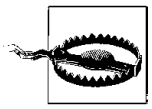
Example 5-3. Concurrent vector

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

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

The `size()` method returns the number of elements in the vector, which may include elements that are still undergoing concurrent construction by `grow_by` and `grow_to_at_least`. Also, it is safe to use 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. You must synchronize construction and access of an element.

A concurrent_vector<T> never moves an element until the array is cleared, which can be an advantage over the STL std::vector (which can move elements to resize the vector), even for single-threaded code. However, concurrent_vector does have more overhead than std::vector. Use concurrent_vector only if you really need to dynamically resize it while other accesses are (or might be) in flight, or if you require that an element never move.



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

concurrent_vector Template Class

Template class for vector that can be concurrently grown and accessed.

```
#include "tbb/concurrent_vector.h"

template<typename T> class concurrent_vector;
```

Members

```
namespace tbb {
    template<typename T>
    class concurrent_vector {
    public:
        typedef size_t size_type;
        typedef T value_type;
        typedef ptrdiff_t difference_type;
        typedef T& reference;
        typedef const T& const_reference;

        // whole vector operations
        concurrent_vector( ) {}
        concurrent_vector( const concurrent_vector& );
        concurrent_vector& operator=( const concurrent_vector&);
        ~concurrent_vector( );
        void clear( );

        // concurrent operations
        size_type grow_by( size_type delta );
        void grow_to_at_least( size_type new_size );
        size_type push_back( const_reference value );
        reference operator[]( size_type index );
        const_reference operator[]( size_type index ) const;

        // parallel iteration
        typedef implementation-defined iterator;
        typedef implementation-defined const_iterator;
        typedef generic_range_type<iterator> range_type;
        typedef generic_range_type<const_iterator> const_range_type;
```

```
        range_type range( size_t grainsize );
        const_range_type range( size_t grainsize ) const;

        // capacity
        size_type size( ) const;
        bool empty( ) const;
        size_type capacity( ) const;
        void reserve( size_type n );
        size_type max_size( ) const;

        // STL support
        iterator begin( );
        iterator end( );
        const_iterator begin( ) const;
        const_iterator end( ) const;

        typedef implementation-defined reverse_iterator;
        typedef implementation-defined const_reverse_iterator;
        iterator rbegin( );
        iterator rend( );
        const_iterator rbegin( ) const;
        const_iterator rend( ) const;
    };
}
```

Whole Vector Operations

These operations are *not* thread-safe on the same instance:

`concurrent_vector( )`
: Effects: constructs an empty vector.

`concurrent_vector( const concurrent_vector& src )`
: Effects: constructs a copy of `src`.

`concurrent_vector& operator=( const concurrent_vector& src )`
: Effects: assigns contents of `src` to `*this`.

  Returns: reference to lefthand side.

`~concurrent_vector( )`
: Effects: erases all elements and destroys the vector.

`void clear( )`
: Effects: erases all elements. Afterward, `size( )==0`.

Concurrent Operations

The methods described in this section safely execute on the same instance of a `concurrent_vector<T>`:

`size_type grow_by( size_type delta )`
: Effects: atomically appends `delta` elements to the end of the vector. The new elements are initialized with `T()`, where `T` is the type of the values in the vector.

  Returns: old size of the vector. If it returns *k*, the new elements are at the half-open index range [*k*...*k*+*delta*).

`void grow_to_at_least( size_type n )`
: Effects: grows the vector until it has at least `n` elements. The new elements are initialized with `T()`, where `T` is the type of the values in the vector.

`size_t push_back( const_reference value );`
: Effects: atomically appends a copy of `value` to the end of the vector.

  Returns: index of the copy.

`reference operator[]( size_type index )`
: Returns: reference to the element with the specified index.

`const_reference operator[]( size_type index ) const;`
: Returns: `const` reference to the element with the specified index.

Parallel Iteration

The types `const_range_type` and `range_type` model the Range Concept and provide the methods in Table 5-2 to access the bounds of the range. The types differ only in that the bounds for a `const_range_type` are of type `const_iterator`, whereas the bounds for a `range_type` are of type `iterator`.

Use the range types in conjunction with `parallel_for`, `parallel_reduce`, and `parallel_scan` to iterate over pairs in a `concurrent_vector`.

