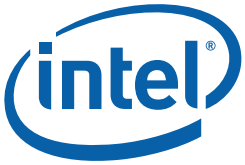


Intel® Threading Building Blocks

Tutorial

Document Number 319872-001US

Revision: 1.13

World Wide Web: http://www.intel.com

Disclaimer and Legal Information

Revision History

Version	Version Information	Date
1.7	Added discussion of `affinity_partitioner` and cache affinity. Deleted Partitioner as a concept. Changed `parallel_while` to `parallel_do`. Add `recursive_mutex`.	2007-Dec-19
1.8	Update copyright to 2008.	2008-Jan-18
1.9	Change pipeline filter constructors to use `filter::mode`.	2008-Mar-31
1.10	Add vc9. Add local_sum to `parallel_reduce` example.	2008-May-1
1.11	Revise discussion of `parallel_reduce`. Fix `set_ref_count` calls in scheduler bypass and recycling examples.	2008-Aug-21
1.12	Update discussion of serial pipeline filters for distinction between `serial_in_order` and `serial_out_of_order`.	2008-Oct-28
1.13	Correct descriptions of class pipeline. Add sample compilation commands to Section 3.1. Update copyright to 2009. Mention `null_mutex`.	2009-Jan-22

Contents

1 Introduction

This tutorial teaches you how to use Intel® Threading Building Blocks, a library that helps you leverage multi-core performance without having to be a threading expert. The subject may seem daunting at first, but usually you only need to know a few key points to improve your code for multi-core processors. For example, you can successfully thread some programs by reading only up to Section 3.4 of this document. As your expertise grows, you may want to dive into more complex subjects that are covered in advanced sections.

1.1 Document Structure

This tutorial is organized to cover the high-level features first, then the low-level features, and finally the mid-level task scheduler. This Using Intel® Threading Building Blocks Tutorial contains the following sections:

Table 1 Document Organization

Section	Description
Chapter 1	Introduces the document.
Chapter 2	Describes how to install the library.
Chapters 3-4	Describe templates for parallel loops.
Chapter 5	Describes templates for concurrent containers.
Chapters 6-9	Describe low-level features for mutual exclusion, atomic operations, timing, and memory allocation.
Chapter 10	Explains the task scheduler.

1.2 Benefits

There are a variety of approaches to parallel programming, ranging from using platform-dependent threading primitives to exotic new languages. The advantage of Intel® 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++. The library differs from typical threading packages in the following ways:

- **Intel® Threading Building Blocks enables you to specify *tasks* instead of threads**. Most threading packages require you to specify threads. Programming

directly in terms of threads can be tedious and lead to inefficient programs, because threads are low-level, heavy constructs that are close to the hardware. Direct programming with threads forces you to efficiently map logical tasks onto threads. In contrast, the Intel® Threading Building Blocks run-time library automatically schedules tasks onto threads in a way that makes efficient use of processor resources.

- **Intel® 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, Intel® Threading Building Blocks focuses on the particular goal of parallelizing computationally intensive work, delivering higher-level, simpler solutions.
- **Intel® Threading Building Blocks is *compatible* with other threading packages.** Because the library is not designed to address all threading problems, it can coexist seamlessly with other threading packages.
- **Intel® Threading Building Blocks emphasizes *scalable, data parallel programming*.** Breaking a program up into separate functional blocks, and assigning a separate thread to each block is a solution that typically does not scale well since typically the number of functional blocks is fixed. In contrast, Intel® Threading Building Blocks emphasizes *data-parallel* programming, enabling multiple threads to work on different parts of a collection. Data-parallel programming scales well to larger numbers of processors by dividing the collection into smaller pieces. With data-parallel programming, program performance increases as you add processors.
- **Intel® Threading Building Blocks relies on *generic programming*.** Traditional libraries specify interfaces in terms of specific types or base classes. Instead, Intel® Threading Building Blocks uses generic programming. The essence of generic programming is writing 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 `sort` that sorts a sequence abstractly defined in terms of iterators on the sequence. The requirements on the iterators are:

  - Provide random access
  - The expression `*i<*j` is true if the item pointed to by iterator `i` should precede the item pointed to by iterator `j`, and false otherwise.
  - The expression `swap(*i,*j)` swaps two elements.

  Specification in terms of requirements on types enables the template to sort many different representations of sequences, such as vectors and deques. Similarly, the Intel® Threading Building Blocks templates specify requirements on types, not particular types, and thus adapt to different data representations. Generic programming enables Intel® Threading Building Blocks to deliver high performance algorithms with broad applicability.

2 Package Contents

Intel® Threading Building Blocks includes dynamic shared library files, header files, and code examples for Windows*, Linux*, and Mac OS* X systems that you can compile and run as described in this chapter.

2.1 Debug Versus Release Libraries

Intel® Threading Building Blocks includes dynamic shared libraries that come in debug and release versions, as described in Table 2.

Table 2: Dynamic Shared Libraries Included in Intel® Threading Building Blocks

Library (*.dll, lib*.so, or lib*.dylib)	Description	When to Use
`tbb_debug` `tbbmalloc_debug`	These versions have extensive internal checking for incorrect use of the library.	Use with code that is compiled with the macro TBB_DO_ASSERT set to 1.
`tbb` `tbbmalloc`	These versions deliver top performance. They eliminate most checking for correct use of the library.	Use with code compiled with TBB_DO_ASSERT undefined or set to zero.

TIP: Test your programs with the debug versions of the libraries first, to assure that you are using the library correctly. With the release versions, incorrect usage may result in unpredictable program behavior.

All versions of the libraries support Intel® Thread Checker and Intel® Thread Profiler. The debug versions always have full support enabled. The release version requires compiling code with the macro `TBB_DO_THREADING_TOOLS` set to 1 for full support.

CAUTION: The instrumentation support for Intel® Thread Checker becomes live after the first initialization of the task library (3.1). If the library components are used before this initialization occurs, Intel® Thread Checker may falsely report race conditions that are not really races.

2.2 Scalable Memory Allocator

Both the debug and release versions of Intel® Threading Building Blocks are divided into two dynamic shared libraries, one with general support and the other with a scalable memory allocator. The latter is distinguished by `malloc` in its name. For example, the release versions for Windows* system are `tbb.dll` and `tbbmalloc.dll` respectively. Applications may choose to use only the general library, or only the scalable memory allocator, or both. Section 9.1 describes which parts of Intel® Threading Building Blocks depend upon which libraries.

2.3 Windows* Systems

The default installation location for Windows* systems depends on the host architecture, as described in Table 3.

Table 3: Intel® Threading Building Blocks Default Installation Location

Host	Default Installation Location
Intel® IA-32 processor	`C:\Program Files\Intel\TBB\<version>\`
Intel® Extended Memory 64 Technology (Intel® EM64T) processor	`C:\Program Files (x86)\Intel\TBB\<version>\`

Table 4 describes the subdirectories' contents for Windows* systems.

Table 4: Intel® Threading Building Blocks Subdirectories on Windows

<table>
<tr><th>Item</th><th>Location</th><th>Environment Variable</th></tr>
<tr><td>Include files</td><td><code>include\tbb*.h</code></td><td>INCLUDE</td></tr>
<tr><td>.lib files</td><td><code><arch>\vc<vcversion>\lib\<lib>.lib</code></td><td>LIB</td></tr>
<tr><td>.dll files</td><td><code><arch>\vc<vcversion>\bin\<lib><malloc>.dll</code>
where <arch> is:
<table>
<tr><th><arch></th><th>Processor</th></tr>
<tr><td>ia32</td><td>Intel® IA-32 processors</td></tr>
<tr><td>em64t</td><td>Intel® EM64T processors</td></tr>
</table>
and <vcversion> is:
<table>
<tr><th><vcversion></th><th>Environment</th></tr>
<tr><td>7.1</td><td>Microsoft Visual Studio .NET* 2003</td></tr>
<tr><td>8</td><td>Microsoft Visual Studio* 2005</td></tr>
</table>
</td><td>PATH</td></tr>
</table>

Item	Location	Environment Variable
	9 — Microsoft Visual Studio* 2008 and <*lib*> is **<*lib*>** / **Version**: `tbb` — Release version; `tbb_debug` — Debug version and <*malloc*> is **<*malloc*>** / **Version**: (none) — General library; `malloc` — Memory allocator	
Examples	`examples\<class>\*\.`	
Microsoft Visual Studio* Solution File for Example	`examples\<class>\*\vc<vcversion>\*.sln` where *class* describes the class being demonstrated and *vcversion* is as described for .dll files.	

The last column shows which environment variables are used by the Microsoft* or Intel compilers to find these subdirectories.

CAUTION: Ensure that the relevant product directories are mentioned by the environment variables; otherwise the compiler might not find the required files.

CAUTION: Windows* run-time libraries come in thread-safe and thread-unsafe forms. Using non-thread-safe versions with Intel® Threading Building Blocks may cause undefined results. When using Intel® Threading Building Blocks, be sure to link with the thread-safe versions. Table 5 shows the required options when using `cl` or `icl`:

Table 5: Compiler Options for Linking with Thread-safe Versions of C/C++ Run-time

Option	Description
`/MDd`	Debug version of thread-safe run-time
`/MD`	Release version of thread-safe run-time

Not using one of these options causes the library to report an error during compilation.

2.3.1 Microsoft Visual Studio* Code Samples

To run one of the solution files in `examples\*\*\.`:

1. Open up the `vc7.1` directory (if using Microsoft Visual Studio .NET* 2003), `vc8` directory (if using Microsoft Visual Studio* 2005), or `vc9` directory (if using Microsoft Visual Studio* 2008).
2. Double-click the `.sln` file.
3. In Microsoft Visual Studio*, press **ctrl-F5** to compile and runs the example. Use **Ctrl-F5**, not **Shift-F5**, so that you can inspect the console window after the example finishes.

The Microsoft Visual Studio* solution files for the examples require that an environment variable specify where the library is installed. The installer sets this variable.

The makefiles for the examples require that `INCLUDE`, `LIB`, and `PATH` be set as indicated in Table 4. The recommended way to set `INCLUDE`, `LIB`, and `PATH` is to do one of the following:

TIP: Check the **Register environment variables**" box when running the installer.

Otherwise, go to the library's `<arch>\vc<vcversion>\bin\` directory and run the batch file `tbbvars.bat` from there, where `<arch>` and `<vcversion>` are described in Table 4.

2.3.2 Integration Plug-In for Microsoft Visual Studio* Projects

The plug-in simplifies integration of Intel® TBB into Microsoft Visual Studio* projects. It can be downloaded from http://threadingbuildingblocks.org → Downloads → Extras. The plug-in enables you to quickly add the following to Microsoft Visual C++* projects:

- the path to the TBB header files
- the path to the TBB libraries
- the specific TBB libraries to link with

The plug-in works with C++ projects created in Microsoft* Visual Studio* 2003, 2005 and 2008 (except Express editions).

To use this functionality unzip the downloaded package `msvs_plugin.zip`, open `it`, and follow the instructions in `README.txt` to install it.

TIP: To check that installation succeeded, select “Tools” → “Add-in Manager” in the main Microsoft Visual Studio* menu and check that the table lists the TBB integration plug-in. Also, the list of installed products in the Microsoft Visual Studio* “Help” → “About dialog” should mention "TBB Integration".

To enable Intel® TBB for a C++ project, in Microsoft Visual Studio* Solution Explorer right-click the project item and open the "TBB Integration" sub-menu, as shown in Figure 1.

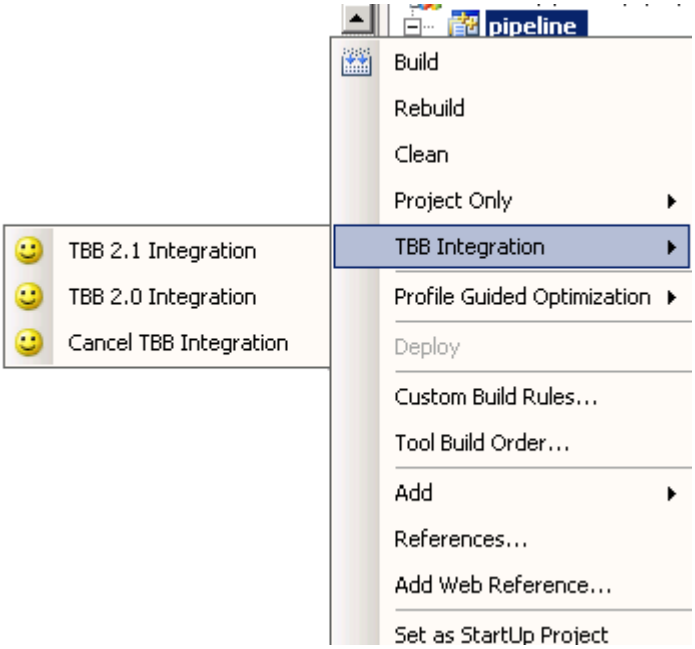


Figure 1: Integration Plug-In

The sub-menu contains:

- A list of installed Intel® TBB versions. Select a version to update your project to use that version.
- "Cancel TBB Integration". Selects this to delete all TBB-specific settings from your project.

