Intel® oneAPI Threading Building Blocks Developer Guide and API Reference

Contents

Intel® oneAPI Threading Building Blocks (oneTBB) 1

This document contains information about Intel® oneAPI Threading Building Blocks (oneTBB). oneTBB is a flexible performance library that can be found in the Intel® oneAPI Base Toolkit or as a stand-alone product. More information and specifications can be found on the Intel® oneAPI Threading Building Blocks (oneTBB) main page.

Documentation for older versions of oneTBB is available for download only. For a list of available documentation downloads by product version, see these pages:

- Download Documentation for Intel Parallel Studio XE
- Download Documentation for Intel System Studio

The following are some important topics for the `novice user`:

- Get Started with oneTBB gives you a brief explanation of what oneTBB is.
- oneTBB Benefits describes how oneTBB differs from typical threading packages.
- Package Contents describes dynamic library files and header files for Windows*, Linux*, and macOS* operating systems used in oneTBB.

The following is an important topic for the `experienced user`:

Migrating from Threading Building Blocks (TBB) describes how to migrate from TBB to oneTBB.

Getting Help and Support

Getting Technical Support

If you did not register your Intel® software product during installation, please do so now at the Intel® Software Development Products Registration Center. Registration entitles you to free technical support, product updates and upgrades for the duration of the support term.

For general information about Intel technical support, product updates, user forums, FAQs, tips and tricks and other support questions, go to: http://www.intel.com/software/products/support/.

Notational Conventions

The following conventions may be used in this document.

Convention	Explanation	Example
Italic	Used for introducing new terms, denotation of terms, placeholders, or titles of manuals.	The filename consists of the *basename* and the *extension*.
`Monospace`	Indicates directory paths and filenames, commands and command line options, function names, methods, classes, data structures in body text, source code.	`ippsapi.h\alt\include` Use the okCreateObjs() function to… `printf("hello, world\n");`
[ ]	Items enclosed in brackets are optional.	Fa[c] Indicates Fa or Fac.
{ | }	Braces and vertical bars indicate the choice of one item from a selection of two or more items.	X{K | W | P} Indicates XK, XW, or XP.
"[" "]" "{" | " }" "|"	Writing a metacharacter in quotation marks negates the syntactical meaning stated above; | the character is taken as a literal.	"[" X "]" [Y] Denotes the letter X enclosed in brackets, optionally followed by the letter Y.
…	The ellipsis indicates that the previous item can be repeated several times.	`filename` … Indicates that one or more filenames can be specified.
,…	The ellipsis preceded by a comma indicates that the previous item can be repeated several times, | separated by commas.	`word` ,… Indicates that one or more words can be specified. If more than one word is specified, the words are comma-separated.

Class members are summarized by informal class declarations that describe the class as it seems to clients, not how it is actually implemented. For example, here is an informal declaration of class `Foo`:

```
class Foo {
public:
    int x();
    int y;
    ~Foo();
};
```

The actual implementation might look like:

```
namespace internal {
    class FooBase  {
    protected:
        int x();
    };


    class Foo_v3: protected FooBase {
    private:
        int internal_stuff;
    public:
        using FooBase::x;
        int y;
    };
}


typedef internal::Foo_v3 Foo;
```

The example shows two cases where the actual implementation departs from the informal declaration:

- `Foo` is actually a typedef to `Foo_v3`.
- Method `x()` is inherited from a protected base class.
- The destructor is an implicit method generated by the compiler.

The informal declarations are intended to show you what you need to know to use the class without the distraction of irrelevant clutter particular to the implementation.

Introduction

Intel® oneAPI Threading Building Blocks (oneTBB) is a library that supports scalable parallel programming using standard ISO C++ code. It does not require special languages or compilers. It is designed to promote scalable data parallel programming. Additionally, it fully supports nested parallelism, so you can build larger parallel components from smaller parallel components. To use the library, you specify tasks, not threads, and let the library map tasks onto threads in an efficient manner.

Many of the library interfaces employ generic programming, in which interfaces are defined by requirements on types and not specific types. The C++ Standard Template Library (STL) is an example of generic programming. Generic programming enables oneTBB to be flexible yet efficient. The generic interfaces enable you to customize components to your specific needs.

NOTE Intel® oneAPI Threading Building Blocks (oneTBB) requires C++11 standard compiler support.

The net result is that oneTBB enables you to specify parallelism far more conveniently than using raw threads, and at the same time can improve performance.

NOTE
Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.intel.com/PerformanceIndex. Notice revision #20201201

oneTBB Benefits

Intel® oneAPI Threading Building Blocks (oneTBB) is a library that helps you leverage multi-core performance without having to be a threading expert. Typically you can improve performance for multi-core processors by implementing the key points explained in the early sections of the Developer Guide. As your expertise grows, you may want to dive into more complex subjects that are covered in advanced sections.

There are a variety of approaches to parallel programming, ranging from using platform-dependent threading primitives to exotic new languages. The advantage of oneTBB 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:

- **oneTBB enables you to specify logical paralleism 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 oneTBB run-time library automatically maps logical parallelism onto threads in a way that makes efficient use of processor resources.
- **oneTBB 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, oneTBB focuses on the particular goal of parallelizing computationally intensive work, delivering higher-level, simpler solutions.
- **oneTBB 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.
- **oneTBB 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, oneTBB 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.
- **oneTBB relies on generic programming**. Traditional libraries specify interfaces in terms of specific types or base classes. Instead, oneAPI 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 oneTBB templates specify requirements on types, not particular types, and thus adapt to different data representations. Generic programming enables oneTBB to deliver high performance algorithms with broad applicability.

oneTBB Developer Guide

Intel® oneAPI Threading Building Blocks (oneTBB)

Package Contents

Intel® oneAPI Threading Building Blocks (oneTBB) includes dynamic library files and header files for Windows*, Linux* and macOS* operating systems as described in this section.

- Debug Versus Release Libraries
- Scalable Memory Allocator
- Windows*
- Linux*
- macOS*

Debug Versus Release Libraries

The following table details the Intel® oneAPI Threading Building Blocks (oneTBB) dynamic shared libraries that come in debug and release versions.

Library	Description	When to Use
`tbb_debugtbbmalloc_debugtbbmalloc_proxy_debugtbbbind_debug`	These versions have extensive internal checking for correct use of the library.	Use with code that is compiled with the macro `TBB_USE_DEBUG` set to 1.
`tbbtbbmalloctbbmalloc_proxytbbbind`	These versions deliver top performance. They eliminate most checking for correct use of the library.	Use with code compiled with `TBB_USE_DEBUG` 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.

oneTBB supports Intel® Inspector, Intel® VTune™ Profiler and Intel® Advisor. Full support of these tools requires compiling with macro `TBB_USE_PROFILING_TOOLS=1`. That symbol defaults to 1 in the following conditions:

- When `TBB_USE_DEBUG=1`.
- On the Microsoft Windows* operating system, when `_DEBUG=1`.

The oneTBB API Reference section explains the default values in more detail.

Caution The instrumentation support for Intel® Inspector becomes live after the first initialization of the task library. If the library components are used before this initialization occurs, Intel® Inspector may falsely report race conditions that are not really races.

Scalable Memory Allocator

Both the debug and release versions of Intel® oneAPI Threading Building Blocks (oneTBB) consists of 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* OS are `tbb<version>.dll` and `tbbmalloc.dll` respectively. Applications may choose to use only the general library, or only the scalable memory allocator, or both. See the links below for more information on memory allocation.

Windows*

This section uses <*tbb_install_dir*> to indicate the top-level installation directory. The following table describes the subdirectory structure for Windows*, relative to <*tbb_install_dir*>.

Item	Location	Environment Variable
Header files	`include\oneapi\tbb.hinclude \oneapi\tbb\*.h`	`INCLUDE`
.lib files	`lib\<arch>\vc<vcversion> \<lib><variant><version>.lib`	`LIB`
.dll files	`redist\<arch>\vc<vcversion> \<lib><variant><version>.dll`	`PATH`
.pdb files	Same as corresponding `.dll` file.	

where

- `<arch>` - `ia32` or `intel64`
- `<lib>` - `tbb`, `tbbmalloc`, `tbbmalloc_proxy` or `tbbbind`
- `<vcversion>`
  - `14` - use for dynamic linkage with the CRT
  - `14_uwp` - use for Windows 10 Universal Windows applications
  - `14_uwd` - use for Universal Windows Drivers
  - `_mt` - use for static linkage with the CRT
- `<variant>` - `_debug` or empty
- `<version>` - binary version

The last column shows which environment variables are used by the Microsoft* Visual C++* or Intel® C++ Compiler Classic or Intel® oneAPI DPC++/C++ Compiler 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.

NOTE Microsoft* C/C++ run-time libraries come in static and dynamic forms. Either can be used with oneTBB. Linking to the oneTBB library is always dynamic.

Linux*

This section uses <*tbb_install_dir*> to indicate the top-level installation directory. The following table describes the subdirectory structure for Linux*, relative to <*tbb_install_dir*>

Item	Location	Environment Variable
Header files	`include/oneapi/tbb.hinclude/oneapi/tbb/*.h`	`CPATH`
Shared libraries	`lib/<arch>/<lib><variant>.so.<version>`	`LIBRARY_PATHLD_LIBRARY_PATH`

where

- `<arch>` - `ia32` or `intel64`
- `<lib>` - `libtbb`, `libtbbmalloc`, `libtbbmalloc_proxy` or `libtbbbind`
- `<variant>` - `_debug` or empty
- `<version>` - binary version in a form of `<major>.<minor>`

macOS*

This section uses <*install_dir*> to indicate the top-level installation directory. The following table describes the subdirectory structure for macOS*, relative to <*install_dir*>.

Item	Location	Environment Variable
Header files	`include/oneapi/tbb.hinclude/oneapi/tbb/*.h`	`CPATH`
Shared libraries	`lib/<lib><variant>.<version>.dylib`	`LIBRARY_PATHDYLD_LIBRARY_PATH`

where

- `<lib>` - `libtbb`, `libtbbmalloc` or `libtbbmalloc_proxy`
- `<variant>` - `_debug` or empty
- `<version>` - binary version in a form of `<major>.<minor>`

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.

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

When compiling oneTBB programs, be sure to link in the oneTBB shared library, otherwise undefined references will occur. The following table shows compilation commands that use the debug version of the library. Remove the "`_debug`" portion to link against the production version of the library.

Operating System	Command line
Windows* OS	`icl /MD example.cpp tbb_debug.lib`
Linux* OS	`icc example.cpp -ltbb_debug`

Initializing and Terminating the Library

Intel® oneAPI Threading Building Blocks (oneTBB) automatically initializes the task scheduler. The initialization process is involved when a thread uses task scheduling services the first time, for example any parallel algorithm, flow graph or task group. The termination happens when the last such thread exits.

Explicit Library Finalization

oneTBB supports an explicit library termination as a preview feature. The `oneapi::tbb::finalize` function called with an instance of class `oneapi::tbb::task_scheduler_handle` blocks the calling thread until all worker threads implicitly created by the library have completed. If waiting for thread completion is not safe, e.g. may result in a deadlock or called inside a task, a parallel algorithm, or a flow graph node, the method fails.

If you know how many active `oneapi::tbb::task_scheduler_handle` instances exist in the program, it is recommended to call `oneapi::tbb::release` function on all but the last one, then call `oneapi::tbb::finalize` for the last instance.

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 `oneapi::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.

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

using namespace oneapi::tbb;
```

```
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)
    {}
};
```

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

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 Advanced Topic: Other Kinds of Iteration Spaces.

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 "oneapi/tbb.h"


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

The `blocked_range` constructed here represents the entire iteration space from 0 to n-1, which `parallel_for` divides into subspaces for each processor. The general form of the constructor is `blocked_range<T>(begin,end,grainsize)`. The `T` specifies the value type. The arguments `begin` and `end` specify the iteration space STL-style as a half-open interval [`begin`,`end`). The argument *grainsize* is explained in the Controlling Chunking section. The example uses the default grainsize of 1 because by default `parallel_for` applies a heuristic that works well with the default grainsize.

- Lambda Expressions
- Automatic Chunking
- Controlling Chunking

- Bandwidth and Cache Affinity
- Partitioner Summary

Lambda Expressions

C++11 lambda expressions make the Intel® oneAPI Threading Building Blocks (oneTBB) `parallel_for` much easier to use. A lambda expression lets the compiler do the tedious work of creating a function object.

Below is the example from the previous section, rewritten with a lambda expression. The lambda expression, replaces both the declaration and construction of function object `ApplyFoo` in the example of the previous section.

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


using namespace oneapi::tbb;


void ParallelApplyFoo( float* a, size_t n ) {
   parallel_for( blocked_range<size_t>(0,n),
      [=](const blocked_range<size_t>& r) {
                      for(size_t i=r.begin(); i!=r.end(); ++i)
                          Foo(a[i]);
                  }
    );
}
```

The [=] introduces the lambda expression. The expression creates a function object very similar to `ApplyFoo`. When local variables like `a` and `n` are declared outside the lambda expression, but used inside it, they are "captured" as fields inside the function object. The [=] specifies that capture is by value. Writing [&] instead would capture the values by reference. After the [=] is the parameter list and definition for the `operator()` of the generated function object. The compiler documentation says more about lambda expressions and other implemented C++11 features. It is worth reading more complete descriptions of lambda expressions than can fit here, because lambda expressions are a powerful feature for using template libraries in general.

C++11 support is off by default in the compiler. The following table shows the option for turning it on.