Table 5-2. Concept for concurrent_vector range R

Pseudosignature	Semantics
`R::iterator R::begin() const`	First item in range
`R::iterator R::end() const`	One past last item in range

`range_type range( size_t grainsize )`
: Returns: range over an entire `concurrent_vector` that permits read-write access.

`const_range_type range( size_t grainsize ) const`
: Returns: range over an entire `concurrent_vector` that permits read-only access.

Capacity

`size_type size() const`
: Returns: number of elements in the vector. The result may include elements that are under construction by concurrent calls to the `grow_by` or `grow_to_at_least` method.

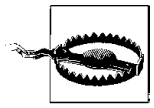
`bool empty( ) const`
: Returns: `size( )==0`.

`size_type capacity( ) const`
: Returns: maximum size the vector can grow to without allocating more memory.

`void reserve( size_type n )`
: Returns: reserve space for at least `n` elements.

: Throws: `std::length_error` if `n>max_size( )`.

`size_type max_size( ) const`
: Returns: highest size vector that might be representable.



`max_size( )` is a requirement on STL containers. The C++ standard defines it as "`size( )` of the largest possible container," which is vague. Threading Building Blocks does what most STL libraries do and computes `max_size( )` on the *very* optimistic assumption that the machine's address space is fully populated with usable memory, and that the container is the only object in this space. In practice, if you allocate a container of size `max_size( )`, you will most likely get an out-of-memory exception. In the freak event that you succeed, the next request will surely run out of memory.

Iterators

The template class `concurrent_vector<T>` supports random access iterators as defined in Section 24.1.4 of the ISO C++ standard. Unlike an `std::vector`, the iterators are not raw pointers. A `concurrent_vector<T>` meets the reversible container requirements in Table 66 of the ISO C++ standard.

`iterator begin( )`
: Returns: iterator pointing to the beginning of the vector.

`iterator end( )`
: Returns: iterator pointing to the end of the vector.

`const_iterator begin( ) const`
: Returns: `const_iterator` pointing to the beginning of the vector.

`const_iterator end( ) const`
: Returns: `const_iterator` pointing to the end of the vector.

`iterator rbegin( )`
: Returns: `const_reverse_iterator(end( ))`.

`iterator rend( )`
: Returns: `const_reverse_iterator(begin( ))`.

`const_reverse_iterator rbegin( ) const`
: Returns: `const_reverse_iterator(end( ))`.

`const_ reverse_iterator rend( ) const`
: Returns: `const_reverse_iterator(begin( ))`.

concurrent_hash_map

A concurrent_hash_map<Key,T,HashCompare> is a hash table that permits concurrent accesses. The table is a map from a key to a type T. The HashCompare traits type defines how to hash a key and how to compare two keys.

Example 5-4 builds a concurrent_hash_map in which the keys are strings and the corresponding data is the number of times each string occurs in the array Data.

Example 5-4. Concurrent hash map

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

using namespace tbb;
using namespace std;

// Structure that defines hashing and comparison operations for user's type.
struct MyHashCompare {
    static size_t hash( const 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 string& x, const string& y ) {
        return x==y;
    }
};

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

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

const size_t N = 1000000;

string Data[N];
```

Example 5-4. Concurrent hash map (continued)

```
void CountOccurrences( ) {
    // Construct empty table.
    StringTable table;

    // Put occurrences into the table
      parallel_for( blocked_range<string*>( Data, Data+N, 100 ),
                                            Tally(table) );