2.4 Linux* Systems

On Linux* systems, the default installation location is `/opt/intel/tbb/<version>/`. Table 6 describes the subdirectories.

Table 6: Intel® Threading Building Blocks Subdirectories on Linux* Systems

<table>
<tr><th>Item</th><th>Location</th><th>Environment Variable</th></tr>
<tr><td>Include files</td><td>include/tbb/*.h</td><td>CPATH</td></tr>
<tr><td>Shared libraries</td><td><arch>/cc<gccversion>_libc<glibcversion>_kernel<kernel version>/lib/lib<lib><malloc>.so
where
<table>
<tr><th><arch></th><th>Processor</th></tr>
<tr><td>ia32</td><td>Intel® IA-32 processors</td></tr>
<tr><td>em64t</td><td>Intel® EM64T processors</td></tr>
<tr><td>itanium</td><td>Intel® Itanium processors</td></tr>
</table>
<table>
<tr><th><*version> strings</th><th>Linux configuration</th></tr>
<tr><td><gccversion></td><td>gcc version number</td></tr>
<tr><td><glibcversion></td><td>glibc.so version number</td></tr>
<tr><td><kernelversion></td><td>Linux kernel version number</td></tr>
</table>
<table>
<tr><th><lib></th><th>Version</th></tr>
<tr><td>tbb</td><td>Release version</td></tr>
<tr><td>tbb_debug</td><td>Debug version</td></tr>
</table>
<table>
<tr><th><malloc></th><th>Version</th></tr>
<tr><td>(none)</td><td>General library</td></tr>
<tr><td>malloc</td><td>Memory allocator</td></tr>
</table></td><td>LIBRARY_PATH
LD_LIBRARY_PATH</td></tr>
<tr><td>Examples</td><td>examples/<class>/*/.</td><td></td></tr>
<tr><td>GNU Makefile for example</td><td>examples/<class>/*/linux/Makefile
where class describes the class being demonstrated.</td><td></td></tr>
</table>

2.5 Mac OS* X Systems

For Mac OS* X systems, the default installation location for the library is `/Library/Frameworks/Intel_TBB.framework/Versions/<version>/`. Table 7 describes the subdirectories.

Table 7: Intel® Threading Building Blocks Subdirectories on Mac OS* X Systems

<table>
<tr><th>Item</th><th>Location</th><th>Environment Variable</th></tr>
<tr><td>Include files</td><td>include/tbb/*.h</td><td>CPATH</td></tr>
<tr><td>Shared libraries</td><td>ia32/cc<<i>gccversion</i>>_os<<i>osversion</i>>/lib/<<i>lib</i>><<i>malloc</i>>.dylib
where:
<table>
<tr><th><*<i>version</i>> string</th><th>OS/X configuration</th></tr>
<tr><td><<i>gccversion</i>></td><td>gcc version number</td></tr>
<tr><td><<i>osversion</i>></td><td>Mac OS* X version number</td></tr>
</table>
<table>
<tr><th><<i>lib</i>></th><th><i>Version</i></th></tr>
<tr><td>libtbb</td><td>Release version</td></tr>
<tr><td>libtbb_debug</td><td>Debug version</td></tr>
</table>
<table>
<tr><th><<i>malloc</i>></th><th>Version</th></tr>
<tr><td>(none)</td><td>General library</td></tr>
<tr><td>malloc</td><td>Memory allocator</td></tr>
</table></td><td>LIBRARY_PATH
DYLD_LIBRARY_PATH</td></tr>
<tr><td>Examples</td><td>examples/<<i>class</i>>/*/.</td><td></td></tr>
<tr><td>GNU Makefile for example</td><td>examples/<<i>class</i>>/*/mac/Makefile
where <i>class</i> describes the class being demonstrated.</td><td></td></tr>
</table>

3 Parallelizing Simple Loops

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 simple loops.

3.1 Initializing and Terminating the Library

Any thread that uses an algorithm template from the library or the task scheduler must have an initialized `tbb::task_scheduler_init` object.

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

A thread may have more than one of these objects initialized at a time. The objects indicate when the task scheduler is needed. 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 termination. Declaring a `task_scheduler_init` in `main()`, as follows, both starts and shuts down the scheduler:

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

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

When compiling TBB programs, be sure to link in the TBB shared library, otherwise undefined references will occur. Table 8 shows compilation commands that use the debug version of the library. Remove the "_debug" portion to link against the production version of the library. Section 2.1 explains the difference. See `doc/Getting_Started.pdf` for other command line possibilities. Section 9.1 describes when the memory allocator library should be linked in explicitly.

Table 8: Sample command lines for simple debug builds

Windows* Systems	icl /MD example.cpp tbb_debug.dll
Linux* Systems	icc example.cpp tbb_debug.lib
Mac OS* X Systems	icc example.cpp tbb_debug.dylib

Appendix B explains how to construct `task_scheduler_init` objects if your program creates threads itself using another interface.

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.

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 same as not specifying the parameter at all. It exists for sake of the method `task_scheduler_init::initialize`.
- The value `task_scheduler_init::deferred`, which defers the initialization until method `task_scheduler_init::initialize(n)` 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 argument should be specified only when doing scaling studies during development. Omit the parameter, or use `task_scheduler_init::automatic`, for production code.

The parameter is ignored if another `task_scheduler_init` is active. That is, disagreements in the number of threads are resolved in favor of the first `task_scheduler_init` to specify a number of threads. 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 threads. Hardware threads are a shared global resource. It is best to leave the decision of how many threads to use to the task scheduler.

TIP: 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.

There is a method `task_scheduler_init::terminate` for terminating the library early before the `task_scheduler_init` is destroyed. The following example defers the decision of the number of threads to use the scheduler, and terminates it early:

```
int main( int argc, char* argv[] ) {
    int nthread = strtol(argv[0],0,0);
    task_scheduler_init init(task_scheduler_init::deferred);
    if( nthread>=1 )
      init.initialize(nthread);
  ... code that uses task scheduler only if nthread>=1 ...
    if( nthread>=1 )
```

```
        init.terminate();
    return 0;
}
```

In the example above, you can omit the call to `terminate()`, because the destructor for `task_scheduler_init` checks if the `task_scheduler_init` was initialized, and if so, performs the termination.

TIP: The task scheduler is somewhat expensive to start up and shut down, put the `task_scheduler_init` in `main`, and do not try to create a scheduler every time you use a parallel algorithm template.

3.2 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. Here is the sequential code to do this:

```
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 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 an STL-style function object, called the *body* object, in which `operator()` processes a chunk. The following code declares the body object. The extra code required for Intel® Threading Building Blocks is shown in blue.

```
#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 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. You can define your own spaces as explained in section 3.4.

An instance of ApplyFoo needs member fields that remember all the local variables that were defined outside the original loop but 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. 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. If they do not, it is almost always the case (as usual in C++) that you must define *both* to be consistent.

Because the body object might be copied, its operator() should not modify the body. 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 const.

The example operator() loads my_a into a local variable a. Though 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.

Once you have the loop body written as a body object, invoke the template function parallel_for, as follows:

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

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

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 STL-style as a half-open interval [*begin*,*end*). The argument *grainsize* is explained in Section 3.2.2. The example uses the default grainsize of 1 because it is using auto_partitioner().

3.2.1 Automatic Grainsize

A parallel loop construct incurs overhead cost for every chunk of work that it schedules. If the chunks are too small, the overhead may exceed the useful work. The grainsize of a parallel loop specifies the number of iterations that is a “reasonable size” chunk to deal out to a processor. An auto_partitioner indicates that chunk sizes should be chosen automatically, depending upon load balancing needs. The

heuristic attempts to limit overheads while still providing ample opportunities for load balancing.

As with most heuristics, there are situations where the `auto_partitioner`'s guess might be suboptimal, and omitting the heuristic would yield better performance. Use `auto_partitioner` unless you need to tune the grainsize for machines of interest.

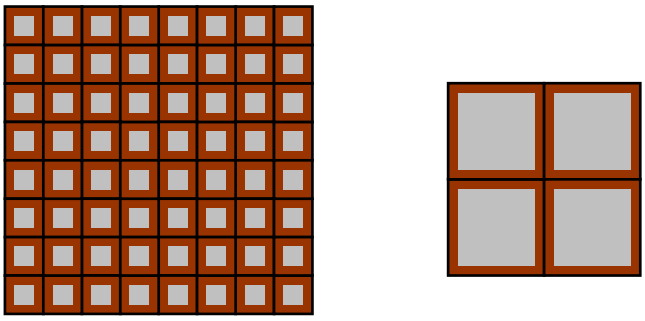
3.2.2 Explicit Grainsize

Class `blocked_range` allows a grainsize to be specified by the constructor `blocked_range<T>(begin,end,grainsize)`. The *grainsize* parameter is in units of iterations, and sets a minimum threshold for parallelization. Specifying `auto_partitioner` and `affinity_partitioner` may cause larger chunk sizes. Specifying a `simple_partitioner` or no partitioner at all guarantees that each chunk `is` no larger than the grainsize. Following is the previous example modified to use an explicit grainsize G.

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

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

Because of the impact of grainsize on parallel loops, it is worth reading the following material even if you rely on `auto_partitioner` and `affinity_partitioner` to choose the grainsize automatically.



Case A Case B

Figure 2: Packaging Overhead Versus Grainsize

Figure 2 illustrates the impact of grainsize by showing the useful work as the gray area inside a brown border that represents overhead. Both Case A and Case B have the same total gray area. Case A shows how too small a grainsize leads to a relatively high proportion of overhead. Case B shows how a large grainsize reduces this proportion, at the cost of reducing potential parallelism. The overhead as a fraction of

useful work depends upon the grainsize, not on the number of grains. Consider this relationship and not the total number of iterations or number of processors when setting a grainsize.

A rule of thumb is that `grainsize` iterations of `operator()` should take at least 10,000-100,000 instructions to execute. 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, because each loop iteration typically requires at least a few instructions per iteration.
2. Run your algorithm on one processor.
3. Start halving the *`grainsize`* parameter and see how much the algorithm slows down as the value decreases.

A slowdown of about 5-10% when running with a single thread is a good setting for most purposes. The drawback of setting a grainsize too high is that it can reduce parallelism. For example, if your grainsize 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 there is other parallelism available higher up in the call tree.

TIP: You do not have to set the grainsize too precisely.

Figure 3 shows the typical "bathtub curve" for execution time versus grainsize, based on the floating point `a[i]=b[i]*c` computation over a million indices. There is little work per iteration. The times were collected on a four-socket machine with eight hardware threads.

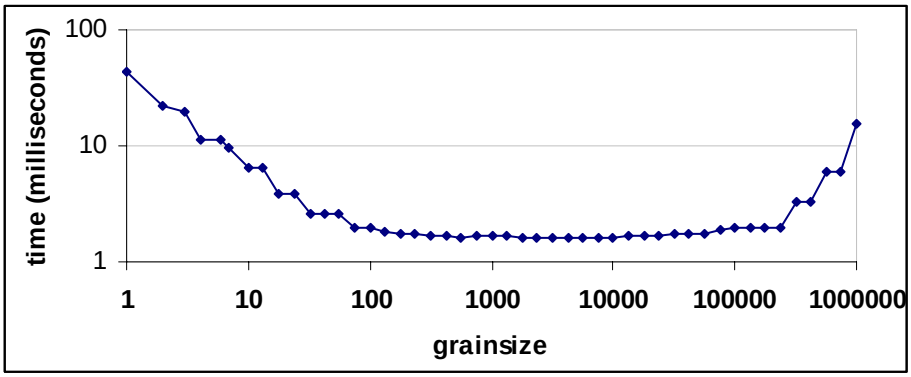


Figure 3: Wallclock Time Versus Grainsize

The scale is logarithmic. The downward slope on the left side indicates that with a grainsize of one, most of the overhead is parallel scheduling overhead, not useful work. An increase in grainsize brings a proportional decrease in parallel overhead. Then the curve flattens out because the parallel overhead becomes insignificant for a

sufficiently large grainsize. At the end on the right, the curve turns up because the chunks are so large that there are fewer chunks than available hardware threads. Notice that a grainsize over the wide range 100-100,000 works quite well.

TIP: 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.

3.2.3 Bandwidth and Cache Affinity

For a sufficiently simple function Foo, the examples might not show good 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.

An alternative to restructuring that works in some cases is `affinity_partitioner`. It not only automatically chooses the grainsize, but also optimizes for cache affinity. Using `affinity_partitioner` can significantly improve performance when:

- The computation does a few operations per data access.
- The data acted upon by the loop fits in cache.
- The loop, or a similar loop, is re-executed over the same data.
- There are more than two hardware threads available. If only two threads are available, the default scheduling in TBB usually provides sufficient cache affinity.