Environment	**Intel® C++ Compiler Classic**	**Intel® oneAPI DPC++/C++ Compiler**
Windows* OS systems	`icl /Qstd=c++11 foo.cpp`	`icx /Qstd=c++11 foo.cpp`
Linux* OS systems	`icc -std=c++11 foo.cpp`	`icx -std=c++11 foo.cpp`

For further compactness, oneTBB has a form of `parallel_for` expressly for parallel looping over a consecutive range of integers. The expression `parallel_for(first,last,step,f)` is like writing `for(auto i=first; i<last; i+=step)f(i)` except that each f(i) can be evaluated in parallel if resources permit. The `step` parameter is optional. Here is the previous example rewritten in the compact form:

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


using namespace oneapi::tbb;


#pragma warning(disable: 588)

```

```
void ParallelApplyFoo(float a[], size_t n) {
    parallel_for(size_t(0), n, [=](size_t i) {Foo(a[i]);});
}
```

The compact form supports only unidimensional iteration spaces of integers and the automatic chunking feature detailed on the following section.

Automatic Chunking

A parallel loop construct incurs overhead cost for every chunk of work that it schedules. Intel® oneAPI Threading Building Blocks (oneTBB) chooses chunk sizes automatically, depending upon load balancing needs. The heuristic attempts to limit overheads while still providing ample opportunities for load balancing.

Caution Typically a loop needs to take at least a million clock cycles to make it worth using `parallel_for`. For example, a loop that takes at least 500 microseconds on a 2 GHz processor might benefit from `parallel_for`.

The default automatic chunking is recommended for most uses. As with most heuristics, however, there are situations where controlling the chunk size more precisely might yield better performance.

Controlling Chunking

Chunking is controlled by a *partitioner* and a *grainsize.* To gain the most control over chunking, you specify both.

- Specify `simple_partitioner()` as the third argument to `parallel_for`. Doing so turns off automatic chunking.
- Specify the grainsize when constructing the range. The thread argument form of the constructor is `blocked_range<T>(begin,end,grainsize)`. The default value of `grainsize` is 1. It is in units of loop iterations per chunk.

If the chunks are too small, the overhead may exceed the performance advantage.

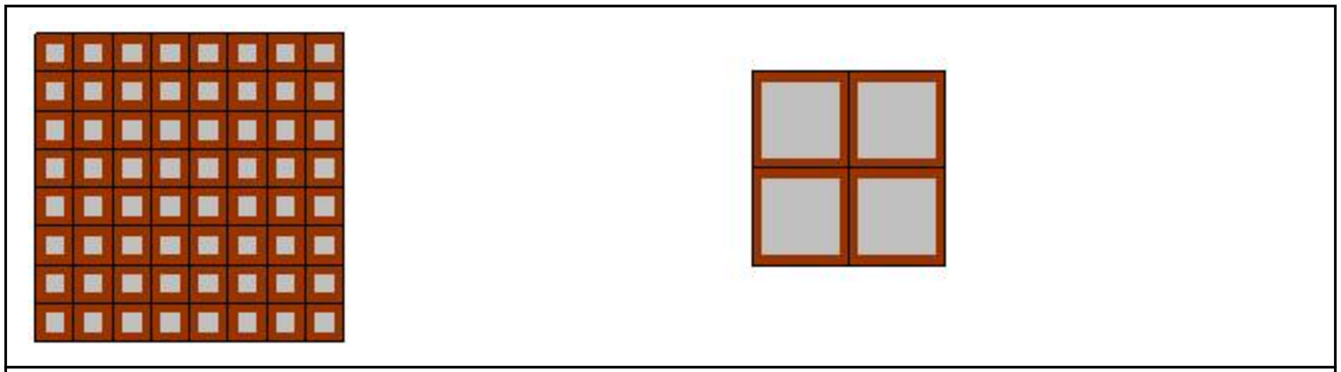
The following code is the last example from parallel_for, modified to use an explicit grainsize `G`.

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


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

The grainsize sets a minimum threshold for parallelization. The `parallel_for` in the example invokes `ApplyFoo::operator()` on chunks, possibly of different sizes. Let *chunksize* be the number of iterations in a chunk. Using `simple_partitioner` guarantees that [G/2] <= *chunksize* <= G.

There is also an intermediate level of control where you specify the grainsize for the range, but use an `auto_partitioner` and `affinity_partitioner`. An `auto_partitioner` is the default partitioner. Both partitioners implement the automatic grainsize heuristic described in Automatic Chunking. An `affinity_partitioner` implies an additional hint, as explained later in Section Bandwidth and Cache Affinity. Though these partitioners may cause chunks to have more than G iterations, they never generate chunks with less than [G/2] iterations. Specifying a range with an explicit grainsize may occasionally be useful to prevent these partitioners from generating wastefully small chunks if their heuristics fail.

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

The above figure 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.

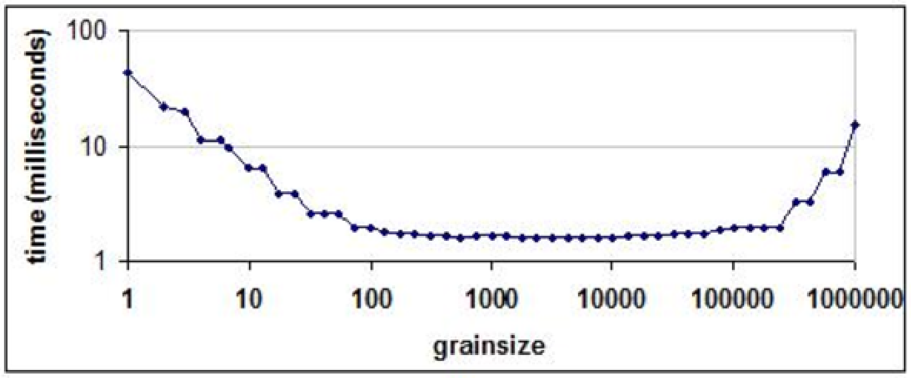
A rule of thumb is that `grainsize` iterations of `operator()` should take at least 100,000 clock cycles to execute. For example, if a single iteration takes 100 clocks, then the `grainsize` needs to be at least 1000 iterations. When in doubt, do the following experiment:

1. Set the `grainsize` parameter higher than necessary. The grainsize is specified in units of loop iterations. If you have no idea of how many clock cycles an iteration might take, start with `grainsize`=100,000. The rationale is that each iteration normally requires at least one clock per iteration. In most cases, step 3 will guide you to a much smaller value.
2. Run your algorithm.
3. Iteratively halve the `grainsize` parameter and see how much the algorithm slows down or speeds up as the value decreases.

A drawback of setting a grainsize too high is that it can reduce parallelism. For example, if the grainsize is 1000 and the loop has 2000 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.

The next figure 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.



Wall Clock 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.

NOTE
Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.intel.com/PerformanceIndex. Notice revision #20201201

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 and tries to distribute the data uniformly among threads. 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 (and especially if the number of threads is not a power of two). If only two threads are available, the default scheduling in Intel® oneAPI Threading Building Blocks (oneTBB) usually provides sufficient cache affinity.

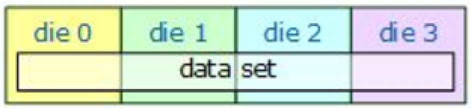
The following code shows how to use `affinity_partitioner`.

```
#include "oneapi/tbb.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 );
}
```

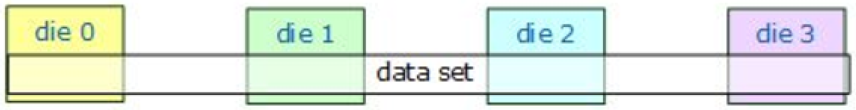
In the example, the `affinity_partitioner` object `ap` lives between loop iterations. It remembers where iterations of the loop ran, so that each iteration can be handed to the same thread that executed it before. The example code gets the lifetime of the partitioner right by declaring the `affinity_partitioner` 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. The following figure shows the situations.

Benefit of Affinity Determined by Relative Size of Data Set and Cache



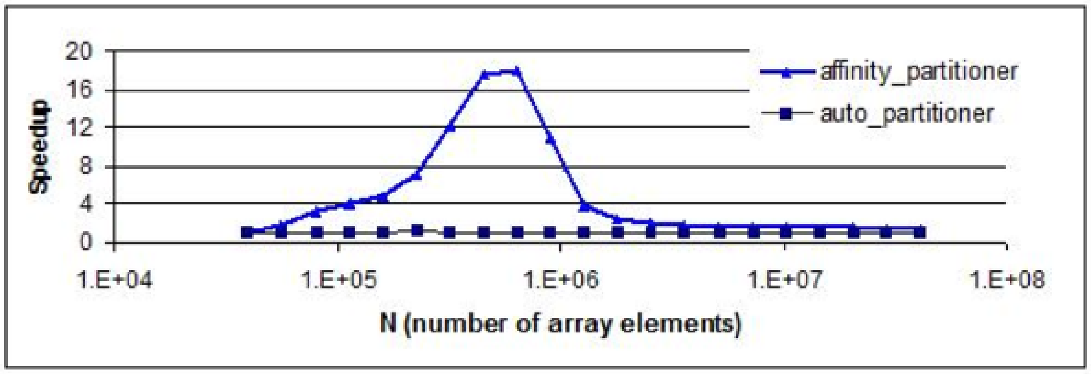
Benefit from affinity.



No benefit from affinity.

The next figure 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.

Improvement from Affinity Dependent on Array Size



NOTE
Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Partitioner Summary

The parallel loop templates `parallel_for` and `parallel_reduce` take an optional *partitioner* argument, which specifies a strategy for executing the loop. The following table summarizes partitioners and their effect when used in conjunction with `blocked_range`.

Partitioner	Description	When Used with **`blocked_range(i,j,g)`**
`simple_partitioner`	Chunksize bounded by grain size.	`g/2 ≤ chunksize ≤ g`
`auto_partitioner` (default)	Automatic chunk size.	`g/2 ≤ chunksize`

Partitioner	Description	When Used with `blocked_range(i,j,g)`
`affinity_partitioner`	Automatic chunk size, cache affinity and uniform distribution of iterations.	`g/2 ≤ chunksize`
`static_partitioner`	Deterministic chunk size, cache affinity and uniform distribution of iterations without load balancing.	`max(g/3, problem_size/num_of_resources) ≤ chunksize`

An `auto_partitioner` is used when no partitioner is specified. In general, the `auto_partitioner` or `affinity_partitioner` should be used, because these tailor the number of chunks based on available execution resources. `affinity_partitioner` and `static_partitioner` may take advantage of `Range` ability to split in a given ratio (see "Advanced Topic: Other Kinds of Iteration Spaces") for distributing iterations in nearly equal chunks between computing resources.

`simple_partitioner` can be useful in the following situations:

- The subrange size for `operator()` must 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.
- A large subrange might use cache inefficiently. For example, suppose the processing of a subrange involves repeated sweeps over the same memory locations. Keeping the subrange below a limit might enable the repeatedly referenced memory locations to fit in cache.
- You want to tune to a specific machine.

parallel_reduce

A loop can do a 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 );
    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`:

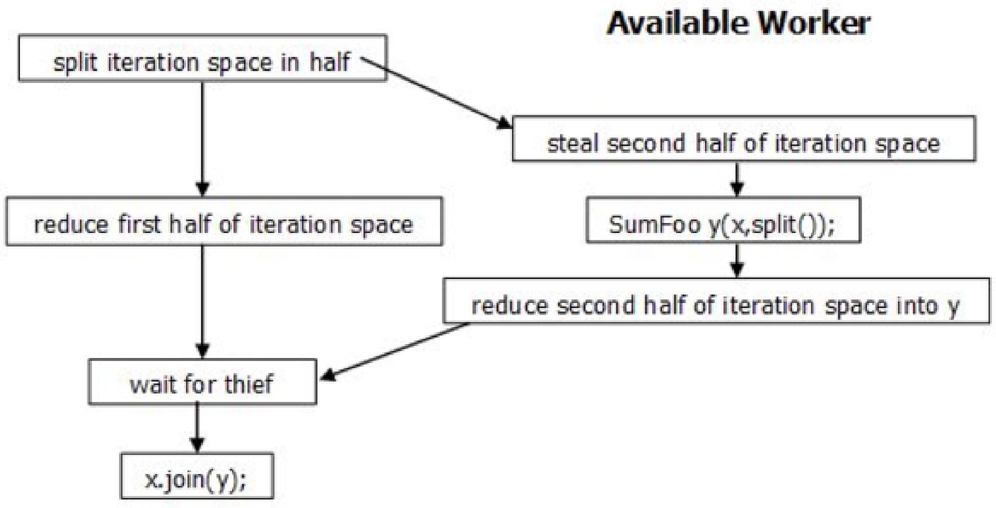
```
class SumFoo {
    float* my_a;
public:
    float my_sum;
    void operator()( const blocked_range<size_t>& r ) {
        float *a = my_a;
        float sum = my_sum;
        size_t end = r.end();
        for( size_t i=r.begin(); i!=end; ++i )
            sum += Foo(a[i]);
        my_sum = sum;
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


    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)
    {}
};
```

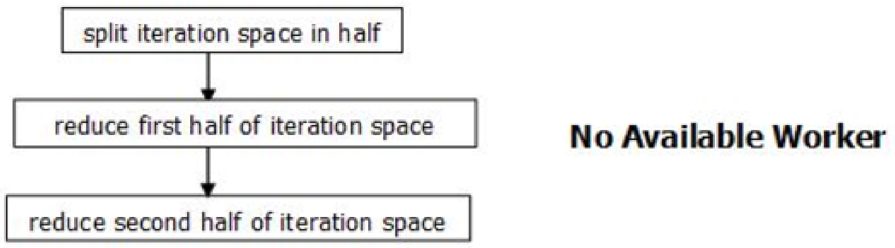
Note the differences with class `ApplyFoo` from parallel_for. First, `operator()` is *not*`const`. This is because it must update SumFoo::my_sum. Second, `SumFoo` has a *splitting constructor* and a method `join` that must be present for `parallel_reduce` to work. The splitting constructor takes as arguments 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` invokes the splitting constructor to create a subtask for the worker. When the subtask completes, `parallel_reduce` uses method `join` to accumulate the result of the subtask. The graph at the top of the following figure shows the split-join sequence that happens when a worker is available:

Graph of the Split-join Sequence


Available Worker
split iteration space in half
steal second half of iteration space
reduce first half of iteration space
SumFoo y(x,split());
reduce second half of iteration space into y
wait for thief
x.join(y);


split iteration space in half
reduce first half of iteration space
No Available Worker
reduce second half of iteration space

An arrows in the above figure indicate 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 Since split/join are not used if workers are unavailable, `parallel_reduce` does not necessarily do recursive splitting.

Caution Since the same body might be used to accumulate multiple subranges, it is critical that `operator()` not discard earlier accumulations. The code below shows an incorrect definition of `SumFoo::operator()`.