    // Display the occurrences
    for( StringTable::iterator i=table.begin(); i!=table.end( ); ++i )
        printf("%s %d\n",i->first.c_str( ),i->second);
}
```

A `concurrent_hash_map` acts as a container of elements of type `std::pair<const Key,T>`. Typically, when accessing a container element, you are interested in either updating it or reading it. The template class `concurrent_hash_map` supports these two operations with the `accessor` and `const_accessor` classes, respectively, which 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. Therefore, multiple `const_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.

The `find` and `insert` methods take an `accessor` or `const_accessor` as an argument. The choice tells `concurrent_hash_map` whether you are asking for update or read-only access, respectively. 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 the `release` method.

Example 5-5 is a rework of the loop body that uses `release` instead of waiting for the destruction of the `accessor` for the lock to be released.

Example 5-5. Use of release method

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

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 the key.

More on HashCompare

In general, the definition of `HashCompare` must provide two signatures:

- A `hash` method that maps a `Key` to a `size_t`
- An `equal` method that determines whether two keys are equal

The signatures fall naturally in a single class because *if two keys are equal, they must hash to the same value*. Otherwise, the hash table might not work. You could trivially meet this requirement by always hashing to 0, but that would cause tremendous inefficiency. Ideally, each key should hash to a different value, or at least the probability of two distinct keys hashing to the same value should be kept low.

The methods of `HashCompare` should be static unless you need to have them behave differently for different instances. If so, construct the `concurrent_hash_map` using the constructor that takes a `HashCompare` as a parameter. Example 5-6 is a variation on an earlier example that uses instance-dependent methods. The instance performs either case-sensitive or case-insensitive hashing and comparison, depending upon an internal flag, `ignore_case`.

Example 5-6. Hash compare

```
// Structure that defines hashing and comparison operations
class VariantHashCompare {
    // If true, then case of letters is ignored.
    bool ignore_case;
public:
    size_t hash( const string& x ) {
        size_t h = 0;
        for( const char* s = x.c_str( ); *s; s++ )
            h = (h*17)^*(ignore_case?tolower(*s):*s);
        return h;
    }
    // True if strings are equal
    bool equal( const string& x, const string& y ) {
        if( ignore_case )
            strcasecmp( x.c_str(), y.c_str( ) )==0;
        else
            return x==y;
    }
    VariantHashCompare( bool ignore_case_ ) : ignore_case( ) {}
};

typedef concurrent_hash_map<string,int, VariantHashCompare>
        VariantStringTable;
VariantStringTable CaseSensitiveTable(VariantHashCompare(false));
VariantStringTable CaseInsensitiveTable(VariantHashCompare(true));
```

The directory *examples/concurrent_hash_map/count_strings* contains a complete example that uses `concurrent_hash_map` to enable multiple processors to cooperatively build a histogram.

concurrent_hash_map<Key,T,HashCompare> Template Class

Template class for associative container with concurrent access.

```
#include "tbb/concurrent_hash_map.h"

template<typename Key, typename T, typename HashCompare> class concurrent_hash_map;
```

Description

A concurrent_hash_map maps keys to values in a way that permits multiple threads to concurrently access values. The keys are unordered. The interface resembles typical STL associative containers, but with some differences that are critical to supporting concurrent access.

The Key and T types must model the CopyConstructible Concept.

The HashCompare type specifies how keys are hashed and compared for equality. It must model the HashCompare Concept defined in Table 5-3.

Table 5-3. HashCompare Concept

Pseudosignature	Semantics
`HashCompare::HashCompare( const HashCompare & )`	Copy constructor
`HashCompare::~HashCompare ( )`	Destructor
`bool HashCompare::equal( const Key& j, const Key& k ) const`	`True` if keys are equal
`size_t HashCompare::hash( const Key& k )`	Hash code for key

As for most hash tables, if two keys are equal, they must hash to the same hash code. That is, for a given HashCompare h and any two keys j and k, the following assertion must hold: `!h.equal(j,k) || h.hash(j)==h.hash(k)`. The importance of this property is the reason that concurrent_hash_map places key equality and hashing in a single object instead of keeping them as separate objects.

Members

```
namespace tbb {
    template<typename Key, typename T, typename HashCompare>
    class concurrent_hash_map {
    public:
        // types
        typedef Key key_type;
        typedef T mapped_type;
        typedef std::pair<const Key,T> value_type;
        typedef size_t size_type;
        typedef ptrdiff_t difference_type;

        // whole-table operations
        concurrent_hash_map( );
        concurrent_hash_map( const concurrent_hash_map& );
        ~concurrent_hash_map( );
        concurrent_hash_map operator=( const concurrent_hash_map& );
        void clear( );
```

```
        // concurrent access
        class const_accessor;
        class accessor;

        // concurrent operations on a table
        bool find( const_accessor& result, const Key& key ) const;
        bool find( accessor& result, const Key& key );
        bool insert( const_accessor& result, const Key& key );
        bool insert( accessor& result, const Key& key );
        bool erase( const Key& key );