The following code shows how to use `affinity_partitioner`.

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

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

void TimeStepFoo( float a[], size_t n, int steps ) {
    for( int t=0; t<steps; ++t )
        ParallelApplyFoo( a, n );
}
```

In the example, the `affinity_partitioner` object ap lives between loop iterations. It remembers where iterations of the loop ran, so that each iteration can be handed to the same thread that executed it before. The example code gets the lifetime of the partitioner right by declaring the `affinity_partitioner` as a local static object. Another approach would be to declare it at a scope outside the iterative loop in `TimeStepFoo`, and hand it down the call chain to `parallel_for`.

If the data does not fit across the system's caches, there may be little benefit. Figure 4 contrasts the situations.

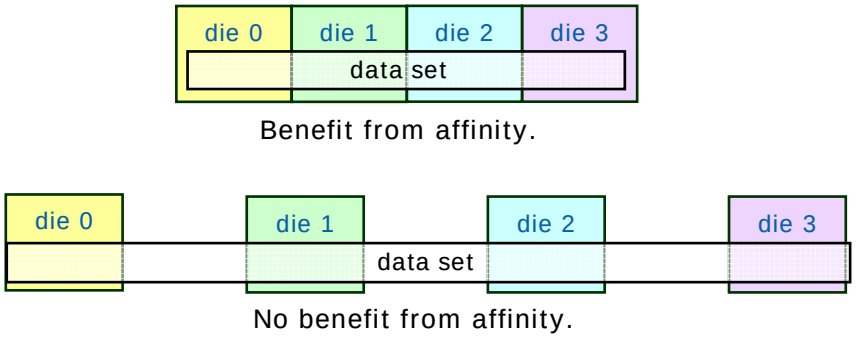


Figure 4: Benefit of Affinity Determined by Relative Size of Data Set and Cache

Figure 5 shows how parallel speedup might vary with the size of a data set. The computation for the example is `A[i]+=B[i]` for `i` in the range [0,N). It was chosen for dramatic effect. You are unlikely to see quite this much variation in your code. The graph shows not much improvement at the extremes. For small N, parallel scheduling overhead dominates, resulting in little speedup. For large N, the data set is too large to be carried in cache between loop invocations. The peak in the middle is the sweet spot for affinity. Hence `affinity_partitioner` should be considered a tool, not a cure-all, when there is a low ratio of computations to memory accesses.

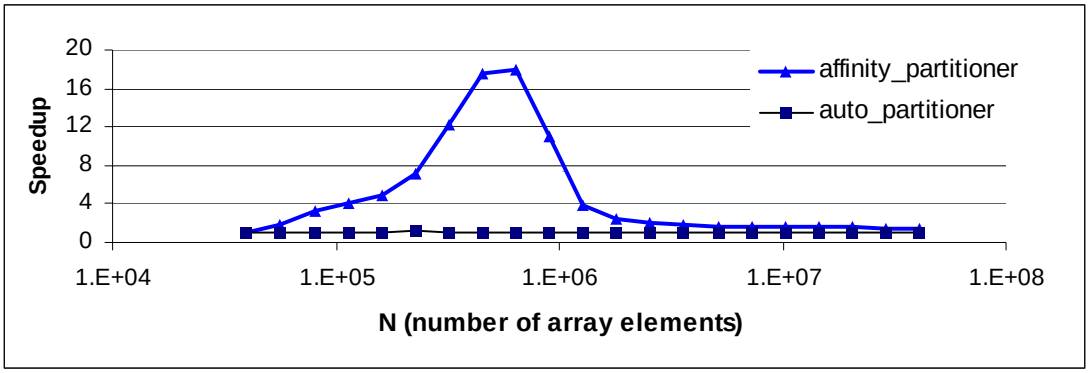


Figure 5: Improvement from Affinity Dependent on Array Size

3.2.4 Partitioner Summary

The parallel loop templates `parallel_for` (3.2) and `parallel_reduce` (3.3) take an optional *partitioner* argument, which specifies a strategy for executing the loop. Table 9 summarizes the three partitioners.

Table 9: Partitioners

Partitioner	Description
`simple_partitioner` (default)[1]	Chunk size = grainsize
`auto_partitioner`	Automatic chunk size.
`affinity_partitioner`	Automatic chunk size and cache affinity.

A `simple_partitioner` is used when no partitioner is specified, in which case the loop breaks the range down to its grainsize. In general, the `auto_partitioner` or `affinity_partitioner` should be used. However, `simple_partitioner` can be useful in the following situations:

- You need to guarantee that a subrange size for `operator()` does not exceed a limit. That might be advantageous, for example, if your `operator()` needs a temporary array proportional to the size of the range. With a limited subrange size, you can use an automatic variable for the array instead of having to use dynamic memory allocation.
- You want to tune to a specific machine.

3.3 parallel_reduce

A loop can do reduction, as in this summation:

```
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 follows:

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

The class SumFoo specifies details of the reduction, such as how to accumulate subsums and combine them. Here is the definition of class SumFoo:

```
class SumFoo {
    float* my_a;
```

[1] In retrospect, the default should be `auto_partitioner`. But `simple_partitioner` is the default for sake of backwards compatibility with TBB 1.0.

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

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

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

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

Note the differences with class ApplyFoo from Section 3.2. First, `operator()` is *not* `const`. This is because it must update `SumFoo::my_sum`. Second, `SumFoo` has a *splitting constructor* and a method join that must be present for `parallel_reduce` to work. 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.

TIP: In the example, the definition of `operator()` uses local temporary variables (`a`, `sum`, `end`) for scalar values accessed inside the loop. This technique can improve performance by making it obvious to the compiler that the values can be held in registers instead of memory. If the values are too large to fit in registers, or have their address taken in a way the compiler cannot track, the technique might not help. With a typical optimizing compiler, using local temporaries for only written variables (such as `sum` in the example) can suffice, because then the compiler can deduce that the loop does not write to any of the other locations, and hoist the other reads to outside the loop.

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 method `join` to accumulate the result of the subtask. The graph at the top of Figure 6 shows the split-join sequence that happens when a worker is available:

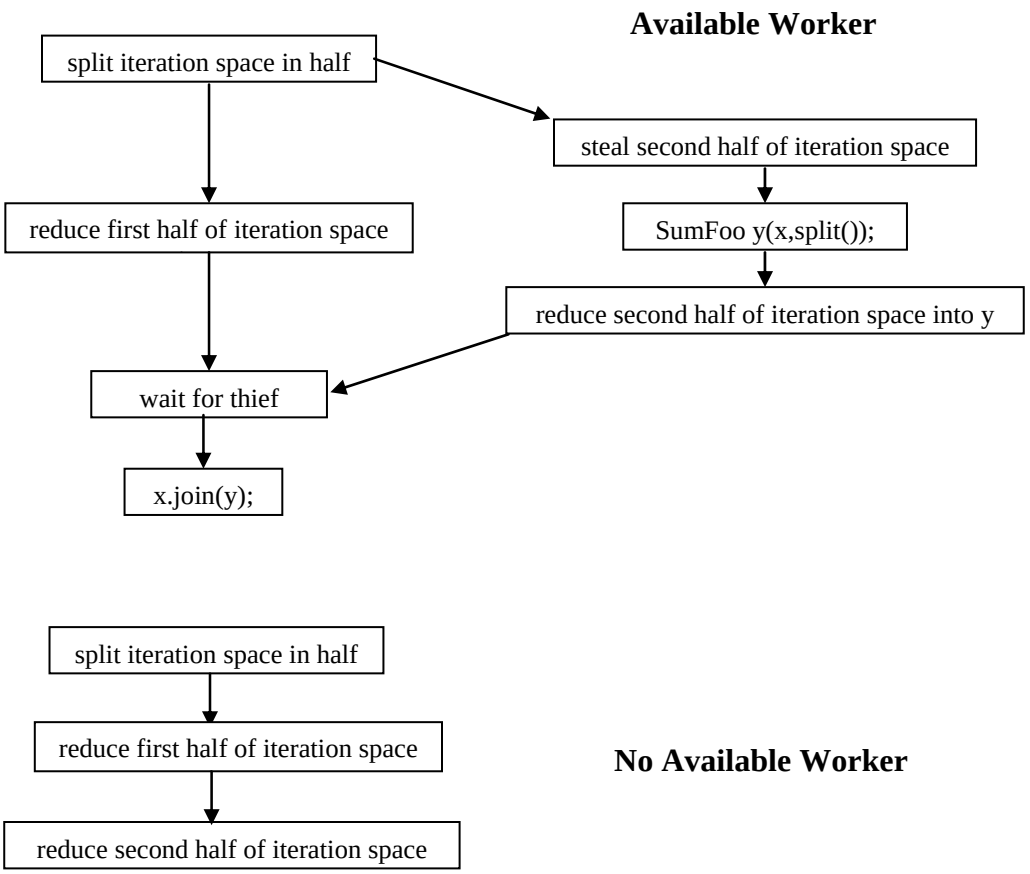


Figure 6: Graph of the Split-join Sequence

An arc in the Figure 6 indicates order in time. 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 creates y must be made thread safe with respect to x. So if the splitting constructor needs to increment a reference count shared with other objects, it should use an atomic increment.

If a worker is not available, the second half of the iteration is reduced using the same body object that reduced the first half. That is the reduction of the second half starts where reduction of the first half finished.

CAUTION: Because split/join are only used if workers are available, parallel_reduce does not necessarily execute in a tree-like fashion.

The rules for partitioners and grain sizes for `parallel_reduce` are the same as for `parallel_for`.

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

- Copy read-only information necessary to run the loop body.

- Initialize the reduction variable(s) to the identity element of the operation(s).

The join method should do the corresponding merge(s). You can do more than one reduction at the same time: you can gather the min and max with a single `parallel_reduce`.

NOTE: The reduction operation can be non-commutative. The example still works if floating-point addition is replaced by string concatenation.

3.3.1 Advanced Example

An example of a more advanced associative operation is to find the index where `Foo(i)` is minimized. A serial version might look like this:

```
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 carried information, and how to merge this information when iterations are spread across multiple threads. Also, the function object must record a pointer to `a` to provide context.

The following code shows the complete function object.

```
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<x.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` as shown below:

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

The directory `examples/parallel_reduce/primes` contains a prime number finder based on `parallel_reduce`.

3.4 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 situation. You can use Intel® Threading Building Blocks to define your own iteration space objects. The object must specify how it can be split into subspaces by providing two methods and a "splitting constructor". If your class is called `R`, the methods and constructor could be as follows:

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

The method `empty` should return true if the range is empty. The method `is_divisible` should return true if the range can be split into two non-empty subspaces, and such a split is worth the overhead. The splitting constructor should 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` roughly into two halves, and update `r` to be the first half, and let constructed object be the second half. The two halves should be non-empty. The parallel algorithm templates call the splitting constructor on `r` only if `r.is_divisible` is true.

The iteration space does not have to be linear. Look at `tbb/blocked_range2d.h` for an example of a range that is two-dimensional. 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.

3.4.1 Code Samples

The directory `examples/parallel_for/seismic` contains a simple seismic wave simulation based on `parallel_for` and `blocked_range`. The directory `examples/parallel_for/tachyon` contains a more complex example of a ray tracer based on `parallel_for` and `blocked_range2d`.

4 Parallelizing Complex Loops

You can successfully parallelize many applications using only the constructs in Chapter 3. However, some situations call for other parallel patterns. This section describes the support for some of these alternate patterns.

4.1 Cook Until Done: parallel_do

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_do`.

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, the items can be safely processed in parallel, and processing each item takes at least a few thousand instructions, you can use `parallel_do` to gain some parallelism.

For example, consider the following serial code:

```
void SerialApplyFooToList( const std::list<Item>& list ) {
    for( std::list<Item>::const iterator i=list.begin() i!=list.end();
++i )
        Foo(*i);
}
```

If `Foo` takes at least a few thousand instructions to run, you can get parallel speedup by converting the loop to use `parallel_do`. To do so, define an object with a const operator(). This is similar to a C++ function object from the C++ standard header `<functional>`, except that `operator()` must be `const`.

```
class ApplyFoo {
public:
    void operator()( Item& item ) const {
        Foo(item);
    }
};
```

The parallel form of `SerialApplyFooToList` is as follows:

```
void ParallelApplyFooToList( const std::list<Item>& list ) {
    parallel_do( list.begin(), list.end(), ApplyFoo() );
}
```

An invocation of `parallel_do` never causes two threads to act on an input iterator concurrently. Thus typical definitions of input iterators for sequential programs work

correctly. This convenience makes `parallel_do` unscalable, because the fetching of work is serial. But in many situations, you still get useful speedup over doing things sequentially.