```
class SumFoo {
    ...
public:
    float my_sum;
    void operator()( const blocked_range<size_t>& r ) {
        ...
        float sum = 0;  // WRONG - should be 'sum = my_sum".
        ...
        for( ... )
```

```
            sum += Foo(a[i]);
        my_sum = sum;
    }
    ...
};
```

With the mistake, the body returns a partial sum for the last subrange instead of all subranges to which `parallel_reduce` applies it.

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.

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<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 );


 return mif.index_of_min;
}
```

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® oneAPI Threading Building Blocks (oneTBB) to define your own iteration space objects. The object must specify how it can be split into subspaces by providing a basic splitting constructor, an optional proportional splitting constructor, and two predicate methods. If your class is called `R`, the methods and constructors should 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;
    // Splits r into subranges r and *this
    R( R& r, split );
    // (optional) Splits r into subranges r and *this in proportion p
    R( R& r, proportional_split p );
    ...
};
```

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 basic splitting constructor should take two arguments:

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

The second argument is not used; it serves only to distinguish the constructor from an ordinary copy constructor. The basic splitting constructor should attempt to split `r` roughly into two halves, and update `r` to be the first half, and set the constructed object as the second half.

Unlike the basic splitting constructor, the proportional splitting constructor is optional and takes the second argument of type `oneapi::tbb::proportional_split`. The type has methods `left` and `right` that return the values of the proportion. These values should be used to split `r` accordingly, so that the updated `r` corresponds to the left part of the proportion, and the constructed object corresponds to the right part.

Both splitting constructors should guarantee that the updated `r` part and the constructed object are not empty. The parallel algorithm templates call the splitting constructors on `r` only if `r.is_divisible` is true.

The iteration space does not have to be linear. Look at `oneapi/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.

Parallelizing Complex Loops

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

- Cook Until Done: parallel_for_each
- Working on the Assembly Line: parallel_pipeline
  - Using Circular Buffers
  - Throughput of pipeline
  - Non-Linear Pipelines
- Summary of Loops and Pipelines

Cook Until Done: parallel_for_each

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 `oneapi::tbb::parallel_for_each`.

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_for_each` 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_for_each`. To do so, define an object with a `const` qualified `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_for_each( list.begin(), list.end(), ApplyFoo() );
}
```

An invocation of `parallel_for_each` 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_for_each` 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_for_each` can acquire work scalably.

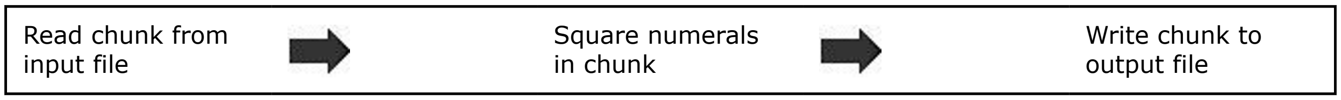
- The iterators can be random-access iterators.
- The body argument to `parallel_for_each`, if it takes a second argument *feeder* of type `parallel_for_each<Item>&`, can add more work by calling `feeder.add(item)`. For example, suppose processing a node in a tree is a prerequisite to processing its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

Working on the Assembly Line: parallel_pipeline

Pipelining is a common parallel pattern that mimics a traditional manufacturing assembly line. Data flows through a series of pipeline filters and each filter processes the data in some way. Given an incoming stream of data, some of these filters 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® oneAPI Threading Building Blocks (oneTBB) classes `parallel_pipeline` and filter implement the pipeline pattern. A simple text processing example will be used to demonstrate the usage of `parallel_pipeline` and filter to perform parallel formatting. The example reads a text file, squares each decimal numeral in the text, and writes the modified text to a new file. Below is a picture of the pipeline.

Caution Since the body object provided to the filters of the `parallel_pipline` 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_pipeline`, depending upon whether `operator()` is acting on the original or a copy. As a reminder of this nuance, `parallel_pipeline` requires that the body object's `operator()` be declared `const`.



Assume that the raw file I/O is sequential. The squaring filter can be done in parallel. That is, if you can serially read `n` chunks very quickly, you can transform each of the `n` chunks in parallel, as long as they are written in the proper order to the output file. Though the raw I/O is sequential, the formatting of input and output can be moved to the middle filter, and thus be parallel.

To amortize parallel scheduling overheads, the filters operate on chunks of text. Each input chunk is approximately 4000 characters. Each chunk is represented by an instance of class `TextSlice`:

```
// Holds a slice of text.
/** Instances *must* be allocated/freed using methods herein, because the C++ declaration
   represents only the header of a much larger object in memory. */
class TextSlice {
    // Pointer to one past last character in sequence
    char* logical_end;
    // Pointer to one past last available byte in sequence.
    char* physical_end;
public:
    // Allocate a TextSlice object that can hold up to max_size characters.
    static TextSlice* allocate( size_t max_size ) {
        // +1 leaves room for a terminating null character.
        TextSlice* t = (TextSlice*)oneapi::tbb::tbb_allocator<char>().allocate( sizeof(TextSlice)
+max_size+1 );
        t->logical_end = t->begin();
        t->physical_end = t->begin()+max_size;
        return t;
    }
    // Free this TextSlice object
    void free() {
        oneapi::tbb::tbb_allocator<char>().deallocate((char*)this, sizeof(TextSlice)+
(physical_end-begin())+1);
    }
    // Pointer to beginning of sequence
    char* begin() {return (char*)(this+1);}
    // Pointer to one past last character in sequence
    char* end() {return logical_end;}
    // Length of sequence
    size_t size() const {return logical_end-(char*)(this+1);}
    // Maximum number of characters that can be appended to sequence
    size_t avail() const {return physical_end-logical_end;}
    // Append sequence [first,last) to this sequence.
    void append( char* first, char* last ) {
        memcpy( logical_end, first, last-first );
        logical_end += last-first;
    }
    // Set end() to given value.
    void set_end( char* p ) {logical_end=p;}
};
```

Below is the top-level code for building and running the pipeline. `TextSlice` objects are passed between filters using pointers to avoid the overhead of copying a `TextSlice`.

```
void RunPipeline( int ntoken, FILE* input_file, FILE* output_file ) {
    oneapi::tbb::parallel_pipeline(
        ntoken,
        oneapi::tbb::make_filter<void,TextSlice*>(
            oneapi::tbb::filter::serial_in_order, MyInputFunc(input_file) )
    &
        oneapi::tbb::make_filter<TextSlice*,TextSlice*>(
            oneapi::tbb::filter::parallel, MyTransformFunc() )
    &
        oneapi::tbb::make_filter<TextSlice*,void>(
            oneapi::tbb::filter::serial_in_order, MyOutputFunc(output_file) ) );
}
```

The parameter `ntoken` to method `parallel_pipeline` controls the level of parallelism. Conceptually, tokens flow through the pipeline. In a serial in-order filter, each token must be processed serially in order. In a parallel filter, multiple tokens can by processed in parallel by the filter. If the number of tokens were unlimited, there might be a problem where the unordered filter in the middle keeps gaining tokens because the output filter cannot keep up. This situation typically leads to undesirable resource consumption by the middle filter. The parameter to method `parallel_pipeline` specifies the maximum number of tokens that can be in flight. Once this limit is reached, the pipeline never creates a new token at the input filter until another token is destroyed at the output filter.

The second parameter specifies the sequence of filters. Each filter is constructed by function `make_filter<inputType, outputType>(mode,functor)`.

- The *inputType* specifies the type of values input by a filter. For the input filter, the type is `void`.
- The *outputType* specifies the type of values output by a filter. For the output filter, the type is `void`.
- The *mode* specifies whether the filter processes items in parallel, serial in-order, or serial out-of-order.
- The *functor* specifies how to produce an output value from an input value.

The filters are concatenated with `operator&`. When concatenating two filters, the *outputType* of the first filter must match the *inputType* of the second filter.

The filters can be constructed and concatenated ahead of time. An equivalent version of the previous example that does this follows:

```
void RunPipeline( int ntoken, FILE* input_file, FILE* output_file ) {
    oneapi::tbb::filter<void,TextSlice*> f1( oneapi::tbb::filter::serial_in_order,
                                   MyInputFunc(input_file) );
    oneapi::tbb::filter<TextSlice*,TextSlice*> f2(oneapi::tbb::filter::parallel,
                                        MyTransformFunc() );
    oneapi::tbb::filter<TextSlice*,void> f3(oneapi::tbb::filter::serial_in_order,
                                  MyOutputFunc(output_file) );
    oneapi::tbb::filter<void,void> f = f1 & f2 & f3;
    oneapi::tbb::parallel_pipeline(ntoken,f);
}
```

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. Thus if an item arrives at `MyOutputFunc` out of the order established by `MyInputFunc`, the pipeline automatically delays invoking `MyOutputFunc::operator()` on the item until its predecessors are processed. There is another kind of serial filter, `serial_out_of_order`, that does not preserve order.

The middle filter operates on purely local data. Thus any number of invocations of its functor can run concurrently. Hence it is specified as a parallel filter.

The functors for each filter are explained in detail now. The output functor is the simplest. All it has to do is write a `TextSlice` to a file and free the `TextSlice`.

```
// Functor that writes a TextSlice to a file.
class MyOutputFunc {
    FILE* my_output_file;
public:
    MyOutputFunc( FILE* output_file );
    void operator()( TextSlice* item ) const;
};


MyOutputFunc::MyOutputFunc( FILE* output_file ) :
    my_output_file(output_file)
{
}


void MyOutputFunc::operator()( TextSlice* out ) const {
    size_t n = fwrite( out->begin(), 1, out->size(), my_output_file );
    if( n!=out->size() ) {
        fprintf(stderr,"Can't write into file '%s'\n", OutputFileName);
        exit(1);
    }
    out->free();
}
```

Method `operator()` processes a `TextSlice`. The parameter `out` points to the `TextSlice` to be processed. Since it is used for the last filter of the pipeline, it returns `void`.

The functor for the middle filter is similar, but a bit more complex. It returns a pointer to the `TextSlice` that it produces.

```
// Functor that changes each decimal number to its square.
class MyTransformFunc {
public:
    TextSlice* operator()( TextSlice* input ) const;
};


TextSlice* MyTransformFunc::operator()( TextSlice* input ) const {
    // Add terminating null so that strtol works right even if number is at end of the input.
    *input->end() = '\0';
    char* p = input->begin();
    TextSlice* out = TextSlice::allocate( 2*MAX_CHAR_PER_INPUT_SLICE );
    char* q = out->begin();
    for(;;) {
        while( p<input->end() && !isdigit(*p) )
            *q++ = *p++;
        if( p==input->end() )
            break;
        long x = strtol( p, &p, 10 );
        // Note: no overflow checking is needed here, as we have twice the
        // input string length, but the square of a non-negative integer n
        // cannot have more than twice as many digits as n.
        long y = x*x;
        sprintf(q,"%ld",y);
        q = strchr(q,0);
    }
    out->set_end(q);
    input->free();
    return out;
}
```

The input functor is the most complicated, because it has to ensure that no numeral crosses a boundary. When it finds what could be a numeral crossing into the next slice, it copies the partial numeral to the next slice. Furthermore, it has to indicate when the end of input is reached. It does this by invoking method `stop()` on a special argument of type `flow_control`. This idiom is required for any functor used for the first filter of a pipline.

```
TextSlice* next_slice = NULL;


class MyInputFunc {
public:
    MyInputFunc( FILE* input_file_ );
    MyInputFunc( const MyInputFunc& f ) : input_file(f.input_file) { }
    ~MyInputFunc();
    TextSlice* operator()( oneapi::tbb::flow_control& fc ) const;
private:
    FILE* input_file;
};


MyInputFunc::MyInputFunc( FILE* input_file_ ) :
    input_file(input_file_) { }


MyInputFunc::~MyInputFunc() {
}


TextSlice* MyInputFunc::operator()( oneapi::tbb::flow_control& fc ) const {
    // Read characters into space that is available in the next slice.
    if( !next_slice )
        next_slice = TextSlice::allocate( MAX_CHAR_PER_INPUT_SLICE );
    size_t m = next_slice->avail();
    size_t n = fread( next_slice->end(), 1, m, input_file );
    if( !n && next_slice->size()==0 ) {
        // No more characters to process
        fc.stop();
        return NULL;
    } else {
        // Have more characters to process.
        TextSlice* t = next_slice;
        next_slice = TextSlice::allocate( MAX_CHAR_PER_INPUT_SLICE );
        char* p = t->end()+n;
        if( n==m ) {
            // Might have read partial number.
            // If so, transfer characters of partial number to next slice.
            while( p>t->begin() && isdigit(p[-1]) )
                --p;
            assert(p>t->begin(),"Number too large to fit in buffer.\n");
            next_slice->append( p, t->end()+n );
        }
        t->set_end(p);
        return t;
    }
}
```

The copy constructor must be defined because the functor is copied when the underlying `oneapi::tbb::filter_t` is built from the functor, and again when the pipeline runs.