        // parallel iteration
        typedef implementation defined range_type;
        typedef implementation defined const_range_type;
        range_type range( size_t grainsize );
        const_range_type range( size_t grainsize ) const;

        // Capacity
        size_type size( ) const;
        bool empty( ) const;
        size_type max_size( ) const;

        // Iterators
        typedef implementation defined iterator;
        typedef implementation defined const_iterator;
        iterator begin( );
        iterator end( );
        const_iterator begin( ) const;
        const_iterator end( ) const;
    };
}
```

Whole-Table Operations

These operations affect an entire table. Do not concurrently invoke them on the same table.

`concurrent_hash_map( )`
: Effects: constructs an empty table.

`concurrent_hash_map( const concurrent_hash_map& table )`
: Effects: copies a table. The table being copied may have map operations running on it concurrently.

`~concurrent_hash_map( )`
: Effects: removes all items from the table and destroys it. This method is not safe to execute concurrently with other methods on the same `concurrent_hash_map`.

`concurrent_hash_map& operator= ( concurrent_hash_map& source )`

Effects: if the source table and destination table (`this`) are distinct, clear the destination table and copy all key-value pairs from the source table to the destination table. Otherwise, do nothing.

Returns: reference to the destination table.

`void clear( )`

Effects: erases all key-value pairs from the table.

Concurrent Access

The member classes `const_accessor` and `accessor` are called *accessors*. Accessors allow multiple threads to concurrently access pairs in a shared `concurrent_hash_map`. An accessor acts as a smart pointer to a pair in a `concurrent_hash_map`. It holds an implicit lock on a pair until the instance is destroyed or the `release` method is called on the accessor.

The `const_accessor` and `accessor` classes differ in the kind of access they permit, as shown in Table 5-4.

Table 5-4. Differences between const_accessor and accessor

Class	value_type	Implied lock on pair
`const_accessor`	`const std::pair<const Key,T>`	Reader lock: permits shared access with other readers
`accessor`	`std::pair<const Key,T>`	Writer lock: blocks access by other threads

Accessors cannot be assigned or copy-constructed because allowing that would greatly complicate the locking semantics.

const_accessor

Provides read-only access to a pair in a `concurrent_hash_map`.

```
#include "tbb/concurrent_hash_map.h"

template<typename Key, typename T, typename HashCompare> class concurrent_hash_
map<Key,T,HashCompare>::const_accessor;
```

Description

A `const_accessor` permits read-only access to a key-value pair in a `concurrent_hash_map`.

Members

```
namespace tbb {
    template<typename Key, typename T, typename HashCompare>
    class concurrent_hash_map<Key,T,HashCompare>::const_accessor {
    public:
        // types
        typedef const std::pair<const Key,T> value_type;
```

```
        // construction and destruction
        const_accessor( );
        ~const_accessor( );

        // inspection
        bool empty( ) const;
        const value_type& operator*( ) const;
        const value_type* operator->( ) const;

        // early release
        void release( );
    };
}
```

`bool empty( ) const`
: Returns: `true` if the instance points to nothing; `false` if the instance points to a key-value pair.

`void release( )`
: Effects: if the instance contains a key-value pair, releases the implied lock on the pair and sets the instance to point to nothing. Otherwise, does nothing.

`const value_type& operator*( ) const`
: Effects: raises an assertion failure if `empty( )` is `true` and `TBB_DO_ASSERT` is defined as nonzero.

  Returns: `const` reference to a key-value pair.

`const value_type* operator->( ) const`
: Returns: `&operator*( )`.

`const_accessor( )`
: Effects: constructs a `const_accessor` that points to nothing.

`~const_accessor`
: Effects: if pointing to a key-value pair, releases the implied lock on the pair.

accessor class

Class that provides read and write access to a pair in a `concurrent_hash_map`.