There are two ways that `parallel_do` can acquire work scalably.

- The iterators can be random-access iterators.
- The body argument to `parallel_do`, if it takes a second argument *feeder* of type `parallel_do<Item>&`, can add more work by calling *`feeder`*`.add(`*`item`*`)`. For example, suppose processing a node in a tree is a prerequisite to processing its descendants. With `parallel_do`, after processing a node, you could use *`feeder`*`.add` to add the descendant nodes. The instance of `parallel_do` does not terminate until all items have been processed.

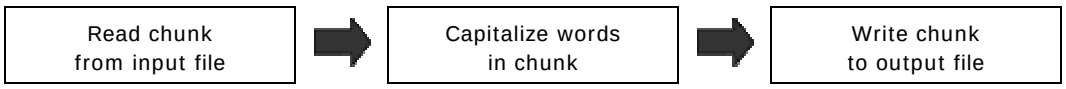
4.1.1 Code Sample

The directory `examples/parallel_do/parallel_preorder` contains a small application that uses `parallel_do` to perform parallel preorder traversal of an acyclic directed graph. The example shows how `parallel_do_feeder` can be used to add more work.

4.2 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.

The Intel® Threading Building Blocks classes `pipeline` and `filter` implement the pipeline pattern. A simple text processing problem will be used 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. Below is a picture of the 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 each of 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 *previous* character is a blank. For the first letter in each chunk, inspect the last letter of the previous chunk. But doing so would introduce a complicating dependence in the middle stage. The solution is to have each chunk also store the last character of the previous 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 class `MyBuffer`. It looks like a typical STL container for characters, except that `begin()[-1]` is legal and holds the last character of the previous chunk.

```
// Buffer that holds block of characters and last character of previous
buffer.
class MyBuffer {
    static const size_t buffer_size = 10000;
    char* my_end;
    // storage[0] holds the last character of the previous buffer.
    char storage[1+buffer_size];
public:
    // Pointer to first character in the buffer
    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 writing 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 all filters from the pipeline
    pipeline.clear();
```

The parameter to method `pipeline::run` controls the level of parallelism. Conceptually, *tokens* flow through the pipeline. In a serial in order stage, each token must be processed serially in order. In a parallel stage, multiple tokens can by processed in parallel by the stage. 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 method `pipeline::run` specifies the maximum number of tokens that can be in flight. Once this limit is reached, class `pipeline` never creates a new token at the input stage until another token is destroyed at the output stage.

This top-level code also shows the method `clear` that removes all stages from the pipeline. This call is optional, because a filter's destructor automatically removes it from a pipeline.[2]

Now look in detail at how the stages are defined. Each stage is derived from class `filter`. First consider the output stage, because it is the simplest.

```
// 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(serial_in_order),
    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 portions that "hook" it to the pipeline are shown in blue. The class is derived from the class `filter`. When its constructor calls the base class constructor for `filter`, it specifies that this is a serial in order filter. It must be a serial in order filter because it must write chunks in the same order the input stage reads them. 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.

---

[2] In TBB 2.0 and prior, filter destructors did not automatically remove filters, and thus the call to `clear` was mandatory.

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

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

MyTransformFilter::MyTransformFilter() :
    tbb::filter(parallel)
{}

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

Also, this stage operates on purely local data. Thus any number of invocations on `operator()` can run concurrently on the same instance of `MyTransformFilter`. It communicates this fact to the pipeline by specifying `parallel` as the parameter to the constructor of its base class `filter`.

The input filter is the most complicated, because it has to decide when it reaches the end of the input, and must allocate buffers.

```
class MyInputFilter: public tbb::filter {
public:
    static const size_t n_buffer = 8;
    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(serial_in_order),
    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 must be `serial_in_order` in this example because the filter reads chunks from a sequential file and the output filter must write the chunks in the same order. All serial_in_order filters process items in the same order. There is another kind of serial stage, `serial_out_of_order`, that does not preserve order. In the example, the override of `operator()` ignores its parameter, because it is generating a stream, not transforming it. It remembers the last character of the previous 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. 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 first and last stages are `serial_in_order`. Therefore, the buffers are retired by the last stage in the order they were allocated by the first stage.

Notice that if the first and last stage were *not* `serial_in_order`, then you would have to keep track of which buffers are currently in use, because buffers might not be retired in the same order they were allocated.

The directory `examples/pipeline/text_filter` contains the complete code for the text filter.

4.2.1 Throughput of pipeline

The throughput of a `pipeline` is the rate at which tokens flow through it, and is limited by two constraints. First, if a `pipeline` is run with *N* tokens, then obviously there 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

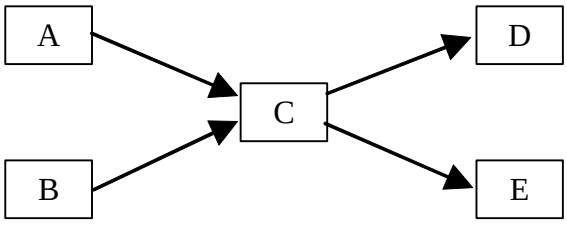
try to keep the sequential stages fast, and when possible, shift work to the parallel stages.

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

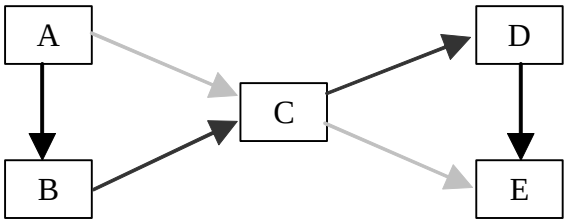
The window size, or sub-problem size for each token, can also limit throughput. Making windows too small may cause overheads to dominate the useful work. Making windows too large may cause them to spill out of cache. A good guideline is to try for a large window size that still fits in cache. You may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

4.2.2 Non-Linear Pipelines

Class `pipeline` supports only linear pipelines. It does not directly handle more baroque plumbing, such as in the diagram below.



However, you can still use `pipeline` for this. Just topologically sort the stages into a linear order, like this:



The light gray arrows are the original arrows that are now implied by transitive closure of the other arrows. It might seem that 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 non-linear 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

above may need to be modified in order 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 non-linear pipelines, it would add a lot of programming complexity, and not improve throughput. The linear limitation of `pipeline` is a good tradeoff of gain versus pain.

4.3 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 high task-pattern level and not worry about low-level manipulation of threads. Because they are generic, you can customize them to your specific needs. Have fun using the loop templates to unlock the power of multi-core.

5 Containers

Intel® Threading Building Blocks provides highly concurrent container classes. These containers can be used with raw Windows or Linux threads, or in conjunction with task-based programming (10.1).

A concurrent container allows multiple threads to concurrently access and update items in the container. Typical C++ STL containers do not permit concurrent update; attempts to modify them concurrently often result in corrupting the container. STL containers can be wrapped in a mutex to make them safe for concurrent access, by letting only one thread operate on the container at a time, but that approach eliminates concurrency, thus restricting parallel speedup.

Containers provided by Intel® Threading Building Blocks offer a much higher level of concurrency, via one or both of the following methods:

- **Fine-grained locking**. With fine-grain 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**. With lock-free algorithms, different threads account and correct for the effects of other interfering threads.

Notice that highly-concurrent containers are come at a cost. They typically have higher overheads than regular STL containers. Operations on highly-concurrent containers may take longer than for STL containers. Therefore, use highly-concurrent containers when the speedup from the additional concurrency that they enable outweighs their slower sequential performance.

CAUTION: As with most objects in C++, the constructor or destructor of a container object must not be invoked concurrently with another operation on the same object. Otherwise the resulting race may cause the operation to be executed on an undefined object.

5.1 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 traits type `HashCompare` defines how to hash a key and how to compare two keys.

The following example builds a `concurrent_hash_map` where the keys are strings and the corresponding data is the number of times each string occurs in the array `Data`.

```
#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];

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

    // Put occurrences into the table
    parallel_for( blocked_range<string*>( Data, Data+N, 1000 ),
                  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 purposes respectively with the classes `accessor` and `const_accessor` that act as smart pointers. An *accessor* represents *update* (*write*) access. As long as it points to an element, all other attempts to look up that key in the table block until the `accessor` is done. A `const_accessor` is similar, except that is represents *read-only* access. 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 methods `find` and `insert` 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. Once the method returns, the access lasts until the `accessor` or `const_accessor` is destroyed. Because having access to an element can block other threads, try to shorten the lifetime of the `accessor` or `const_accessor`. To do so, declare it in the innermost block possible. To release access even sooner than the end of the block, use method `release`. The following example is a rework of the loop body that uses `release` instead of depending upon destruction to end thread lifetime:

```
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 `key`.

5.1.1 More on HashCompare

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

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

The signatures are said to travel together in a single class because *if two keys are equal, then 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, then you should construct the `concurrent_hash_map` using the constructor that takes a *HashCompare* as a parameter. The following example is a variation on an earlier example with instance-

dependent methods. The instance performs both case-sensitive or case-insensitive hashing, and comparison, depending upon an internal flag ignore_case.

```
// 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 ) const {
        size_t h = 0;
        for( const char* s = x.c_str(); *s; s++ )
            h = (h*16777179)^*(ignore_case?tolower(*s):*s);
        return h;
    }
    // True if strings are equal
    bool equal( const string& x, const string& y ) const {
        if( ignore_case )
            strcasecmp( x.c_str(), y.c_str() )==0;
        else
            return x==y;
    }
    VariantHashCompare( bool ignore_case_ ) : 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.

5.2 concurrent_vector

A concurrent_vector<*T*> is a dynamically growable array of *T*. It is safe to grow a concurrent_vector while other threads are also operating on elements of it, or even growing it themselves. 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.

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 example, the following routine safely appends a C string to a shared vector:

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

```
}
```

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.

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

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

5.2.1 Clearing is Not Concurrency Safe

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

5.3 concurrent_queue

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.

Notice the behavior of concurrent 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. Use of `concurrent_queue` guarantees is that if a thread pushes two values, and another thread pops those two values, they will be popped in the same order that they were pushed.

There are blocking and non-blocking flavors of pop. Method `pop_if_present` is non-blocking. It attempts to pop a value, and if it cannot because the queue is empty, returns anyway. Method pop blocks until it pops a value. If a thread must wait for an item to become available, and 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. Method empty() 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 method set_capacity. 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.

5.3.1 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. The iterators go forwards only, and are too slow to be very useful in production code. If a queue is modified, all iterators pointing to it become invalid and unsafe to use. The following snippet dumps a queue. The operator<< is defined for a Foo.

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

5.3.2 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_do (4.1) or pipeline (4.2) instead. These options are often more efficient than queues for the following reasons:

- A queue is inherently a bottle neck, 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 a value, it could take time until it pops the value, and in the meantime the value (and whatever it references) becomes “cold” in cache. Or worse yet, another thread pops the value, and the value (and whatever it references) must be moved to the other processor.

In contrast, parallel_do and pipeline avoid these bottlenecks. Because their threading is implicit, they optimize use of worker threads so that they do other work until a value shows up. They also try to keep items hot in cache. For example, when

another work item is added to a `parallel_do`, it is kept local to the thread that added it unless another idle thread can steal it before the “hot” thread processes it. This way, items are more often processed by the hot thread.

5.4 Summary of Containers

The high-level containers in Intel® Threading Building Blocks enable common idioms for concurrent access. They are suitable for scenarios where the alternative would be a serial container with a lock around it.

6 Mutual Exclusion

Mutual exclusion controls how many threads can simultaneously run a region of code. In Intel® Threading Building Blocks, mutual exclusion is implemented by *mutexes* and *locks.* A mutex is an object on which a thread can acquire a lock. Only one thread at a time can have a lock on a mutex; other threads have to wait their turn.

The simplest mutex is `spin_mutex`. A thread trying to acquire a lock on a `spin_mutex` busy waits until it can acquire the lock. A `spin_mutex` is appropriate when the lock is held for only a few instructions. For example, the following code uses a mutex `FreeListMutex` to protect a shared variable `FreeList`. It checks that only a single thread has access to `FreeList` at a time. The black font shows the usual sequential code. The blue text shows code added to make the code thread-safe.