- Using Circular Buffers
- Throughput of pipeline
- Non-Linear Pipelines

Using Circular Buffers

Circular buffers can sometimes be used to minimize the overhead of allocating and freeing the items passed between pipeline filters. If the first filter to create an item and last filter to consume an item are both `serial_in_order`, the items can be allocated and freed via a circular buffer of size at least `ntoken`, where `ntoken` is the first parameter to `parallel_pipeline`. Under these conditions, no checking of whether an item is still in use is necessary.

The reason this works is that at most `ntoken` items can be in flight, and items will be freed in the order that they were allocated. Hence by the time the circular buffer wraps around to reallocate an item, the item must have been freed from its previous use in the pipeline. If the first and last filter are *not*`serial_in_order`, then you have to keep track of which buffers are currently in use, because buffers might not be retired in the same order they were allocated.

Throughput of pipeline

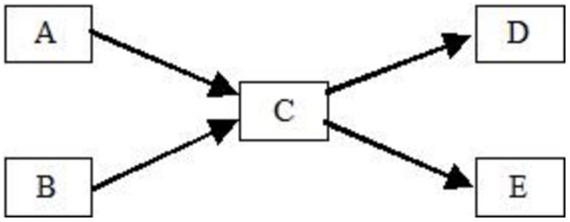
The throughput of a pipeline is the rate at which tokens flow through it, and is limited by two constraints. First, if a pipeline is run with `N` tokens, then obviously there cannot be more than `N` operations running in parallel. Selecting the right value of `N` may involve some experimentation. Too 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 filter. This is true even for a pipeline with no parallel filters. No matter how fast the other filters are, the slowest sequential filter is the bottleneck. So in general you should try to keep the sequential filters fast, and when possible, shift work to the parallel filters.

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

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.

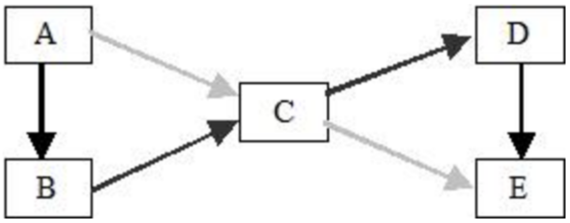
Non-Linear Pipelines

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



However, you can still use a pipeline for this. Just topologically sort the filters 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 filters, 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 filters. This is because filters A and B could process the token concurrently, and likewise filters D and E could process the token concurrently.



In the linear pipeline, the latency is five filters. The behavior of filters 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 filter other than to be passed along to the next filter 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 filter. If `parallel_pipeline` supported non-linear pipelines, it would add a lot of programming complexity, and not improve throughput. The linear limitation of `parallel_pipeline` is a good tradeoff of gain versus pain.

Summary of Loops and Pipelines

The high-level loop and pipeline templates in Intel® oneAPI Threading Building Blocks (oneTBB) 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 these templates to unlock the power of multi-core.

Parallelizing Data Flow and Dependence Graphs

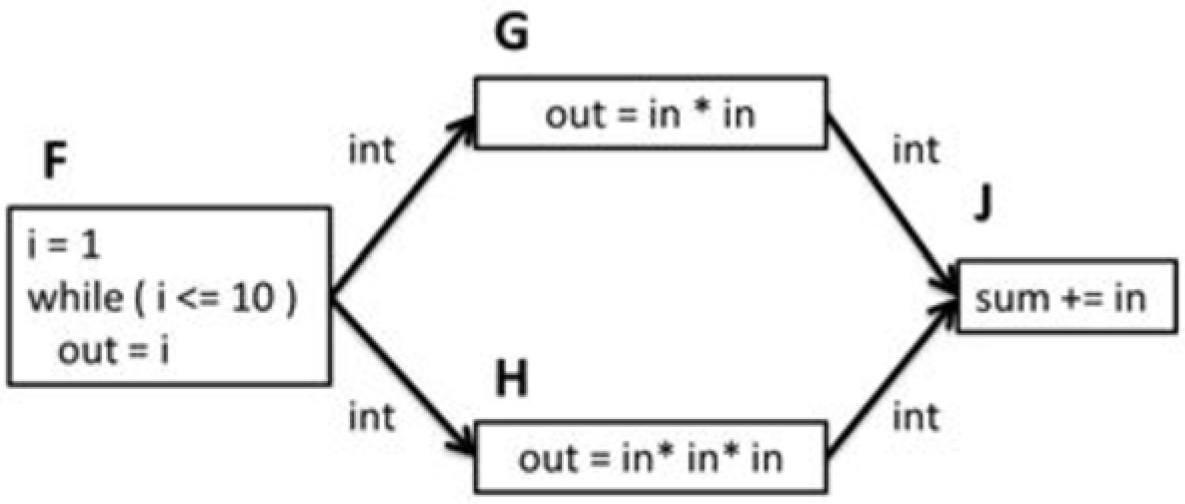
Parallelizing Data Flow and Dependency Graphs

In addition to loop parallelism, the Intel® oneAPI Threading Building Blocks (oneTBB) library also supports graph parallelism. It's possible to create graphs that are highly scalable, but it is also possible to create graphs that are completely sequential.

Using graph parallelism, computations are represented by nodes and the communication channels between these computations are represented by edges. When a node in the graph receives a message, a task is spawned to execute its body object on the incoming message. Messages flow through the graph across the edges that connect the nodes. The following sections present two examples of applications that can be expressed as graphs.

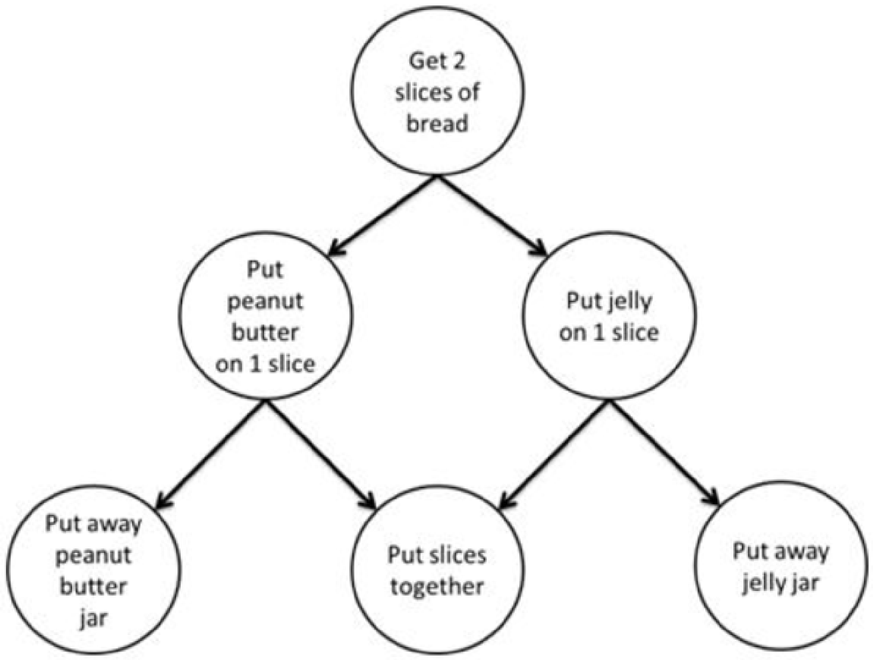
The following figure shows a *streaming* or *data flow* application where a sequence of values is processed as each value passes through the nodes in the graph. In this example, the sequence is created by a function F. For each value in the sequence, G squares the value and H cubes the value. J then takes each of the squared and cubed values and adds them to a global sum. After all values in the sequence are completely processed, sum is equal to the sum of the sequence of squares and cubes from 1 to 10. In a streaming or data flow graph, the values actually flow across the edges; the output of one node becomes the input of its successor(s).

Simple Data Flow Graph



The following graphic shows a different form of graph application. In this example, a dependence graph is used to establish a partial ordering among the steps for making a peanut butter and jelly sandwich. In this partial ordering, you must first get the bread before spreading the peanut butter or jelly on the bread. You must spread on the peanut butter before you put away the peanut butter jar, and likewise spread on the jelly before you put away the jelly jar. And, you need to spread on both the peanut butter and jelly before putting the two slices of bread together. This is a partial ordering because, for example, it doesn't matter if you spread on the peanut butter first or the jelly first. It also doesn't matter if you finish making the sandwich before putting away the jars.

Dependence Graph for Making a Sandwich



While it can be inferred that resources, such as the bread, or the jelly jar, are shared between ordered steps, it is not explicit in the graph. Instead, only the required ordering of steps is explicit in a dependence graph. For example, you must "Put jelly on 1 slice" **before** you "Put away jelly jar".

The flow graph interface in the oneTBB library allows you to express data flow and dependence graphs such as these, as well as more complicated graphs that include cycles, conditionals, buffering and more. If you express your application using the flow graph interface, the runtime library spawns tasks to exploit the parallelism that is present in the graph. For example, in the first example above, perhaps two different values might be squared in parallel, or the same value might be squared and cubed in parallel. Likewise in the second example, the peanut butter might be spread on one slice of bread in parallel with the jelly being spread on the other slice. The interface expresses what is legal to execute in parallel, but allows the runtime library to choose at runtime what will be executed in parallel.

The support for graph parallelism is contained within the namespace `oneapi::tbb::flow` and is defined in the `flow_graph.h` header file.

Basic Flow Graph Concepts

- Flow Graph Basics: Graph Object
- Flow Graph Basics: Nodes
- Flow Graph Basics: Edges
- Flow Graph Basics: Mapping Nodes to Tasks
- Flow Graph Basics: Message Passing Protocol
- Flow Graph Basics: Single-push vs. Broadcast-push
- Flow Graph Basics: Buffering and Forwarding
- Flow Graph Basics: Reservation

Flow Graph Basics: Graph Object

Conceptually a flow graph is a collection of nodes and edges. Each node belongs to exactly one graph and edges are made only between nodes in the same graph. In the flow graph interface, a graph object represents this collection of nodes and edges, and is used for invoking whole graph operations such as waiting for all tasks related to the graph to complete, resetting the state of all nodes in the graph, and canceling the execution of all nodes in the graph.

The code below creates a graph object and then waits for all tasks spawned by the graph to complete. The call to `wait_for_all` in this example returns immediately since this is a trivial graph with no nodes or edges, and therefore no tasks are spawned.

```
graph g;
g.wait_for_all();
```

Flow Graph Basics: Nodes

A node is a class that inherits from oneapi::tbb::flow::graph_node and also typically inherits from oneapi::tbb::flow::sender<T> , oneapi::tbb::flow::receiver<T> or both. A node performs some operation, usually on an incoming message and may generate zero or more output messages. Some nodes require more than one input message or generate more than one output message.

While it is possible to define your own node types by inheriting from graph_node, sender and receiver, it is more typical that predefined node types are used to construct a graph.

A `function_node` is a predefined type available in `flow_graph.h` and represents a simple function with one input and one output. The constructor for a `function_node` takes three arguments:

```
template< typename Body> function_node(graph &g, size_t concurrency, Body body)
```

Parameter	**Description**
Body	Type of the body object.
g	The graph the node belongs to.
concurrency	The concurrency limit for the node. You can use the concurrency limit to control how many invocations of the node are allowed to proceed concurrently, from 1 (serial) to an unlimited number.
body	User defined function object, or lambda expression, that is applied to the incoming message to generate the outgoing message.

Below is code for creating a simple graph that contains a single function_node. In this example, a node n is constructed that belongs to graph g, and has a second argument of 1, which allows at most 1 invocation of the node to occur concurrently. The body is a lambda expression that prints each value v that it receives, spins for v seconds, prints the value again, and then returns v unmodified. The code for the function spin_for is not provided.

```
graph g;
function_node< int, int > n( g, 1, []( int v ) -> int {
    cout << v;
    spin_for( v );
    cout << v;
    return v;
} );
```

After the node is constructed in the example above, you can pass messages to it, either by connecting it to other nodes using edges or by invoking its function try_put. Using edges is described in the next section.

```
n.try_put( 1 );
n.try_put( 2 );
n.try_put( 3 );
```

You can then wait for the messages to be processed by calling wait_for_all on the graph object:

```
g.wait_for_all();
```

In the above example code, the function_node n was created with a concurrency limit of 1. When it receives the message sequence 1, 2 and 3, the node n will spawn a task to apply the body to the first input, 1. When that task is complete, it will then spawn another task to apply the body to 2. And likewise, the node will wait for that task to complete before spawning a third task to apply the body to 3. The calls to try_put do not block until a task is spawned; if a node cannot immediately spawn a task to process the message, the message will be buffered in the node. When it is legal, based on concurrency limits, a task will be spawned to process the next buffered message.

In the above graph, each message is processed sequentially. If however, you construct the node with a different concurrency limit, parallelism can be achieved:

```
function_node< int, int > n( g, oneapi::tbb::flow::unlimited, []( int v ) -> int {
    cout << v;
    spin_for( v );
    cout << v;
    return v;
} );
```

You can use unlimited as the concurrency limit to instruct the library to spawn a task as soon as a message arrives, regardless of how many other tasks have been spawned. You can also use any specific value, such as 4 or 8, to limit concurrency to at most 4 or 8, respectively. It is important to remember that spawning a task does not mean creating a thread. So while a graph may spawn many tasks, only the number of threads available in the library's thread pool will be used to execute these tasks.

Suppose you use unlimited in the function_node constructor instead and call try_put on the node:

```
n.try_put( 1 );
n.try_put( 2 );
n.try_put( 3 );
g.wait_for_all();
```

The library spawns three tasks, each one applying n's lambda expression to one of the messages. If you have a sufficient number of threads available on your system, then all three invocations of the body will occur in parallel. If however, you have only one thread in the system, they execute sequentially.

Flow Graph Basics: Edges

Most applications contain multiple nodes with edges connecting them to each other. In the flow graph interface, edges are directed channels over which messages are passed. They are created by calling the function `make_edge( p, s )` with two arguments: `p`, the predecessor, and `s`, the successor. You can modify the example used in the **Nodes** topic to include a second node that squares the value it receives before printing it and then connect that to the first node with an edge.