```
#include "tbb/concurrent_hash_map.h"

template<typename Key, typename T, typename HashCompare>

class concurrent_hash_map<Key,T,HashCompare>::accessor;
```

Description

An accessor permits read and write access to a key-value pair in a `concurrent_hash_map`. It is derived from a `const_accessor`, and thus can be implicitly cast to a `const_accessor`.

Members

```
namespace tbb {
    template<typename Key, typename T, typename HashCompare>
    class concurrent_hash_map<Key,T,HashCompare>::accessor:
        concurrent_hash_map<Key,T,HashCompare>::const_accessor {
    public:
```

```
            typedef std::pair<const Key,T> value_type;
            value_type& operator*( ) const;
            value_type* operator->( ) const;
        };
    }
```

`value_type& operator*( ) const`

Effects: raises an assertion failure if `empty( )` is `true` and `TBB_DO_ASSERT` is defined as nonzero.

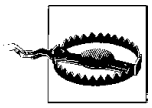
Returns: reference to a key-value pair.

`value_type* operator->( ) const`

Returns: `&operator*( )`.

Concurrent Operations: find, insert, erase

The operations `find`, `insert`, and `erase` are the only operations that may be concurrently invoked on the same `concurrent_hash_map`. These operations search the table for a key-value pair that matches a given key. The `find` and `insert` methods each have two variants. One takes a `const_accessor` argument and provides read-only access to the desired key-value pair. The other takes an `accessor` argument and provides write access.



If the non-`const` variant succeeds in finding the key, the consequent write access blocks any other thread from accessing the key until the accessor object is destroyed. Where possible, use the `const` variant to improve concurrency.

The result of the map operation is true if the operation succeeds.

`bool find( const_accessor& result, const Key& key ) const`

Effects: searches a table for a pair with the given key. If the key is found, provides read-only access to the matching pair.

Returns: `true` if the key is found; `false` if the key is not found.

`bool find( accessor& result, const Key& key )`

Effects: searches a table for a pair with the given key. If the key is found, provides write access to the matching pair.

Returns: `true` if the key is found; `false` if the key is not found.

`bool insert( const_accessor& result, const Key& key )`

Effects: searches a table for a pair with the given key. If not present, inserts a new pair into the table. The new pair is initialized with `pair(key,T( ))`. Provides read-only access to the matching pair.

Returns: `true` if a new pair is inserted; `false` if the key is already in the map.

`bool insert( accessor& result, const Key& key )`

Effects: searches a table for a pair with the given key. If not present, inserts a new pair into the table. The new pair is initialized with `pair(key,T())`. Provides write access to the matching pair.

Returns: `true` if a new pair is inserted; `false` if the key is already in the map.

`bool erase( const Key& key )`

Effects: searches a table for a pair with the given key. Removes the matching pair if it exists.

Returns: `true` if the pair is removed; `false` if the key is not in the map.

Parallel Iteration

The types `const_range_type` and `range_type` model the Range Concept and provide methods to access the bounds of the range, as shown in Table 5-5. The types differ only in that the bounds for a `const_range_type` are of type `const_iterator`, whereas the bounds for a `range_type` are of type `iterator`.

Use the range types in conjunction with `parallel_for`, `parallel_reduce`, and `parallel_scan` to iterate over pairs in a `concurrent_hash_map`.

Table 5-5. Concept for concurrent_hash_map range R

Pseudosignature	Semantics
`R::iterator R::begin() const`	First item in range
`R::iterator R::end() const`	One past last item in range

`const_range_type range( size_t grainsize ) const`

Effects: constructs a `const_range_type` representing all keys in the table. The `grainsize` parameter is in units of hash table slots. Each slot typically has only one key-value pair.

Returns: a `const_range_type` object for the table.

`range_type range( size_t grainsize )`

Returns: like `const_range_type`, but returns a `range_type` object for the table.

Capacity

`size_type size() const`

Returns: number of key-value pairs in the table.

This method takes constant time, but it is slower than the corresponding method in most STL containers.

`bool empty( ) const`

Returns: `size( )==0`.

This method takes constant time, but it is slower than the corresponding method in most STL containers.

`size_type max_size( ) const`

Returns: inclusive upper bound on the number of key-value pairs that the table can hold.

Iterators

The template class `concurrent_hash_map` supports forward iterators; that is, iterators that can advance only forward across the table. Reverse iterators are not supported. All elements will be visited in a walk from `begin` to `end`, but there is no guarantee on the order of the walk.

`iterator begin( )`

Returns: iterator pointing to the beginning of the key-value sequence.

`iterator end( )`

Returns: iterator pointing to the end of the key-value sequence.

`const_iterator begin( ) const`

Returns: `const_iterator` pointing to the beginning of the key-value sequence.

`const_iterator end( ) const`

Returns: `const_iterator` pointing to the end of the key-value sequence.