```
Node* FreeList;
typedef spin_mutex FreeListMutexType;
FreeListMutexType FreeListMutex;

Node* AllocateNode() {
    Node* n;
    {
        FreeListMutexType::scoped_lock lock(FreeListMutex);
        n = FreeList;
        if( n )
            FreeList = n->next;
    }
    if( !n )
        n = new Node();
    return n;
}

void FreeNode( Node* n ) {
    FreeListMutexType::scoped_lock lock(FreeListMutex);
    n->next = FreeList;
    FreeList = n;
}
```

The constructor for scoped_lock waits until there are no other locks on `FreeListMutex`. The destructor releases the lock. The braces inside routine `AllocateNode` may look unusual. Their role is to keep the lifetime of the lock as short as possible, so that other waiting threads can get their chance as soon as possible.

CAUTION: Be sure to name the lock object, otherwise it will be destroyed too soon. For example, if the creation of the scoped_lock object in the example is changed to

```
FreeListMutexType::scoped_lock (FreeListMutex);
```

then the scoped_lock is destroyed when execution reaches the semicolon, which releases the lock *before* `FreeList` is accessed.

An alternative way to write AllocateNode is as follows:

```
Node* AllocateNode() {
    Node* n;
    FreeListMutexType::scoped_lock lock;
    lock.acquire(FreeListMutex);
    n = FreeList;
    if( n )
        FreeList = n->next;
    lock.release();
    if( !n )
        n = new Node();
    return n;
}
```

Method acquire waits until it can acquire a lock on the mutex; method release releases the lock.

It is recommended that you add extra braces where possible, to clarify to maintainers which code is protected by the lock.

If you are familiar with C interfaces for locks, you may be wondering why there are not simply acquire and release methods on the mutex object itself. The reason is that the C interface would not be exception safe, because if the protected region threw an exception, control would skip over the release. With the object-oriented interface, destruction of the scoped_lock object causes the lock to be released, no matter whether the protected region was exited by normal control flow or an exception. This is true even for our version of AllocateNode that used methods acquire and release – the explicit release causes the lock to be released earlier, and the destructor then sees that the lock was released and does nothing.

All mutexes in Intel® Threading Building Blocks have a similar interface, which not only makes them easier to learn, but enables generic programming. For example, all of the mutexes have a nested scoped_lock type, so given a mutex of type *M*, the corresponding lock type is *M*::scoped_lock.

TIP: It is recommended that you always use a typedef for the mutex type, as shown in the previous examples. That way, you can change the type of the lock later without having to edit the rest of the code. In the examples, you could replace the typedef with typedef queuing_mutex FreeListMutexType, and the code would still be correct.

6.1.1 Mutex Flavors

Connoisseurs of mutexes distinguish various attributes of mutexes. It helps to know some of these, because they involve tradeoffs of generality and efficiency. Picking the right one often helps performance. Mutexes can be described by the following qualities, also summarized in Table 10

- **Scalable**. Some mutexes are called *scalable*. In a strict sense, this is not an accurate name, because a mutex limits execution to one thread at a time. A *scalable mutex* is one that does not do *worse* than this. A mutex can do worse than serialize execution if the waiting threads consume excessive processor cycles and memory bandwidth, reducing the speed of threads trying to do real work. Scalable mutexes are often slower than non-scalable mutexes under light contention, so a non-scalable mutex may be better. When in doubt, use a scalable mutex.
- **Fair**. Mutexes can be *fair* or *unfair*. A fair mutex lets threads through in the order they arrived. Fair mutexes avoid starving threads. Each thread gets its turn. However, unfair mutexes can be faster, because they let threads that are running go through first, instead of the thread that is next in line which may be sleeping on account of an interrupt.
- **Recursive**. Mutexes can be *recursive* or *non-recursive*. A recursive mutex allows a thread that is already holding a lock on the mutex to acquire another lock on the mutex. This is useful in some recursive algorithms, but typically adds overhead to the lock implementation.
- **Yield or Block**. This is an implementation detail that impacts performance. On long waits, a TBB mutex either *yields* or *blocks*. Here *yields* means to repeatedly poll whether progress can be made, and if not, temporarily yield[3] the processor. To *block* means to yield the processor until the mutex permits progress. Use the yiedling mutexes if waits are typically short and blocking mutexes if waits are typically long.

The following is a summary of mutex behaviors:

- `spin_mutex` is non-scalable, unfair, non-recursive, and spins in user space. It would seem to be the worst of all possible worlds, except that it is *very fast* in *lightly contended* situations. If you can design your program so that contention is somehow spread out among many `spin_mutex` objects, you can improve performance over using other kinds of mutexes. If a mutex is heavily contended, your algorithm will not scale anyway. Consider redesigning the algorithm instead of looking for a more efficient lock.
- `queuing_mutex` is scalable, fair, non-recursive, and spins in user space. Use it when scalability and fairness are important.
- `spin_rw_mutex` and `queuing_rw_mutex` are similar to `spin_mutex` and `queuing_mutex`, but additionally support *reader* locks.
- `mutex` and `recursive_mutex` are wrappers around the system's "native" mutual exclusion. On Windows* systems it is implemented on top of `CRITICAL_SECTION`. On Linux* and Mac OS* X systems it is implemented on top of `pthread` mutex. The advantages of using the wrapper are that it adds an exception-safe interface and it provides an interface identical to the other mutexes in Intel® Threading Building Blocks, which makes it easy to swap in a different kind of mutex later if warranted by performance measurements.
- `null_mutex` and `null_rw_mutex` do nothing. They can be useful as template arguments. For example, suppose you are defining a container template and know

---

[3] The yielding is implemented via `SwitchToThread()` on Microsoft Windows* systems and by `sched_yield()` on other systems.

that some instantiations will be shared by multiple threads and need internal locking, but others will be private to a thread and not need locking. You can define the template to take a Mutex type parameter. The parameter can be one of the real mutex types when locking is necessary, and `null_mutex` when locking is unnecessary.

Table 10: Traits and Behaviors of Mutexes

Mutex	Scalable	Fair	Recursive	Long Wait	Size
`mutex`	OS dependent	OS dependent	no	blocks	≥ 3 words
`recursive_mutex`	OS dependent	OS dependent	yes	blocks	≥ 3 words
`spin_mutex`	no	no	no	yields	1 byte
`queuing_mutex`	✓	✓	no	yields	1 word
`spin_rw_mutex`	no	no	no	yields	1 word
`queuing_rw_mutex`	✓	✓	no	yields	1 word
`null_mutex`[4]	moot	✓	✓	never	empty
`null_rw_mutex`	moot	✓	✓	never	empty

6.1.2 Reader Writer Mutexes

Mutual exclusion is necessary when at least one thread *writes* to a shared variable. But it does no harm to permit multiple readers into a protected region. The reader-writer variants of the mutexes, denoted by `_rw_` in the class names, enable multiple readers by distinguishing *reader locks* from *writer locks.* There can be more than one reader lock on a given mutex.

Requests for a reader lock are distinguished from requests for a writer lock via an extra boolean parameter in the constructor for `scoped_lock`. The parameter is false to request a reader lock and true to request a writer lock. It defaults to `true` so that when omitted, a `spin_rw_mutex` or `queuing_rw_mutex` behaves like its non-`_rw_` counterpart. The next section shows an example where the parameter is explicitly `false` in order to obtain a reader lock.

[4] Null mutexes are considered fair by TBB because they cannot cause starvation. They lack any non-static data members.

6.1.3 Upgrade/Downgrade

It is possible to upgrade a reader lock to a writer lock, by using the method `upgrade_to_writer`. Here is an example.

```
std::vector<string> MyVector;
typedef spin_rw_mutex MyVectorMutexType;
MyVectorMutexType MyVectorMutex;

void AddKeyIfMissing( const string& key ) {
    // Obtain a reader lock on MyVectorMutex
    MyVectorMutexType::scoped lock
lock(MyVectorMutex,/*is_writer=*/false);
    size_t n = MyVector.size();
    for( size_t i=0; i<n; ++i )
        if( MyVector[i]==key ) return;
    if( !MyVectorMutex.upgrade_to_writer() )
        // Check if key was added while lock was temporarily released
        for( int i=n; i<MyVector.size(); ++i )
           if(MyVector[i]==key ) return;
    vector.push_back(key);
}
```

Note that the vector must sometimes be searched again. This is necessary because `upgrade_to_writer` might have to temporarily release the lock before it can upgrade. Otherwise, deadlock might ensue, as discussed in Section 6.1.4. Method `upgrade_to_writer` returns a `bool` that is true if it successfully upgraded the lock without releasing it, and false if the lock was released temporarily. Thus when `upgrade_to_writer` returns false, the code must rerun the search to check that the key was not inserted by another writer. The example presumes that keys are always added to the end of the vector, and that keys are never removed. Because of these assumptions, it does not have to re-search the entire vector, but only the elements beyond those originally searched. The key point to remember is that when `upgrade_to_writer` returns false, any assumptions established while holding a reader lock may have been invalidated, and must be rechecked.

For symmetry, there is a corresponding method `downgrade_to_reader`, though in practice there are few reasons to use it.

6.1.4 Lock Pathologies

Locks can introduce performance and correctness problems. If you are new to locking, here are some of the problems to avoid:

6.1.4.1 Deadlock

Deadlock happens when threads are trying to acquire more than one lock, and each holds some of the locks the other threads need to proceed. More precisely, deadlock happens when:

- There is a cycle of threads
- Each thread holds at least one lock on a mutex, and is waiting on a mutex for which the *next* thread in the cycle already has a lock.
- No thread is willing to give up its lock.

Think of classic gridlock at an intersection – each car has “acquired” part of the road, but needs to “acquire” the road under another car to get through. Two common ways to avoid deadlock are:

- Avoid needing to hold two locks at the same time. Break your program into small actions in which each can be accomplished while holding a single lock.
- Always acquire locks in the same order. For example, if you have “outer container” and “inner container” mutexes, and need to acquire a lock on one of each, you could always acquire the “outer sanctum” one first. Another example is “acquire locks in alphabetical order” in a situation where the locks have names. Or if the locks are unnamed, acquire locks in order of the mutex’s numerical addresses.
- Use atomic operations instead of locks, as discussed in the following section.

6.1.4.2 Convoying

Another common problem with locks is *convoying*. Convoying occurs when the operating system interrupts a thread that is holding a lock. All other threads must wait until the interrupted thread resumes and releases the lock. Fair mutexes can make the situation even worse, because if a waiting thread is interrupted, all the threads behind it must wait for it to resume.

To minimize convoying, try to hold the lock as briefly as possible. Precompute whatever you can before acquiring the lock.

To avoid convoying, use atomic operations instead of locks where possible.

7 Atomic Operations

You can avoid mutual exclusion using atomic operations. When a thread performs an atomic operation, the other threads see it as happening instantaneously. The advantage of atomic operations is that they are relatively quick compared to locks, and do not suffer from deadlock and convoying. The disadvantage is that they only do a limited set of operations, and often these are not enough to synthesize more complicate operations efficiently. But nonetheless you should not pass up an opportunity to use an atomic operation in place of mutual exclusion. Class `atomic<T>` implements atomic operations with C++ style.

A classic use of atomic operations is for thread-safe reference counting. Suppose `x` is a reference count of type `int`, and the program needs to take some action when the reference count becomes zero. In single-threaded code, you could use a plain `int` for `x`, and write `--x; if(x==0) action()`. But this method might fail for multithreaded code, because two threads might interleave their operations as shown in the following table, where t_a and t_b represent machine registers, and time progresses downwards:

Table 11: Interleaving of Machine Instructions

Thread A	Thread B
t_a = x	
	t_b = x
x = t_a - 1	
	x = t_b – 1
if(x==0)	
	if(x==0)

Though the code intended for `x` to be decremented twice, it ends up with only one less than its original value. Also, another problem results because the test of *x* is separate from the decrement: If *x* starts out as two, and both threads decrement *x* before either thread evaluates the `if` condition, *both* threads would call `action()`. To correct this problem, you need to ensure that only one thread at a time does the decrement *and* ensure that the value checked by the "if" is the result of the decrement. You can do this by introducing a mutex, but it is much faster and simpler to declare *x* as `atomic<int>` and write "`if(--`*x*`==0) action()`". The method `atomic<int>::operator--` acts atomically; no other thread can interfere.

`atomic<`*T*`>` supports atomic operations on type *T*, which must be an integral or pointer type. There are five fundamental operations supported, with additional interfaces in the form of overloaded operators for syntactic convenience. For example, `++`, `--`, `-=`, and `+=`operations on `atomic<`*T*`>` are all forms of the fundamental operation *fetch-and-*

add. The following are the five fundamental operations on a variable *x* of type `atomic<T>`.

Table 12: Fundamental Operations on a Variable x of Type atomic<T>

`= x`	read the value of *x*
`x =`	write the value of *x*, and return it
`x.fetch_and_store(y)`	do *y*=*x* and return the old value of *x*
`x.fetch_and_add(y)`	do *x*+=*y* and return the old value of *x*
`x.compare_and_swap(y,z)`	if *x* equals *z*, then do *x*=*y*. In either case, return old value of *x*.

Because these operations happen atomically, they can be used safely without mutual exclusion. Consider the following example:

```
atomic<unsigned> counter;

unsigned GetUniqueInteger() {
    return counter.fetch_and_add(1);
}
```

The routine `GetUniqueInteger` returns a different integer each time it is called, until the counter wraps around. This is true no matter how many threads call `GetUniqueInteger` simultaneously.

The operation `compare_and_swap` is fundamental operation to many non-blocking algorithms. A problem with mutual exclusion is that if a thread holding a lock is suspended, all other threads are blocked until the holding thread resumes. Non-blocking algorithms avoid this problem by using atomic operations instead of locking. They are generally complicated and require sophisticated analysis to verify. However, the following idiom is straightforward and worth knowing. It updates a shared variable `globalx` in a way that is somehow based on its old value:

```
atomic<int> globalx;

int UpdateX() {     // Update x and return old value of x.
    do {
        // Read globalX
        oldx = globalx;
        // Compute new value
        newx = ...expression involving oldx....
        // Store new value if another thread has not changed globalX.
    } while( globalx.compare_and_swap(newx,oldx)!=oldx );
    return oldx;
}
```

Worse, some threads iterate the loop until no other thread interferes. Typically, if the update takes only a few instructions, the idiom is faster than the corresponding mutual-exclusion solution.

CAUTION: If the following sequence thwarts your intent, then the update idiom is inappropriate:

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

The problem is called the *ABA problem*. It is frequently a problem in designing non-blocking algorithms for linked data structures. See the Internet for more information.

7.1.1 Why atomic<T> Has No Constructors

Template class `atomic<`*T*`>` deliberately has no constructors, because examples like `GetUniqueInteger` in Chapter 7 are commonly required to work correctly even before all file-scope constructors have been called. If `atomic<`*T*`>` had constructors, a file-scope instance might be initialized after it had been referenced. You can rely on zero-initialization to initialize an `atomic<`*T*`>` to zero.

7.1.2 Memory Consistency

Some architectures, such as Itanium, have “weak memory consistency”, in which memory operations on different addresses may be reordered by the hardware for sake of efficiency. The subject is complex, and it is recommended that the interested reader consult other works (Intel 2002, Robison 2003) on the subject. If you are programming only IA-32 and Intel® Extended Memory 64 Technology (Intel® EM64T) processor platforms, you can skip this section.

Class atomic<T> permits you to enforce certain ordering of memory operations as described in Table 13:

Table 13: Ordering Constraints

Kind	Description	Default For
`acquire`	Operations after the atomic operation never move over it.	read
`release`	Operations before the atomic operation never move over it.	write

sequentially consistent	Operations on either side never move over the atomic operation and the sequentially consistent atomic operations have a global order.	`fetch_and_store`
		`fetch_and_add`
		`compare_and_swap`

The rightmost column lists the operations that default to a particular constraint. Use these defaults to avoid unexpected surprises. For read and write, the defaults are the only constraints available. However, if you are familiar with weak memory

consistency, you might want to change the default sequential consistency for the other operations to weaker constraints. To do this, use variants that take a template argument. The argument can be `acquire` or `release`, which are values of the enum type `memory_semantics`.

For example, suppose various threads are producing parts of a data structure, and you want to signal a consuming thread when the data structure is ready. One way to do this is to initialize an atomic counter with the number of busy producers, and as each producer finishes, it executes:

```
refcount.fetch_and_add<release>(-1);
```

The argument `release` guarantees that the producer's writes to shared memory occurs before `refcount` is decremented. Similarly, when the consumer checks `refcount`, the consumer must use an `acquire` fence, which is the default for reads, so that the consumer's reads of the data structure do not happen until after the consumer sees `refcount` become 0.

8 Timing

When measuring the performance of parallel programs, it is usually *wall clock* time, not CPU time, that matters. The reason is that better parallelization typically increases aggregate CPU time by employing more CPUs. The goal of parallelizing a program is usually to make it run *faster* in real time.

The class `tick_count` in Intel® Threading Building Blocks provides a simple interface for measuring wall clock time. A `tick_count` value obtained from the static method `tick_count::now()` represents the current absolute time. Subtracting two `tick_count` values yields a relative time in `tick_count::interval_t`, which you can convert to seconds, as in the following example:

```
tick_count t0 = tick_count::now();
... do some work ...
tick_count t1 = tick_count::now();
printf("work took %g seconds\n",(t1-t0).seconds());
```

Unlike some timing interfaces, `tick_count` is guaranteed to be safe to use across threads. It is valid to subtract `tick_count` values that were created by different threads. A `tick_count` difference can be converted to seconds.

The resolution of `tick_count` corresponds to the highest resolution timing service on the platform that is valid across threads in the same process. Since the CPU timer registers are *not* valid across threads on some platforms, this means that the resolution of `tick_count` can not be guaranteed to be consistent across platforms.

9 Memory Allocation

Intel® Threading Building Blocks provides two memory allocator templates that are similar to the STL template class `std::allocator`. These two templates, `scalable_allocator<T>` and `cache_aligned_allocator<T>`, address critical issues in parallel programming as follows:

- **Scalability**. Problems of scalability arise when using memory allocators originally designed for serial programs, on threads that might have to compete for a single shared pool in a way that allows only one thread to allocate at a time. Use the memory allocator template `scalable_allocator<T>` to avoid such scalability bottlenecks. This template can improve the performance of programs that rapidly allocate and free memory.
- **False sharing**. Problems of sharing arise when two threads access different words that share the same cache line. The problem is that a cache line is the unit of information interchange between processor caches. If one processor modifies a cache line and another processor reads (or writes) the same cache line, the cache line must be moved from one processor to the other, even if the two processors are dealing with different words within the line. False sharing can hurt performance because cache lines can take hundreds of clocks to move.

  Use the class `cache_aligned_allocator<T>` to always allocate on a cache line. Two objects allocated by `cache_aligned_allocator` are guaranteed to not have false sharing. If an object is allocated by `cache_aligned_allocator` and another object is allocated some other way, there is no guarantee. The interface to `cache_aligned_allocator` is identical to `std::allocator`, so you can use it as the *allocator* argument to STL template classes.

The following code shows how to declare an STL vector that uses `cache_aligned_allocator` for allocation:

```
std::vector<int,cache_aligned_allocator<int> >;
```

TIP: The functionality of `cache_aligned_allocator<T>` comes at some cost in space, because it must allocate at least one cache line's worth of memory, even for a small object. So use `cache_aligned_allocator<T>` only if false sharing is likely to be a real problem.

The scalable memory allocator incorporates McRT technology developed by Intel's PSL CTG team.

9.1 Which Dynamic Libraries to Use

The template `scalable_allocator<T>` requires the Intel® Threading Building Blocks scalable memory allocator library as described in Section 2.2. It does not require the

Intel® Threading Building Blocks general library, and can be used independently of the rest of Intel® Threading Building Blocks.

The templates `tbb_allocator<T>` and `cache_aligned_allocator<T>` use the scalable allocator library if it is present otherwise it reverts to using `malloc` and `free`. Thus, you can use these templates even in applications that choose to omit the scalable memory allocator library.

The rest of Intel® Threading Building Blocks can be used with or without the Intel® Threading Building Blocks scalable memory allocator library.

Table 14: Templates and Libraries

Template	Requirements	Notes
`scalable_allocator<T>`	Intel® Threading Building Blocks scalable memory allocator library. See Section 2.2	
`tbb_allocator<T>` `cache_aligned_allocator<T>`		Uses the scalable allocator library if it is present otherwise it reverts to using `malloc` and `free`.

10 The Task Scheduler

This section introduces the Intel® Threading Building Blocks *task scheduler* . The task scheduler is the engine that powers the loop templates. When practical, you should use the loop templates instead of the task scheduler, because the templates hide the complexity of the scheduler. However, if you have an algorithm that does not naturally map onto one of the high-level templates, use the task scheduler. All of the scheduler functionality that is used by the high-level templates is available for you to use directly, so you can built new high-level templates that are just as powerful as the existing ones.

10.1 Task-Based Programming

When striving for performance, programming in terms of threads can be a poor way to do multithreaded programming. It is much better to formulate your program in terms of *logical tasks*, not threads, for several reasons.

- Matching parallelism to available resources
- Faster task startup and shutdown
- More efficient evaluation order
- Improved load balancing
- Higher–level thinking

The following paragraphs explain these points in detail.

The threads you create with a threading package are *logical* threads, which map onto the *physical threads* of the hardware. For computations that do not wait on external devices, highest efficiency usually occurs when there is exactly one running logical thread per physical thread. Otherwise, there can be inefficiencies from the mismatch. *Undersubscription* occurs when there are not enough running logical threads to keep the physical threads working. *Oversubscription* occurs when there are more running logical threads than physical threads. Oversubscription usually leads to *time sliced* execution of logical threads, which incurs overheads as discussed in Appendix A, Costs of Time Slicing. The scheduler tries to avoid oversubscription, by having one logical thread per physical thread, and mapping tasks to logical threads, in a way that tolerates interference by other threads from the same or other processes.

The key advantage of tasks versus logical threads is that tasks are much *lighter weight* than logical threads. 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. This is because a thread has its own copy of a lot of resources, such as register state and a stack. On Linux, a thread even has its own process id. A task in Intel® Threading Building Blocks, in contrast, is typically a small routine, and also, cannot be preempted at the task level (though its logical thread can be preempted).

Tasks in Intel® Threading Building Blocks are efficient too because *the scheduler is unfair.* Thread schedulers typically distribute time slices in a round-robin fashion. This distribution is called “fair”, because each logical thread gets its fair share of time. Thread schedulers are typically fair because it is the safest strategy to undertake without understanding the higher-level organization of a program. In task-based programming, the task scheduler does have some higher-level information, and so can sacrifice fairness for efficiency. Indeed, it often delays starting a task until it can make useful progress. Section 10.3 explains how this works, and how it saves both time and space.

The scheduler does *load balancing.* In addition to using the right number of threads, it is important to distribute work evenly across those threads. As long as you break your program into enough small tasks, the scheduler usually does a good job of assigning tasks to threads to balance load. With thread-based programming, you are often stuck dealing with load-balancing yourself, which can be tricky to get right.

Finally, the main advantage of using tasks instead of threads is that they let you think at a higher, task-based, level. With thread-based programming, you are forced to think at the low level of physical threads to get good efficiency, because you have one logical thread per physical thread to avoid undersubscription or oversubscription. You also have to deal with the relatively coarse grain of threads. With tasks, you can concentrate the logical dependences between tasks, and leave the efficient scheduling to the scheduler.

10.1.1 When Task-Based Programming Is Inappropriate

Using the task scheduler is usually the best approach to threading for performance, however there are cases when the task scheduler is not appropriate. The task scheduler is intended for high-performance algorithms composed from non-blocking tasks. It still works if the tasks rarely block. However, if threads block frequently, there is a performance loss when using the task scheduler because while the thread is blocked, it is not working on any tasks. Blocking typically occurs while waiting for I/O or mutexes for long periods. If threads hold mutexes for long periods, your code is not likely to perform well anyway, no matter how many threads it has. If you have blocking tasks, it is best to use full-blown threads for those. The task scheduler is designed so that you can safely mix your own threads with Intel® Threading Building Blocks tasks.

10.2 Simple Example: Fibonacci Numbers

This section uses computation of the *n*th Fibonacci number as an example. This example uses an inefficient method[5] to compute Fibonacci numbers, but it demonstrates the basics of a task library using a simple recursive pattern. To get scalable speedup out of task-based programming, you need to specify a lot of tasks. This is typically done in Intel® Threading Building Blocks with a recursive task pattern.

This is the serial code:

```
long SerialFib( long n ) {
    if( n<2 )
        return n;
    else
        return SerialFib(n-1)+SerialFib(n-2);
}
```

The top-level code for the parallel task-based version is:

```
long ParallelFib( long n ) {
    long sum;
    FibTask& a = *new(task::allocate_root()) FibTask(n,&sum);
    task::spawn_root_and_wait(a);
    return sum;
}
```

This code uses a task of type `FibTask` to do the real work. It involves the following distinct steps :

1. Allocate space for the task. This is done by a special "overloaded new" and method `task::allocate_root`. The `_root` suffix in the name denotes the fact that the task created has no parent. It is the root of a task tree. Tasks must be allocated by special methods so that the space can be efficiently recycled when the task completes.
2. Construct the task with the constructor `FibTask(n,&sum)` invoked by `new`. When the task is run in step 3, it computes the nth Fibonacci number and stores it into `*sum`.
3. Run the task to completion with `task::spawn_root_and_wait`.

The real work is inside struct `FibTask`. Its definition is shown below.

```
class FibTask: public task {
public:
    const long n;
    long* const sum;
```

---

[5] An efficient method is to compute $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1}$ and take the upper left element. The exponentiation can be done quickly via repeated squaring.