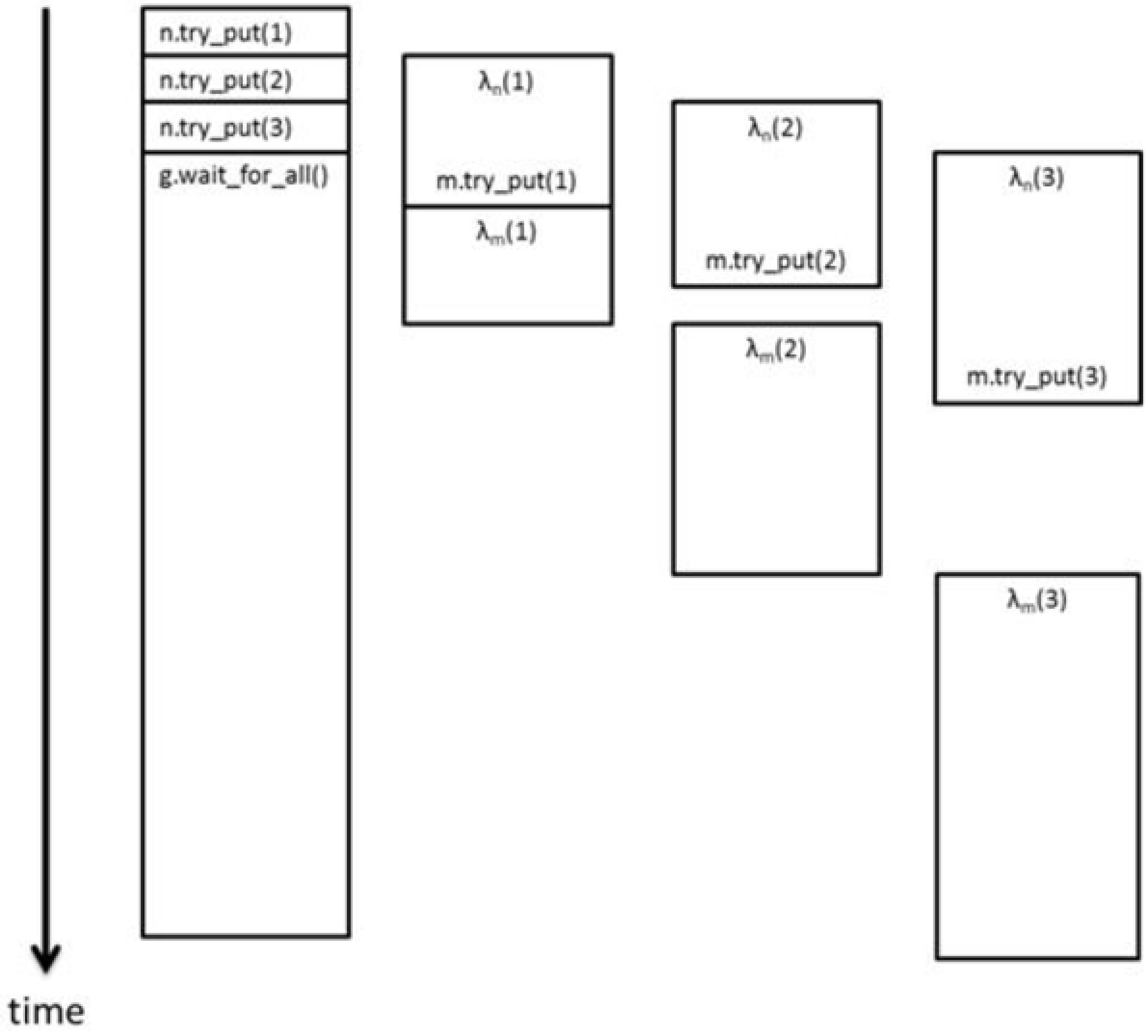
```
graph g;
function_node< int, int > n( g, unlimited, []( int v ) -> int {
    cout << v;
    spin_for( v );
    cout << v;
    return v;
} );
function_node< int, int > m( g, 1, []( int v ) -> int {
    v *= v;
    cout << v;
    spin_for( v );
    cout << v;
    return v;
} );
make_edge( n, m );
n.try_put( 1 );
n.try_put( 2 );
n.try_put( 3 );
g.wait_for_all();
```

Now there are two `function_node``s, ``n` and `m`. The call to `make_edge` creates an edge from `n` to `m`. The node n is created with unlimited concurrency, while `m` has a concurrency limit of 1. The invocations of `n` can all proceed in parallel, while the invocations of `m` will be serialized. Because there is an edge from `n` to `m`, each value `v`, returned by `n`, will be automatically passed to node `m` by the runtime library.

Flow Graph Basics: Mapping Nodes to Tasks

The following figure shows the timeline for one possible execution of the two node graph example in the previous section. The bodies of n and m will be referred to as λ_n and λ_m, respectively. The three calls to try_put spawn three tasks; each one applies the lambda expression, λ_n, on one of the three input messages. Because n has unlimited concurrency, these tasks can execute concurrently if there are enough threads available. The call to `g.wait_for_all()` blocks until there are no tasks executing in the graph. As with other `wait_for_all` functions in oneTBB, the thread that calls `wait_for_all` is not spinning idly during this time, but instead can join in executing other tasks from the work pool.

Execution Timeline of a Two Node Graph



As each task from n finishes, it puts its output to m, since m is a successor of n. Unlike node n, m has been constructed with a concurrency limit of 1 and therefore does not spawn all tasks immediately. Instead, it sequentially spawns tasks to execute its body, λ_m, on the messages in the order that they arrive. When all tasks are complete, the call to `wait_for_all` returns.

NOTE All execution in the flow graph happens asynchronously. The calls to try_put return control to the calling thread quickly, after either immediately spawning a task or buffering the message being passed. Likewise, the body tasks execute the lambda expressions and then put the result to any successor nodes. Only the call to `wait_for_all` blocks, as it should, and even in this case the calling thread may be used to execute tasks from the oneTBB work pool while it is waiting.

The above timeline shows the sequence when there are enough threads to execute all of the tasks that can be executed in parallel. If there are fewer threads, some spawned tasks will need to wait until a thread is available to execute them.

Flow Graph Basics: Message Passing Protocol

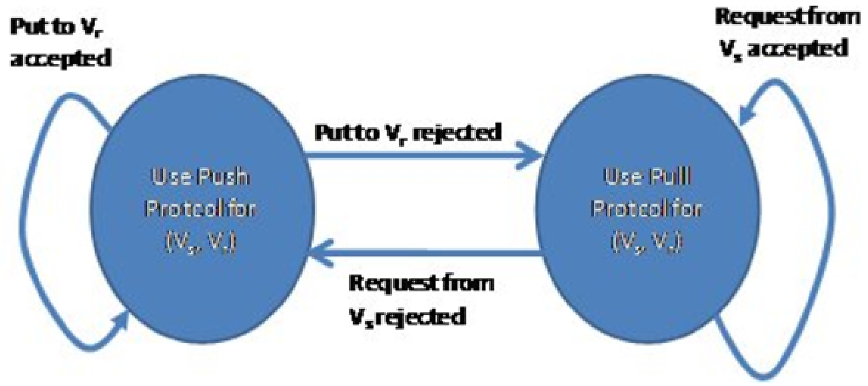
Intel® oneAPI Threading Building Blocks (oneTBB) flow graph operates by passing messages between nodes. A node may not be able to receive and process a message from its predecessor. For a graph to operate most-efficiently, if this occurs the state of the edge between the nodes can change its state to pull so when the successor is able to handle a message it can query its predecessor to see if a message is available. If the edge did not reverse from push to pull, the predecessor node would have to repeatedly attempt to forward its message until the successor accepts it. This would consume resources needlessly.

Once the edge is in pull mode, when the successor is not busy, it will try to pull a message from a predecessor.

1. If a predecessor has a message, the successor will process it and the edge will remain in pull mode.
2. If the predecessor has no message, the edge between the nodes will switch from pull to push mode.

The state diagram of this Push-Pull protocol is:

The dynamic push / pull protocol.


Put to V_r accepted
Request from V_s accepted
Put to V_r rejected
Use Push Protocol for (V_s, V_r)
Use Pull Protocol for (V_s, V_r)
Request from V_s rejected

Flow Graph Basics: Single-push vs. Broadcast-push

Nodes in the Intel® oneAPI Threading Building Blocks (oneTBB) flow graph communicate by pushing and pulling messages. Two policies for pushing messages are used, depending on the type of the node:

- **single-push**: No matter how many successors to the node exist and are able to accept a message, each message will be only sent to one successor.
- **broadcast-push**: A message will be pushed to every successor which is connected to the node by an edge in push mode, and which accepts the message.

The following code demonstrates this difference:

```
using namespace oneapi::tbb::flow;


std::atomic<size_t> g_cnt;

```

```
struct fn_body1 {
    std::atomic<size_t> &body_cnt;
    fn_body1(std::atomic<size_t> &b_cnt) : body_cnt(b_cnt) {}
    continue_msg operator()( continue_msg /*dont_care*/) {
        ++g_cnt;
        ++body_cnt;
        return continue_msg();
    }
};


void run_example1() {  // example for Flow_Graph_Single_Vs_Broadcast.xml
    graph g;
    std::atomic<size_t> b1;  // local counts
    std::atomic<size_t> b2;  // for each function _node body
    std::atomic<size_t> b3;  //
    function_node<continue_msg> f1(g,serial,fn_body1(b1));
    function_node<continue_msg> f2(g,serial,fn_body1(b2));
    function_node<continue_msg> f3(g,serial,fn_body1(b3));
    buffer_node<continue_msg> buf1(g);
    //
    // single-push policy
    //
    g_cnt = b1 = b2 = b3 = 0;
    make_edge(buf1,f1);
    make_edge(buf1,f2);
    make_edge(buf1,f3);
    buf1.try_put(continue_msg());
    buf1.try_put(continue_msg());
    buf1.try_put(continue_msg());
    g.wait_for_all();
    printf( "after single-push test, g_cnt == %d, b1==%d, b2==%d, b3==%d\n", (int)g_cnt,
(int)b1, (int)b2, (int)b3);
    remove_edge(buf1,f1);
    remove_edge(buf1,f2);
    remove_edge(buf1,f3);
    //
    // broadcast-push policy
    //
    broadcast_node<continue_msg> bn(g);
    g_cnt = b1 = b2 = b3 = 0;
    make_edge(bn,f1);
    make_edge(bn,f2);
    make_edge(bn,f3);
    bn.try_put(continue_msg());
    bn.try_put(continue_msg());
    bn.try_put(continue_msg());
    g.wait_for_all();
    printf( "after broadcast-push test, g_cnt == %d, b1==%d, b2==%d, b3==%d\n", (int)g_cnt,
(int)b1, (int)b2, (int)b3);
}
```

The output of this code is

```
after single-push test, g_cnt == 3, b1==3, b2==0, b3==0
after broadcast-push test, g_cnt == 9, b1==3, b2==3, b3==3
```

The single-push test uses a `buffer_node`, which has a "single-push" policy for forwarding messages. Putting three messages to the `buffer_node` results in three messages being pushed. Notice also only the first `function_node` is sent to; in general there is no policy for which node is pushed to if more than one successor can accept.

The broadcast-push test uses a `broadcast_node`, which will push any message it receives to all accepting successors. Putting three messages to the `broadcast_node` results in a total of nine messages pushed to the `function_nodes`.

Only nodes designed to buffer (hold and forward received messages) have a "single-push" policy; all other nodes have a "broadcast-push" policy.

Please see the Sending to One or Multiple Successors section of Flow Graph Tips and Tricks, and Flow Graph Basics: Buffering and Forwarding for more information.

Flow Graph Basics: Buffering and Forwarding

Intel® oneAPI Threading Building Blocks (oneTBB) flow graph nodes use messages to communicate data and to enforce dependencies. If a node passes a message successfully to any successor, no further action is taken with the message by that node. As noted in the section on Single-push vs. Broadcast-push, a message may be passed to one or to multiple successors, depending on the type of the node, how many successors are connected to the node, and whether the message is pushed or pulled.

There are times when a node cannot successfully push a message to any successor. In this case what happens to the message depends on the type of the node. The two possibilities are:

- The node stores the message to be forwarded later.
- The node discards the message.

If a node discards messages that are not forwarded, and this behavior is not desired, the node should be connected to a buffering node that does store messages that cannot be pushed.

If a message has been stored by a node, there are two ways it can be passed to another node:

- A successor to the node can pull the message using `try_get()` or `try_reserve()`.
- A successor can be connected using `make_edge()`.

If a `try_get()` successfully forwards a message, it is removed from the node that stored it. If a node is connected using `make_edge` the node will attempt to push a stored message to the new successor.

Flow Graph Basics: Reservation

Intel® oneAPI Threading Building Blocks (oneTBB) flow graph `join_node` has four possible policies: `queueing`, `reserving`, `key_matching` and `tag_matching`. `join_nodes` need messages at every input before they can create an output message. The reserving `join_node` does not have internal buffering, and it does not pull messages from its inputs until it has a message at each input. To create an output message it temporarily reserves a message at each input port, and only if all input ports succeed reserving messages will an output message be created. If any input port fails to reserve a message, no message will be pulled by the `join_node`.

To support the reserving `join_node` some nodes support **reservation** of their outputs. The way reservation works is:

- When a node connected to a reserving `join_node` in push state tries to push a message, the `join_node` always rejects the push and the edge connecting the nodes is switched to pull mode.
- The reserving input port calls `try_reserve` on each edge in pull state. This may fail; if so, the reserving input port switches that edge to push state, and tries to reserve the next node connected by an edge in pull state. While the input port's predecessor is in reserved state, no other node can retrieve the reserved value.
- If each input port successfully reserves an edge in pull state, the reserving `join_node` will create a message using the reserved messages and try to push the resulting message to any nodes connected to it.

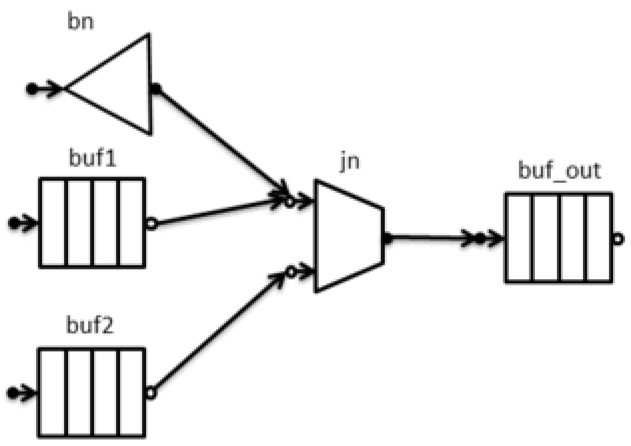
- If the message is successfully pushed to a successor, the predecessors that were reserved are signaled that the messages were used (by calling `try_consume()`.) Those messages will be discarded by the predecessor nodes, because they have been successfully pushed.
- If the message was not successfully pushed to any successor, the predecessors that were reserved are signaled that the messages were not used (by calling `try_release()`.) At this point, the messages may be pushed to or pulled by other nodes.

Because the reserving `join_node` will only attempt to push when each input port has at least one edge in a pull state, and will only attempt to create and push a message if all input ports succeed reserving messages, at least one of the predecessors to each of the reserving `join_node` input ports must be reservable.

The following example demonstrates a reserving `join_node`'s behavior. `buffer_nodes` buffer their output, so they accept a switch of their output edge from push to pull mode. `broadcast_nodes` do not buffer messages and do not support `try_get()` or `try_reserve()`.

```
void run_example2() {  // example for Flow_Graph_Reservation.xml
    graph g;
    broadcast_node<int> bn(g);
    buffer_node<int> buf1(g);
    buffer_node<int> buf2(g);
    typedef join_node<tuple<int,int> reserving> join_type;
    join_type jn(g);
    buffer_node<join_type::output_type> buf_out(g);
    join_type::output_type tuple_out;
    int icnt;


    // join_node predecessors are both reservable buffer_nodes
    make_edge(buf1,input_port<0>jn));
    make_edge(bn,input_port<0>jn));      // attach a broadcast_node
    make_edge(buf2,input_port<1>jn));
    make_edge(jn, buf_out);
    bn.try_put(2);
    buf1.try_put(3);
    buf2.try_put(4);
    buf2.try_put(7);
    g.wait_for_all();
    while (buf_out.try_get(tuple_out)) {
        printf("join_node output == (%d,%d)\n",get<0>tuple_out), get<1>tuple_out) );
    }
    if(buf1.try_get(icnt)) printf("buf1 had %d\n", icnt);
    else printf("buf1 was empty\n");
    if(buf2.try_get(icnt)) printf("buf2 had %d\n", icnt);
    else printf("buf2 was empty\n");
}
```

In the example above, port 0 of the reserving `join_nodejn` has two predecessors: a `buffer_nodebuf1` and a `broadcast_nodebn`. Port 1 of the `join_node` has one predecessor, `buffer_nodebuf2`.

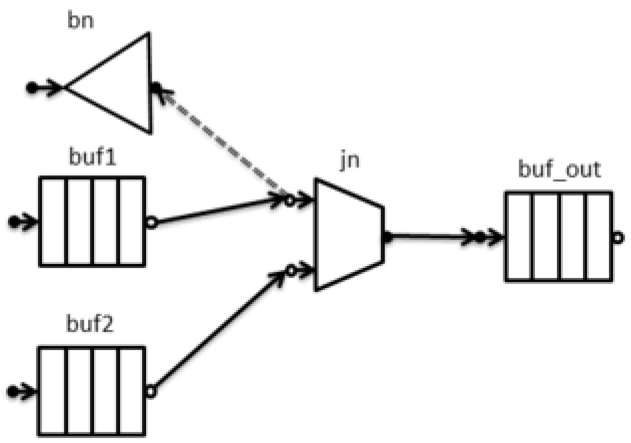


We will discuss one possible execution sequence (the scheduling of tasks may differ slightly, but the end result will be the same.)

```
bn.try_put(2);
```

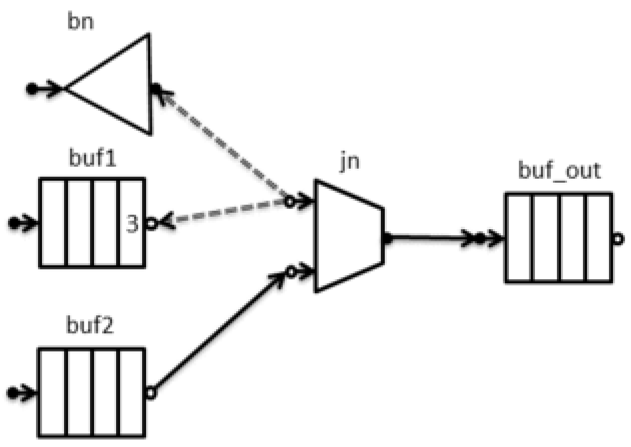
`bn` attempts to forward 2 to `jn`. `jn` does not accept the value and the arc from `bn` to `jn` reverses. Because neither bn nor jn buffer messages, the message is dropped. Because not all the inputs to `jn` have available predecessors, `jn` does nothing further.

Caution Any node which does not support reservation will not work correctly when attached to a reserving `join_node`. This program demonstrates why this occurs; connecting non-reserving nodes to nodes requiring support for reservation is **not** recommended practice.



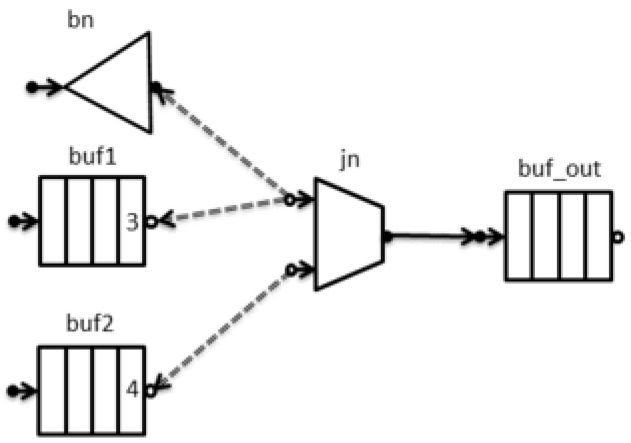
```
buf1.try_put(3);
```

`buf1` attempts to forward 3 to `jn`. `jn` does not accept the value and the arc from `buf1` to `jn` reverses. Because not all the inputs to `jn` have available predecessors, `jn` does nothing further.



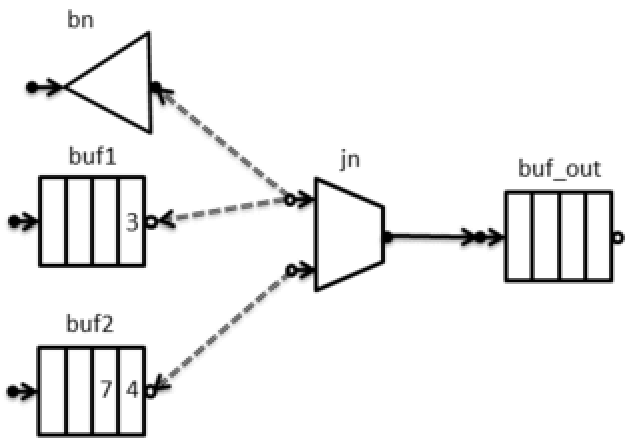
```
buf2.try_put(4);
```

`buf2` attempts to forward 4 to `jn`. `jn` does not accept the value and the arc from `buf2` to `jn` reverses. Now both inputs of `jn` have predecessors, a task to build and forward a message from `jn` will be spawned. We assume that task is not yet executing.



```
buf2.try_put(7);
```

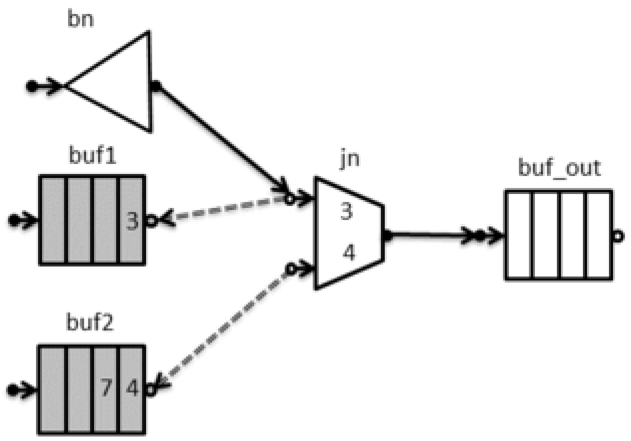
`buf2` has no successor (because the arc to `jn` is reversed,) so it stores the value 7.



Now the task spawned to run `jn` runs.

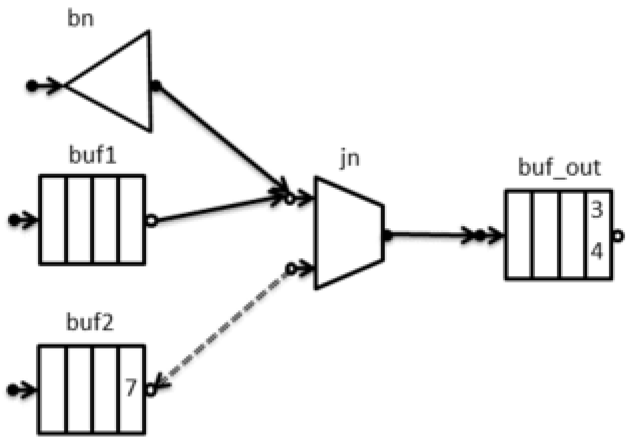
- `jn` tries to reserve `bn`, which fails. The arc to `bn` switches back to the forward direction.

- `jn` tries to reserve `buf1`, which succeeds (reserved nodes are colored grey.) `jn` receives the value 3 from `buf1`, but it remains in `buf1` (in case the attempt to forward a message from `jn` fails.)
- `jn` tries to reserve `buf2`, which succeeds. `jn` receives the value 4 from `buf2`, but it remains in `buf2`.
- `jn` constructs the output message `tuple<3,4>`.



Now `jn` pushes its message to `buf_out`, which accepts it. Because the push succeeded, `jn` signals `buf1` and `buf2` that the reserved values were used, and the buffers discard those values. Now `jn` attempts to reserve again.

- No attempt to pull from `bn` is made, because the edge from `bn` to `jn` is in push state.
- `jn` tries to reserve `buf1`, which fails. The arc to `buf1` switches back to the forward direction.
- `jn` does not try any further actions.



No further activity occurs in the graph, and the `wait_for_all()` will complete. The output of this code is

```
join_node output == (3,4)
buf1 was empty
buf2 had 7
```

Graph Application Categories

Most flow graphs fall into one of two categories:

- **Data flow graphs.** In this type of graph, data is passed along the graph's edges. The nodes receive, transform and then pass along the data messages.
- **Dependence graphs.** In this type of graph, the data operated on by the nodes is obtained through shared memory directly and is not passed along the edges.

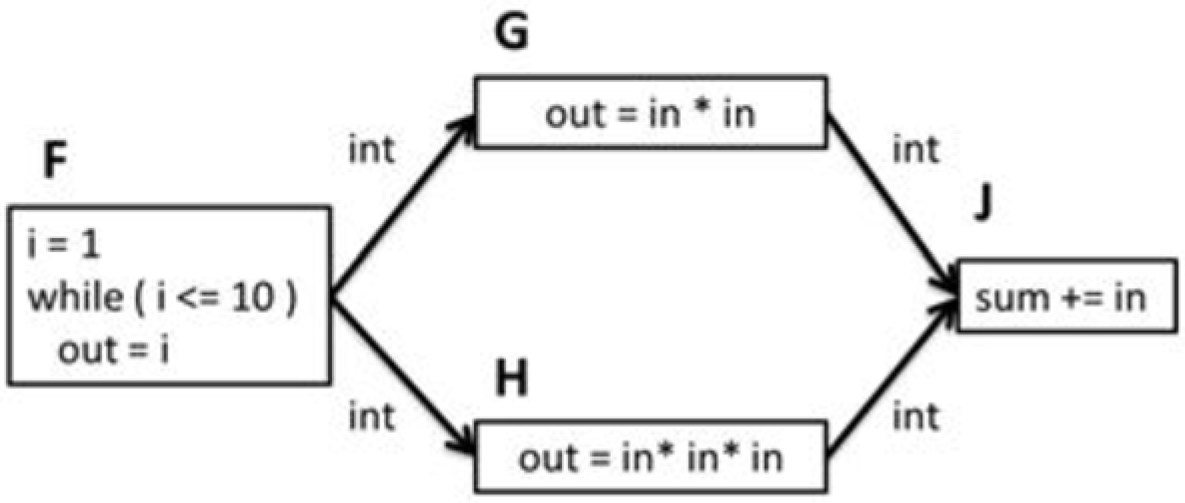
- Data Flow Graph
- Dependence Graph

Data Flow Graph

In a data flow graph, nodes are computations that send and receive data messages. Some nodes may only send messages, others may only receive messages, and others may send messages in response to messages that they receive.

In the following data flow graph, the left-most node generates the integer values from 1 to 10 and passes them to two successor nodes. One of the successors squares each value it receives and passes the result downstream. The second successor cubes each value it receives and passes the result downstream. The right-most node receives values from both of the middle nodes. As it receives each value, it adds it to a running sum of values. When the application is run to completion, the value of sum will be equal to the sum of the sequence of squares and cubes from 1 to 10.