```
    FibTask( long n_, long* sum_ ) :
        n(n_), sum(sum_)
    {}
    task* execute() {             // Overrides virtual function
task::execute
        if( n<CutOff ) {
            *sum = SerialFib(n);
        } else {
            long x, y;
            FibTask& a = *new( allocate_child() ) FibTask(n-1,&x);
            FibTask& b = *new( allocate_child() ) FibTask(n-2,&y);
            // Set ref_count to "two children plus one for the wait".
            set_ref_count(3);
            // Start b running.
            spawn( b );
            // Start a running and wait for all children (a and b).
            spawn_and_wait_for_all( a );
            // Do the sum
            *sum = x+y;
        }
        return NULL;
    }
};
```

It is a relatively large piece of code, compared to `SerialFib`, because it expresses parallelism without the help of any extensions to standard C++.

Like all tasks scheduled by Intel® Threading Building Blocks, `FibTask` is derived from class `task`. Fields `n` and `sum` hold respectively the input value and pointer to the output. These are copies of the arguments passed to the constructor for `FibTask`. Method `execute` does the actual computation. Every task must provide a definition of `execute` that overrides the pure virtual method `task::execute`. The definition should do the work of the task, and return either NULL, or a pointer to the next task to run. In this simple example, it returns NULL. More is said about the non-NULL case in Section 10.4.3.

Method `FibTask::execute()`does the following:

- Checks if `n` is so small that serial execution would be faster. Finding the right value of `CutOff` requires some experimentation. A value of at least 16 works well in practice for getting the most of the possible speedup out of this example. Resorting to a sequential algorithm when the problem size becomes small is characteristic of most divide-and-conquer patterns for parallelism. Finding the point at which to switch requires experimentation, so be sure to write your code in a way that allows you to experiment.
- If the `else` is taken, the code creates and runs two child tasks that compute the (n-1)th and (n-2)th Fibonacci numbers. Here, inherited method `allocate_child()` is used to allocate space for the task. Remember that the top-level routine `ParallelFib` used `allocate_root()` to allocate space for a task. The difference is that here the task is creating *child* tasks. This relationship is indicated by the choice of allocation method.

- Calls `set_ref_count(3)`. The number `3` represents the two children and an additional implicit reference that is required by method `spawn_and_wait_for_all`. Make sure to call `set_reference_count(3)` before spawning any children. Failure to do so results in undefined behavior. The debug version of the library usually detects and reports this type of error.
- Spawns two child tasks. Spawning a task indicates to the scheduler that it can run the task whenever it chooses, possibly in parallel with executing other tasks. The execution policy is explained later in Section 10.3. The first spawning, by method `spawn`, returns immediately without waiting for the child task to start executing. The second spawning, by method `spawn_and_wait_for_all`, causes the parent to wait until all currently allocated child tasks are finished.
- After the two child tasks complete, the parent computes `x+y` and stores it in `*sum`.

At first glance, the parallelism might appear to be limited, because the task creates only two child tasks. The trick here is *recursive parallelism.* The two child tasks each create two child tasks, and so on, until `n<Cutoff`. This chain reaction creates a lot of potential parallelism. The advantage of the task scheduler is that it turns this potential parallelism into real parallelism in a very efficient way, because it chooses tasks to run in a way that keeps physical threads busy with relatively little context switching.

10.3 How Task Scheduling Works

The scheduler evaluates a *task graph.* The graph is a directed graph where nodes are tasks, and each points to its *parent* which is another task that is waiting on it to complete, or NULL. Method `task::parent()` gives you read-only access to the parent pointer. Each task has a *refcount* that counts the number of tasks that have it as a parent. Each task also has a *depth*, which is usually one more than the depth of its parent. The following snapshot shows a task graph for the Fibonacci example.

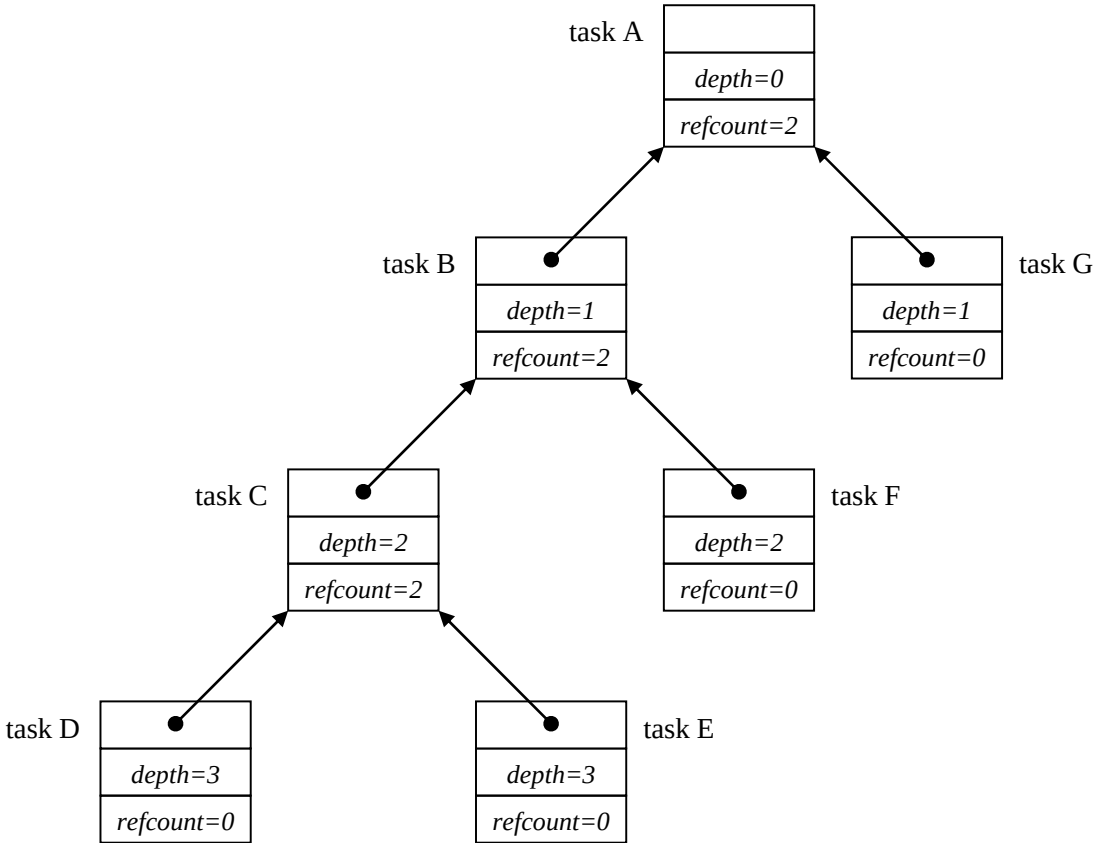


Figure 7: Task Graph for the Fibonacci Example

In the snapshot, the tasks with non-zero reference counts (A, B, and C) wait for their child tasks. The leaf tasks are running or ready to run.

The scheduler runs tasks in a way that tends to minimize both memory demands and cross-thread communication. The intuition is that a balance must be reached between depth-first and breadth-first execution. Assuming that the tree is finite, depth-first is best for sequential execution for the following reasons:

- **Strike when the cache is hot**. The deepest tasks are the most recently created tasks, and therefore are hottest in cache. Also, if they can complete, then task C can continue executing, and though not the hottest in cache, it is still warmer than the older tasks above it.
- **Minimize space**. Executing the shallowest task leads to breadth-first unfolding of the tree. This creates an exponential number of nodes that coexist simultaneously. In contrast, depth-first execution creates the same number of nodes, but only a linear number have to exist at the same time, because it stacks the other ready tasks (E, F, and G in the picture).

Though breadth-first execution has a severe problem with memory consumption, it does maximize parallelism if you have an infinite number of physical threads. Since physical threads are limited, it is better to use only enough breadth-first execution to

keep the available processors busy. The scheduler implements breadth-first execution as follows:

- Each thread has its own *ready pool*, which is an array of lists of tasks.
- A task goes into the pool when it is deemed ready to run.
- Each thread steals tasks from other pools when necessary.[6]

Figure 8 shows a snapshot of a pool that corresponds to the task graph in Figure 7.

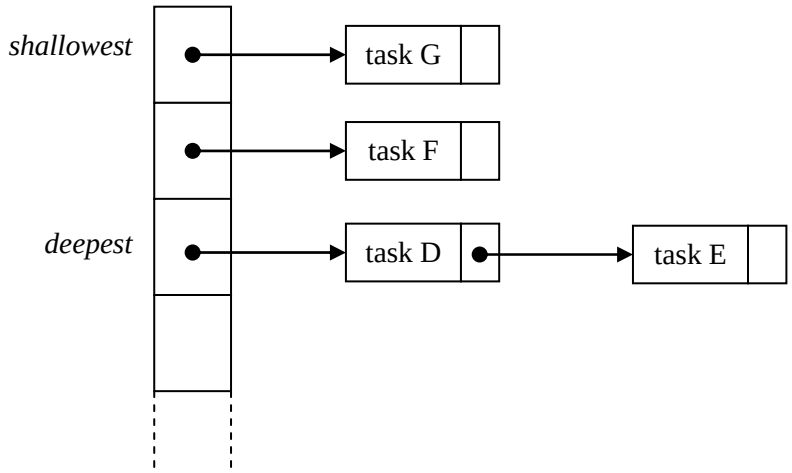


Figure 8: A Thread's Ready Pool

The pool comprises an array of lists. The array is subscripted by the task's depth. The lists are treated as stacks. Tasks are pushed onto the left side of a list, and likewise popped from the left side. There are two intertwined actions on each ready pool: putting tasks into the pool, and getting tasks out of the pools to run them.

The rule for getting is that when a thread participates in task graph evaluation and needs a new task to run, it gets the task by the first of the following rules that applies:

1. Use the task returned by method `execute` for the previous task. This rule does not apply if `execute` returned `NULL`.
2. Take the task at the front of the *deepest* list of its *own* pool. This rule does not apply if all lists in its pool are empty.
3. Steal from the front of the *shallowest* list of *another* randomly chosen pool. If the chosen pool is empty, the thread tries this rule again until it succeeds.

Getting is always automatic; it just happens as part of task graph evaluation. Putting can be explicit or automatic.

---

[6] The task scheduler is inspired by the early Cilk scheduler. See "Cilk: An Efficient Multithreaded Runtime System", PPoPP 95 to read about it.

Here are the ways in which a task can be put into a ready pool: The task always goes into the ready pool of the putting thread. Stealing from another pool is allowed; donating to another pool is not.

There are three ways that a task can be put into a ready pool:

- The task is explicitly spawned, for example, by method `spawn`.
- A task has been marked for re-execution by method `task::recycle_to_reexecute`.
- The task's reference count becomes zero after being implicitly decremented when a child task completes. This does not always happen when the last child task completes, because sometimes a fictitious "guard reference" is added, in scenarios where automatic spawning of a task is not wanted.

To summarize, the task scheduler's fundamental strategy is "breadth-first theft and depth-first work". The breadth-first theft rule raises parallelism sufficiently to keep threads busy. The depth-first work rule keeps each thread operating efficiently once it has sufficient work to do.

10.4 Useful Task Techniques

This section explains programming techniques for making best use of the scheduler.

10.4.1 Recursive Chain Reaction

The scheduler works best with tree-structured task graphs, because that is where the strategy of "breadth-first theft and depth-first work" applies very well. Also, tree-structured task graphs allow fast creation of many tasks. For example, if a master task tries to create *N* children directly, it will take O(*N*) steps. But with tree structured forking, it takes only O(lg(*N*)) steps.

Often domains are not obviously tree structured, but you can easily map them to trees. For example, `parallel_for` (in `tbb/parallel_for`) works over an iteration space, for example, a sequence of integers. Section 3.4 shows how the iteration space is defined in terms of how to split it into two halves. Template function `parallel_for` uses that definition to recursively map the iteration space onto a binary tree.

10.4.2 Continuation Passing

Method `spawn_and_wait_for_all` is a convenient way to wait on child tasks, but incurs some inefficiency if a thread becomes idle. The idle thread attempts to keep busy by stealing tasks from other threads. The scheduler limits possible victim tasks to those deeper than the waiting task. This limit modifies the policy that the shallowest task should be chosen. The limit restrains memory demands in worse-case

scenarios. A way around the constraint is for the parent to not wait, but simply spawn both children and return. The children are allocated not as children of the parent, but as children of the parent's *continuation task*, which is a task that runs when both children complete. The "continuation-passing" variant of FibTask is shown below, with the changed portions in blue ink.