Simple Data Flow Graph



The following code snippet shows an implementation of the **Simple Data Flow Graph** shown above:

```
int sum = 0;
graph g;
function_node< int, int > squarer( g, unlimited, [](const int &v) {
    return v*v;
} );
function_node< int, int > cuber( g, unlimited, [](const int &v) {
    return v*v*v;
} );
function_node< int, int > summer( g, 1, [&](const int &v ) -> int {
    return sum += v;
} );
make_edge( squarer, summer );
make_edge( cuber, summer );


for ( int i = 1; i <= 10; ++i ) {
  squarer.try_put(i);
  cuber.try_put(i);
}
g.wait_for_all();


cout << "Sum is " << sum << "\n";
```

In the implementation above, the following function_nodes are created:

- one to square values
- one to cube values
- one to add values to the global sum

Since the squarer and cuber nodes are side-effect free, they are created with an unlimited concurrency. The summer node updates the sum through a reference to a global variable and therefore is not safe to execute in parallel. It is therefore created with a concurrency limit of 1. The node F from **Simple Data Flow Graph** above is implemented as a loop that puts messages to both the squarer and cuber node.

A slight improvement over the first implementation is to introduce an additional node type, a `broadcast_node`. A `broadcast_node` broadcasts any message it receives to all of its successors.

This enables replacing the two `try_put`'s in the loop with a single `try_put`:

```
broadcast_node<int> b(g);
make_edge( b, squarer );
make_edge( b, cuber );
for ( int i = 1; i <= 10; ++i ) {
  b.try_put(i);
}
g.wait_for_all();
```

An even better option, which will make the implementation even more like the **Simple Data Flow Graph** above, is to introduce an `input_node`. An `input_node`, as the name implies only sends messages and does not receive messages. Its constructor takes two arguments:

```
template< typename Body > input_node( graph &g, Body body)
```

The body is a function object, or lambda expression, that contains a function operator:

```
Output Body::operator()( oneapi::tbb::flow_control &fc );
```

You can replace the loop in the example with an `input_node`

```
input_node< int > src( g, src_body(10) );
make_edge( src, squarer );
make_edge( src, cuber );
src.activate();
g.wait_for_all();
```

The runtime library will repeatedly invoke the function operator in `src_body` until `fc.stop()` is invoked inside the body. You therefore need to create body that will act like the body of the loop in the **Simple Data Flow Graph** above. The final implementation after all of these changes is shown below:

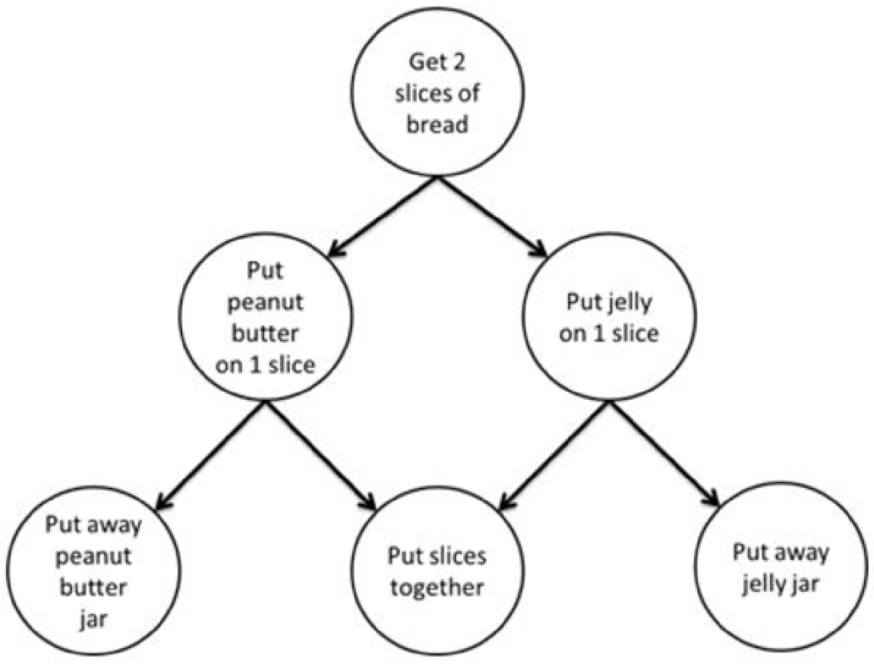
```
class src_body {
    const int my_limit;
    int my_next_value;
public:
    src_body(int l) : my_limit(l), my_next_value(1) {}
    int operator()( oneapi::tbb::flow_control& fc ) {
        if ( my_next_value <= my_limit ) {
            return my_next_value++;
        } else {
            fc.stop();
            return int();
        }
    }
};


int main() {
  int sum = 0;
  graph g;
  function_node< int, int > squarer( g, unlimited, [](const int &v) {
      return v*v;
  } );
  function_node< int, int > cuber( g, unlimited, [](const int &v) {
      return v*v*v;
  } );
  function_node< int, int > summer( g, 1, [&](const int &v ) -> int {
      return sum += v;
  } );
  make_edge( squarer, summer );
  make_edge( cuber, summer );
  input_node< int > src( g, src_body(10) );
  make_edge( src, squarer );
  make_edge( src, cuber );
  src.activate();
  g.wait_for_all();
  cout << "Sum is " << sum << "\n";
}
```

This final implementation has all of the nodes and edges from the **Simple Data Flow Graph** above. In this simple example, there is not much advantage in using an `input_node` over an explicit loop. But, because an `input_node` is able to react to the behavior of downstream nodes, it can limit memory use in more complex graphs. For more information, see:ref:**create_token_based_system** .

Dependence Graph

In a dependence graph, the nodes invoke body objects to perform computations and the edges create a partial ordering of these computations. At runtime, the library spawns and schedules tasks to execute the body objects when it is legal to do so according to the specified partial ordering. The following figure shows an example of an application that could be expressed using a dependence graph.

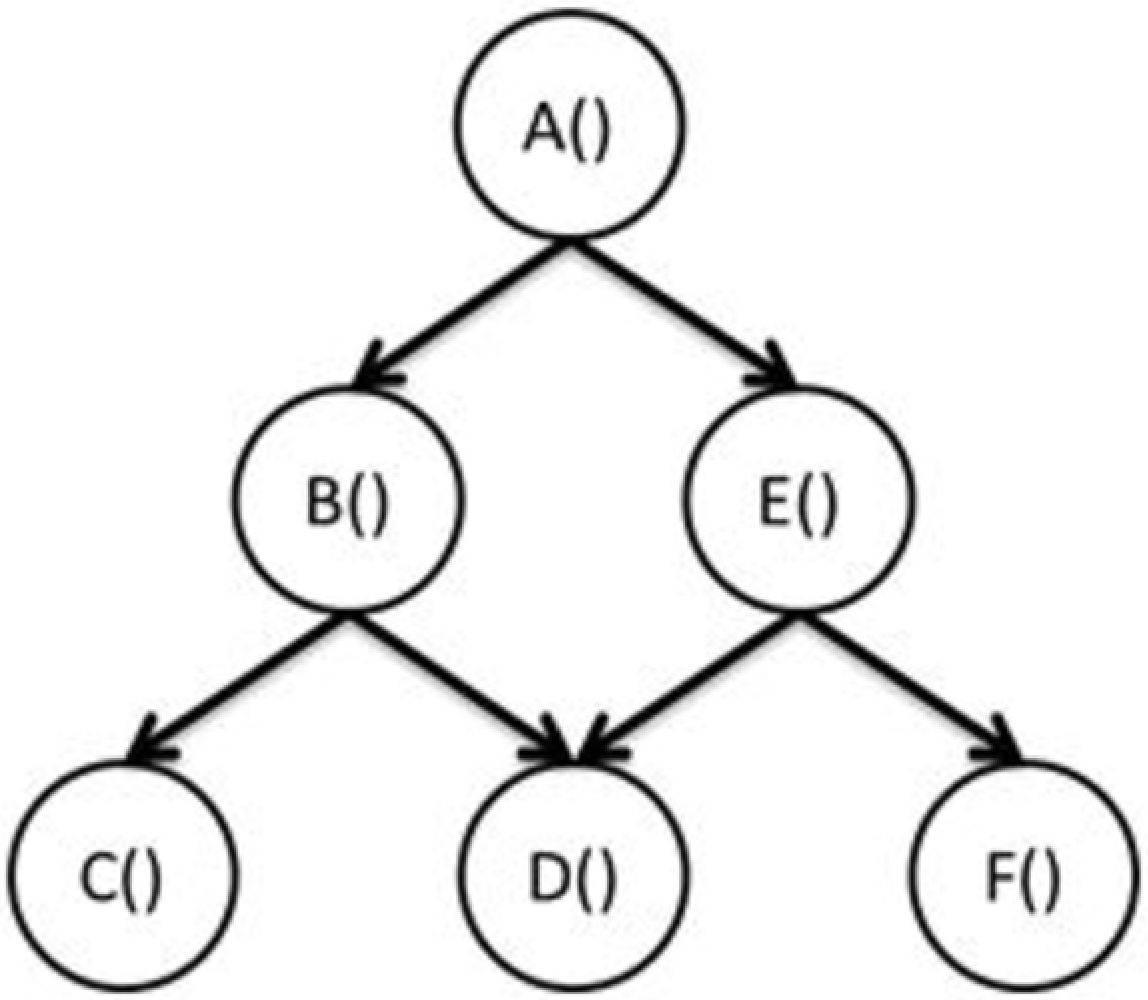
Dependence Graph for Making a Sandwich



Dependence graphs are a special case of data flow graphs, where the data passed between nodes are of type oneapi::tbb::flow::continue_msg. Unlike a general data flow graph, nodes in a dependence graph do not spawn a task for each message they receive. Instead, they are aware of the number of predecessors they have, count the messages they receive and only spawn a task to execute their body when this count is equal to the total number of their predecessors.

The following figure shows another example of a dependence graph. It has the same topology as the figure above, but with simple functions replacing the sandwich making steps. In this partial ordering, function A must complete executing before any other computation starts executing. Function B must complete before C and D start executing; and E must complete before D and F start executing. This is a partial ordering because, for example, there is no explicit ordering requirement between B and E or C and F.

Simple Dependence Graph



To implement this as a flow graph, continue_node objects are used for the nodes and continue_msg objects as the messages. A continue_node constructor takes two arguments:

```
template< typename Body > continue_node( graph &g, Body body)
```

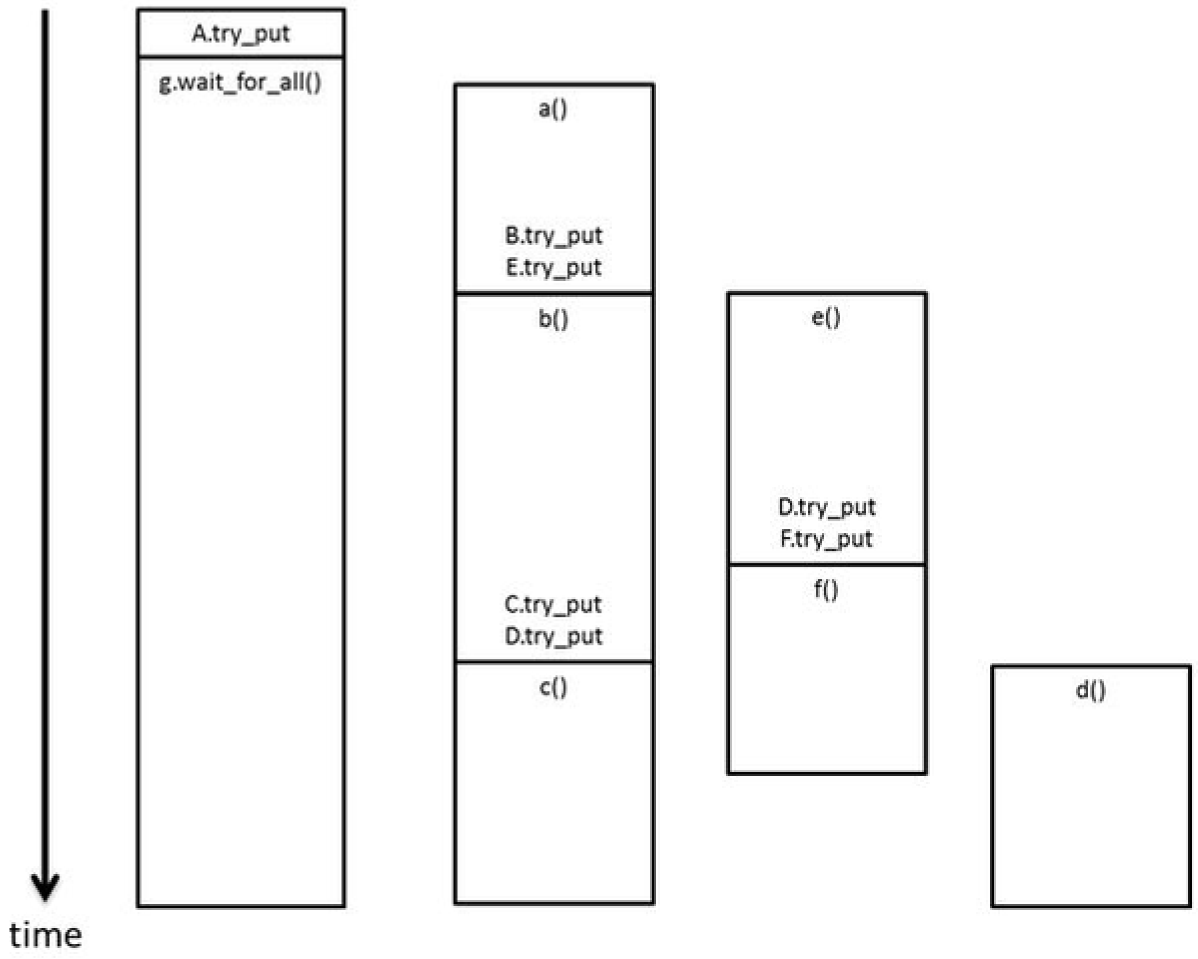
The first argument is the graph it belongs to and the second is a function object or lambda expression. Unlike a function_node, a continue_node is always assumed to have unlimited concurrency and will immediately spawn a task whenever its dependencies are met.