```
struct FibContinuation: public task {
    long* const sum;
    long x, y;
    FibContinuation( long* sum_ ) : sum(sum_) {}
    task* execute() {
        *sum = x+y;
        return NULL;
    }
};

struct FibTask: public task {
    const long n;
    long* const sum;
    FibTask( long n_, long* sum_ ) :
        n(n_), sum(sum_)
    {}
    task* execute() {
        if( n<CutOff ) {
            *sum = SerialFib(n);
            return NULL;
        } else {
            FibContinuation& c =
                *new( allocate_continuation() ) FibContinuation(sum);
            FibTask& a = *new( c.allocate_child() ) FibTask(n-2,&c.x);
            FibTask& b = *new( c.allocate_child() ) FibTask(n-1,&c.y);
            // Set ref_count to "two children plus one for the wait".
            c.set_ref_count(23);
            c.spawn( b );
            c.spawn( a );
            return NULL;
        }
    }
};
```

The following differences between the original version and the continuation version need to be understood:

The big difference is that in the original version x and y were local variables in method execute. In the continuation-passing version, they cannot be local variables, because the parent returns before its children complete. Instead, they are fields of the continuation task FibContinuation.

The allocation logic is changed. The continuation is allocated with allocate_continuation. It is similar to allocate_child, except that the depth of the continuation is the same as the parent, not one deeper as it would be for a child. Also, it forwards the *parent* of this to c, and sets the *parent* of this to NULL. The following figure summarizes the transformation:

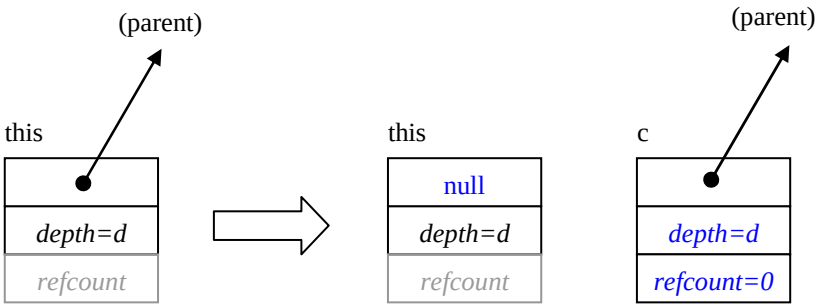


Figure 9: Action of `allocate_child`

A property of the transformation is that it does not change the reference count of the parent, and thus avoids interfering with reference-counting logic.

The reference count is set to `2`, the number of children. In the original version, it was set to `3` because `spawn_and_wait_for_all` required the augmented count. Furthermore, the code sets the reference count of the continuation instead of the parent, because it is the execution of the continuation that waits on the children.

The pointer `sum` is passed to the continuation by the constructor, because it is now `FibContinuation` that stores into `*sum`. The children are still allocated with `allocate_child`, but notice that now they are allocated as children of the continuation `c`, not the parent. This is so that `c`, and not `this`, becomes the "dependent" of the children that is automatically spawned when both children complete. If you accidentally used `this.allocate_child()`, then the parent task would run again after both children completed.

If you remember how the original top-level code, `ParallelFib`, was written, you might be worried now that continuation-passing style breaks the code, because now the root `FibTask` completes before the children are done, and the top-level code used `spawn_root_and_wait` to wait on the root `FibTask`. This is not a problem, because `spawn_root_and_wait` is designed to work correctly with continuation-passing style. An invocation `spawn_root_and_wait(`*x*`)` does not actually wait for *x* to complete. Instead, it constructs a dummy dependent of *x*, and waits for the dependent's reference count to be decremented. Because `allocate_continuation` forwards this dummy dependent to the continuation, the dummy dependent's reference count is not decremented until the continuation completes.

10.4.3 Scheduler Bypass

Scheduler bypass is an improvement where you directly specify the next task to run. Continuation-passing style often opens up an opportunity for scheduler bypass. For example, in the continuation-passing example, it turns out that once `FibTask::execute()` returns, by the "getting" rules described in Section 10.3, task "a" is always the next task taken from the ready pool. Putting the task into the ready pool and then getting it back out incurs some overhead that can be avoided. To do

this, method execute() should not spawn the task, but instead return a pointer to it as the result. The following example shows the necessary changes:

```
struct FibTask: public task {
    ...
    task* execute() {
        if( n<CutOff ) {
            *sum = SerialFib(n);
            return NULL;
        } else {
            FibContinuation& c =
                *new( allocate_continuation() ) FibContinuation(sum);
            FibTask& a = *new( c.allocate_child() ) FibTask(n-2,&c.x);
            FibTask& b = *new( c.allocate_child() ) FibTask(n-1,&c.y);
            // Set ref_count to "two children".
            c.set_ref_count(2);
            c.spawn( b );
            c.spawn( a );
            return NULL;
            return &a;
        }
    }
};
```

10.4.4 Recycling

Not only can you bypass the scheduler, you might also bypass task allocation and deallocation. The opportunity frequently arises for recursive tasks that do scheduler bypass, because the child is handed by a return statement at just the moment the parent completes. The following code shows the changes required to implement recycling in the scheduler-bypass example:

```
struct FibTask: public task {
    const long n;
    long* const sum;
    ...
    task* execute() {
        if( n<CutOff ) {
            *sum = SerialFib(n);
            return NULL;
        } else {
            FibContinuation& c =
                *new( allocate_continuation() ) FibContinuation(sum);
            FibTask& a = *new( c.allocate_child() ) FibTask(n-2,&c.x);
            FibTask& b = *new( c.allocate_child() ) FibTask(n-1,&c.y);
            recycle_as_child_of(c);
            n -= 2;
            sum = &c.x;
            // Set ref_count to "two children".
            c.set_ref_count(2);
            c.spawn( b );
            return &a;
            return this;
```

```
        }
    }
};
```

The child that was previously called a is now the recycled `this`. The call `recycle_as_child_of(c)` has several effects:

- It marks `this` as to *not* be automatically destroyed when `execute()` returns.
- It sets the depth of `this` to be 1 more than the depth of `c`.
- It sets the dependent of `this` to be `c`. To prevent reference-counting problems, `recycle_as_child_of` has a prerequisite that this must have a NULL dependent. This is the case after `allocate_continuation` occurs.

When recycling, ensure that the original task's fields are not used after the task might start running. The example uses the scheduler bypass trick to ensure this. You can spawn the recycled task instead, as long as none of its fields are used after the spawning. This restriction applies even to any `const` fields, because after spawning the task might run and be destroyed before the parent progresses any further.

NOTE: A similar method, `task::recycle_as_continuation()`recycles a task as a continuation instead of a child.

10.4.5 Empty Tasks

You might need a task that does not do anything but wait for its children to complete. The header `task.h` defines class `empty_task` for this purpose. Its definition is as follows:

```
// Task that does nothing. Useful for synchronization.
class empty_task: public task {
    /*override*/ task* execute() {
        return NULL;
    }
};
```

A good example of `empty_task` in action is provided in `tbb/parallel_for.h`, in method `start_for::execute()`. The code there uses continuation-passing style. It creates two child tasks, and uses an `empty_task` as the continuation when the child tasks complete. The top level routine `parallel_for` (in `tbb/parallel_for.h`) waits on the root

10.4.6 Lazy Copying

It can be useful to copy a data structure only when another thread steals a task. For example, `tbb/parallel_reduce.h` uses a method, `start_reduce::execute()`, that forks the "loop body" object you provided only when the thread runs a stolen task. The forking permits the thief to run locally afterwards until it is done and joins its

result to the original thread's result. Because the fork/join incur some overhead, they are only worth doing when stealing occurs.

Method `task::is_stolen_task` provides a way to detect stealing. Call it on a running task, typically by the task itself. Informally, it returns true if the task is stolen. Formally, it returns true if the thread that owns the task is not the thread that owns the thread's dependent. For the usual fork-join task patterns, the informal and formal definitions have the same effect, because usually when a task is created, it is created by the thread that owns its dependent. For example, the dependent is typically the parent or a continuation created by the parent.

The exception to the rule can occur if method `allocate_additional_child_of(t)` is used. This method can be used by a task to create a child of *another* task *t*, even if *t* already has running children. This is in contrast to method `allocate_child`, to which a call must finish before any sibling starts running. The two methods are distinct because you pay some overhead for the flexibility of `allocate_additional_child_of`. For example, in `tbb/parallel_do.h`, method `allocate_additional_child_of` is used by running child tasks to create new siblings. In that case, `task::is_stolen_task` will report true unless the child is stolen by the thread that is running *t*. The name `task::might_be_running_on_different_thread_than_dependent()` would be more accurate but tedious.

10.5 Task Scheduler Summary

The task scheduler works most efficiently for fork-join parallelism with lots of forks, so that the task-stealing can cause sufficient breadth-first behavior to occupy threads, which then conduct themselves in a depth-first manner until they need to steal more work.

The task scheduler is not the simplest possible scheduler because it is designed for speed. If you need to use it directly, it may be best to hide it behind a higher-level interface, as the templates `parallel_for`, `parallel_reduce`, etc. do. Some of the details to remember are:

- Always use new(*allocation_method*) *T* to allocate a `task`, where *allocation_method* is one of the allocation methods of class `task`. Do not create local or file-scope instances of a `task`.
- All siblings should be allocated before any start running, unless you are using `allocate_additional_child_of`.
- Exploit continuation passing, scheduler bypass, and task recycling to squeeze out maximum performance.
- If a task completes, and was not marked for re-execution, it is automatically destroyed. Also, its dependent's reference count is decremented, and if it hits zero, the dependent is automatically spawned.

Appendix A Costs of Time Slicing

Time slicing enables there to be more logical threads than physical threads. Each logical thread is serviced for a *time slice* by a physical thread. If a thread runs longer than a time slice, as most do, it relinquishes the physical thread until it gets another turn. This appendix details the costs incurred by time slicing.

The most obvious is the time for *context switching* between logical threads. Each context switch requires that the processor save all its registers for the previous logical thread that it was executing, and load its registers for the next logical thread that it runs.

A more subtle cost is *cache cooling*. Processors keep recently accessed data in cache memory, which is very fast, but also relatively small compared to main memory. When the processor runs out of cache memory, it has to evict items from cache and put them back into main memory. Typically, it chooses the least recently used items in the cache. (The reality of set-associative caches is a bit more complicated, but this is not a cache primer.) When a logical thread gets its time slice, as it references a piece of data for the first time, this data will be pulled into cache, taking hundreds of cycles. If is referenced frequently enough to not be evicted, each subsequent reference will find it in cache, and only take a few cycles. Such data is called “hot in cache”. Time slicing undoes this, because if a thread A finishes its time slice, and subsequently thread B runs on the same physical thread, B will tend to evict data that was hot in cache for A, unless both threads need the data. When thread A gets its next time slice, it will need to reload evicted data, at the cost of hundreds of cycles for each cache miss. Or worse yet, the next time slice for thread A may be on a different physical thread that has a different cache altogether.

Another cost is *lock preemption.* This happens if a thread acquires a lock on a resource, and its time slice runs out before it releases the lock. No matter how short a time the thread intended to hold the lock, it is now going to hold it for at least as long as it takes for its next turn at a time slice to come up. Any other threads waiting on the lock either pointlessly busy-wait, or lose the rest of their time slice. The effect is called *convoying,* because the threads end up “bumper to bumper” waiting for the preempted thread in front to resume driving.

Appendix B Mixing With Other Threading Packages

Intel® Threading Building Blocks can be mixed with other threading packages. No special effort is required to use the containers, synchronization primitives, or atomic operations with other threading packages. However, using the parallel algorithms or task scheduler requires extra effort, because each thread that uses one of those features must construct its own `task_scheduler_init` object that is live while the feature is in use.

Here is an example that parallelizes an outer loop with OpenMP and an inner loop with Intel® Threading Building Blocks.

```
int M, N;

struct InnerBody {
    ...
};

void TBB_NestedInOpenMP() {
#pragma omp parallel
    {
        task_scheduler_init init;
#pragma omp for
        for( int i=0; i<M; ++j ) {
            parallel_for( blocked_range<int>(0,N,10), InnerBody(i) );
        }
    }
}
```

The details of `InnerBody` are omitted for brevity. What is important is the placement of the `task_scheduler_init` declaration. The `#pragma omp parallel` causes the OpenMP to create a team of threads, and each thread executes the block statement associated with the pragma. Each thread must construct its own `task_scheduler_init` inside the block. The `#pragma omp for` indicates that the compiler should use the previously created thread team to execute the loop in parallel. Because this pragma does not create threads, it has no corresponding `task_scheduler_init` declaration.

Here is the same example written using POSIX* Threads.

```
int M, N;

struct InnerBody {
    ...
};
```

```
void* OuterLoopIteration( void* args ) {
    task_scheduler_init init;
    int i = (int)args;
    parallel_for( blocked_range<int>(0,N,10), InnerBody(i) );
}

void TBB_NestedInPThreads() {
    std::vector<pthread_t> id( M );
    // Create thread for each outer loop iteration
    for( int i=0; i<M; ++i )
        pthread_create( &id[i], NULL, OuterLoopIteration, NULL );
    // Wait for outer loop threads to finish
    for( int i=0; i<M; ++i )
        pthread_join( &id[i], NULL );
}
```

References

[1] "Memory Consistency & .NET", Arch D. Robison, Dr. Dobb's Journal, April 2003.

[2] A Formal Specification of Intel® Itanium® Processor Family Memory Ordering, Intel Corporation, October 2002.