The following code snippet is an implementation of the example in this figure.

```
typedef continue_node< continue_msg > node_t;
typedef const continue_msg & msg_t;


int main() {
  oneapi::tbb::flow::graph g;
  node_t A(g, [](msg_t){ a(); } );
  node_t B(g, [](msg_t){ b(); } );
  node_t C(g, [](msg_t){ c(); } );
  node_t D(g, [](msg_t){ d(); } );
  node_t E(g, [](msg_t){ e(); } );
  node_t F(g, [](msg_t){ f(); } );
  make_edge(A, B);
  make_edge(B, C);
  make_edge(B, D);
  make_edge(A, E);
  make_edge(E, D);
  make_edge(E, F);
  A.try_put( continue_msg() );
  g.wait_for_all();
  return 0;
}
```

One possible execution of this graph is shown below. The execution of D does not start until both B and E are finished. While a task is waiting in the wait_for_all, its thread can participate in executing other tasks from the oneTBB work pool.

Execution Timeline for a Dependence Graph



Again, it is important to note that all execution in the flow graph happens asynchronously. The call to A.try_put returns control to the calling thread quickly, after incrementing the counter and spawning a task to execute the body of A. Likewise, the body tasks execute the lambda expressions and then put a continue_msg to all successor nodes, if any. Only the call to wait_for_all blocks, as it should, and even in this case the calling thread may be used to execute tasks from the oneTBB work pool while it is waiting.

The above timeline shows the sequence when there are enough threads to execute all of the tasks that can be executed concurrently in parallel. If there are fewer threads, then some tasks that are spawned will need to wait until a thread is available to execute them.

Predefined Node Types

You can define your own node types by inheriting from class graph_node, class sender and class receiver but it is likely that you can create your graph with the predefined node types already available in flow_graph.h. Below is a table that lists all of the predefined types with a basic description. See the Developer Reference for a more detailed description of each node.

Predefined Node Type	Description
input_node	A single-output node, with a generic output type. When activated, it executes a user body to generate its output. Its body is invoked if downstream nodes have accepted the previous

Predefined Node Type	Description
	generated output. Otherwise, the previous output is temporarily buffered until it is accepted downstream and then the body is again invoked.
function_node	A single-input single-output node that broadcasts its output to all successors. Has generic input and output types. Executes a user body, and has controllable concurrency level and buffering policy. For each input exactly one output is returned.
continue_node	A single-input, single-output node that broadcasts its output to all successors. It has a single input that requires 1 or more inputs of type continue_msg and has a generic output type. It executes a user body when it receives N continue_msg objects at its input. N is equal to the number of predecessors plus any additional offset assigned at construction time.
multifunction_node	A single-input multi-output node. It has a generic input type and several generic output types. It executes a user body, and has controllable concurrency level and buffering policy. The body can output zero or more messages on each output port.
broadcast_node	A single-input, single-output node that broadcasts each message received to all successors. Its input and output are of the same generic type. It does not buffer messages.
buffer_node, queue_node, priority_queue_node, and sequencer_node.	Single-input, single-output nodes that buffer messages and send their output to one successor. The order in which the messages are sent are node specific (see the Developer Reference). These nodes are unique in that they send to only a single successor and not all successors.
join_node	A multi-input, single-output node. There are several generic input types and the output type is a tuple of these generic types. The node combines one message from each input port to create a tuple that is broadcast to all successors. The policy used to combine messages is selectable as queueing, reserving or tag-matching.
split_node	A single-input, multi-output node. The input type is a tuple of generic types and there is one output port for each of the types in the tuple. The node receives a tuple of values and outputs each element of the tuple on a corresponding output port.
write_once_node, overwrite_node	Single-input, single-output nodes that buffer a single message and broadcast their outputs to all successors. After broadcast, the nodes retain the last message received, so it is available to any future successor. A write_once_node will only accept the first message it receives, while the

Predefined Node Type	Description
	overwrite_node will accept all messages, broadcasting them to all successors, and replacing the old value with the new.
limiter_node	A multi-input, single output node that broadcasts its output to all successors. The main input type and output type are of the same generic type. The node increments an internal counter when it broadcasts a message. If the increment causes it to reach its user-assigned threshold, it will broadcast no more messages. A special input port can be used to adjust the internal count, allowing further messages to be broadcast. The node does not buffer messages.
indexer_node	A multi-input, single-output node that broadcasts its output message to all of its successors. The input type is a list of generic types and the output type is a tagged_msg. The message is of one of the types listed in the input and the tag identifies the port on which the message was received. Messages are broadcast individually as they arrive at the input ports.
composite_node	A node that might have 0, 1 or multiple ports for both input and output. The composite_node packages a group of other nodes together and maintains a tuple of references to ports that border it. This allows for the corresponding ports of the composite_node to be used to make edges which hitherto would have been made from the actual nodes in the composite_node.
async_node (preview feature)	A node that allows a flow graph to communicate with an external activity managed by the user or another runtime. This node receives messages of generic type, invokes the user-provided body to submit a message to an external activity. The external activity can use a special interface to return a generic type and put it to all successors of async_node.

Flow Graph Tips and Tricks

- Flow Graph Tips for Waiting for and Destroying a Flow Graph
  - Always Use wait_for_all()
  - Avoid Dynamic Node Removal
  - Destroying Graphs That Run Outside the Main Thread
- Flow Graph Tips on Making Edges
  - Use make_edge and remove_edge
  - Sending to One or Multiple Successors
  - Communication Between Graphs
  - Using input_node
  - Avoiding Data Races
- Flow Graph Tips on Nested Parallelism
  - Use Nested Algorithms to Increase Scalability

Flow Graph Tips for Waiting for and Destroying a Flow Graph

Always Use wait_for_all()

One of the most common mistakes made in flow graph programming is to forget to call wait_for_all. The function graph::wait_for_all blocks until all tasks spawned by the graph are complete. This is not only useful when you want to wait until the computation is done, but it is necessary to call wait_for_all before destroying the graph, or any of its nodes. For example, the following function will lead to a program failure:

```
void no_wait_for_all() {
    graph g;
    function_node< int, int > f( g, 1, []( int i ) -> int {
        return spin_for(i);
    } );
    f.try_put(1);


    // program will fail when f and g are destroyed at the
    // end of the scope, since the body of f is not complete
}
```

In the function above, the graph g and its node f are destroyed at the end of the function’s scope. However, the task spawned to execute f’s body is still in flight. When the task completes, it will look for any successors connected to its node, but by then both the graph and the node have been deleted out from underneath it. Placing a g.wait_for_all() at the end of the function prevents the premature destruction of the graph and node.

If you use a flow graph and see mysterious behavior, check first to see that you have called wait_for_all.

Avoid Dynamic Node Removal

These are the basic guidelines regarding nodes and edges:

- Avoid dynamic node removal
- Adding edges and nodes is supported
- Removing edges is supported

It is possible to add new nodes and edges and to remove old edges from a flow graph as nodes are actively processing messages in the graph. However, removing nodes is discouraged. Destroying a graph or any of its nodes while there are messages being processed in the graph can lead to premature deletion of memory that will be later touched by tasks in the graph causing program failure. Removal of nodes when the graph is not idle may lead to intermittent failures and hard to find failures, so it should be avoided.

Destroying Graphs That Run Outside the Main Thread

Make sure to enqueue a task to wait for and destroy graphs that run outside the main thread.

You may not always want to block the main application thread by calling wait_for_all(). However, it is safest to call wait_for_all on a graph before destroying it. A common solution is to enqueue a task to build and wait for the graph to complete. For example, assume you really do not want to call a wait_for_all in the example from Always Use wait_for_all(), Instead you can enqueue a task that creates the graph and waits for it:

```
class background_task {
public:
  void operator()() {
    graph g;
    function_node< int, int > f( g, 1, []( int i ) -> int {
      return spin_for(i);
    } );
    f.try_put(1);
    g.wait_for_all();
  }
};


void no_wait_for_all_enqueue() {
  task_arena a;
  a.enqueue(background_task());
  // do other things without waiting…
}
```

In the code snippet above, the enqueued task executes at some point, but it's not clear when. If you need to use the results of the enqueued task, or even ensure that it completes before the program ends, you will need to use some mechanism to signal from the enqueued task that the graph is complete.

Flow Graph Tips on Making Edges

- Use make_edge and remove_edge
- Sending to One or Multiple Successors
- Communication Between Graphs
- Using input_node
- Avoiding Data Races

Use make_edge and remove_edge

These are the basic guidelines for creating and removing edges:

- use make_edge and remove_edge
- Avoid using register_successor and register_predecessor
- Avoid using remove_successor and remove_predecessor

As a convention, to communicate the topology, use only functions flow::make_edge and flow::remove_edge. The runtime library uses node functions, such as sender<T>::register_successor, to create these edges, but those functions should not be called directly. The runtime library calls these node functions directly to implement optimizations on the topology at runtime.

Sending to One or Multiple Successors

An important characteristic of the predefined nodes is whether they push their output to a single successor or broadcast to all successors. The following predefined nodes push messages to a single successor:

- buffer_node
- queue_node
- priority_queue_node
- sequencer_node

Other nodes push messages to all successors that will accept them.

The nodes that push to only a single successor are all buffer nodes. Their purpose is to hold messages temporarily, until they are consumed downstream. Consider the example below:

```
void use_buffer_and_two_nodes() {
  graph g;


  function_node< int, int, rejecting > f1( g, 1, []( int i ) -> int {
    spin_for(0.1);
    cout << "f1 consuming " << i << "\n";
    return i;
  } );


  function_node< int, int, rejecting > f2( g, 1, []( int i ) -> int {
    spin_for(0.2);
    cout << "f2 consuming " << i << "\n";
    return i;
  } );


  priority_queue_node< int > q(g);


  make_edge( q, f1 );
  make_edge( q, f2 );
  for ( int i = 10; i > 0; --i ) {
    q.try_put( i );
  }
  g.wait_for_all();
}
```

First, function_nodes by default queue up the messages they receive at their input. To make a priority_queue_node work properly with a function_node, the example above constructs its function_nodes with its buffer policy set to rejecting. So, f1 and f2 do not internally buffer incoming messages, but instead rely on upstream buffering in the priority_queue_node.

In the above example, each message buffered by the priority_queue_node is sent to either f1 or f2, but not both.

Let’s consider the alternative behavior; that is; what if the priority_queue_node broadcasts to all successors. What if some, but not all, nodes accept a message? Should the message be buffered until all nodes accept it, or be only delivered to the accepting subset? If the node continues to buffer the message, should it eventually deliver the messages in the same order to all nodes or in the current priority order at the time the node accepts the next message? For example, assume a priority_queue_node only contains “9” when a successor node, f1, accepts “9” but another successor node, f2, rejects it. Later a value “100” arrives and f2 is available to accept messages. Should f2 receive “9” next or “100”, which has a higher priority? In any case, trying to ensure that all successors receive each message creates a garbage collection problem and complicates reasoning. Therefore, these buffering nodes push each message to only one successor. And, you can use this characteristic to create useful graph structures such as the one shown in the graph above, where each message will be processed in priority order, by either f1 or f2.

But what if you really do want both f1 and f2 to receive all of the values, and in priority order? You can easily create this behavior by creating one priority_queue_node for each function_node, and pushing each value to both queues through a broadcast_node, as shown below:

```
graph g;


function_node< int, int, rejecting > f1( g, 1, []( int i ) -> int {
  spin_for(0.1);
  cout << "f1 consuming " << i << "\n";
  return i;
} );


function_node< int, int, rejecting > f2( g, 1, []( int i ) -> int {
  spin_for(0.2);
  cout << "f2 consuming " << i << "\n";
  return i;
} );


priority_queue_node< int > q1(g);
priority_queue_node< int > q2(g);
broadcast_node< int > b(g);


make_edge( b, q1 );
make_edge( b, q2 );
make_edge( q1, f1 );
make_edge( q2, f2 );
for ( int i = 10; i > 0; --i ) {
  b.try_put( i );
}
g.wait_for_all();
```

So, when connecting a node in your graph to multiple successors, be sure to understand whether the output will broadcast to all of the successors, or just a single successor.

Communication Between Graphs

All graph nodes require a reference to a graph object as one of the arguments to their constructor. It is only safe to construct edges between nodes that are part of the same graph. An edge expresses the topology of your graph to the runtime library. Connecting two nodes in different graphs can make it difficult to reason about whole graph operations, such as calls to graph::wait_for_all and exception handling. To optimize performance, the library may make calls to a node’s predecessor or successor at times that are unexpected by the user.

If two graphs must communicate, do NOT create an edge between them, but instead use explicit calls to try_put. This will prevent the runtime library from making any assumptions about the relationship of the two nodes, and therefore make it easier to reason about events that cross the graph boundaries. However, it may still be difficult to reason about whole graph operations. For example, consider the graphs below:

```
graph g;
function_node< int, int > n1( g, 1, [](int i) -> int {
    cout << "n1\n";
    spin_for(i);
    return i;
} );
function_node< int, int > n2( g, 1, [](int i) -> int {
    cout << "n2\n";
    spin_for(i);
    return i;
} );
make_edge( n1, n2 );


graph g2;
function_node< int, int > m1( g2, 1, [](int i) -> int {
    cout << "m1\n";
    spin_for(i);
    return i;
} );
function_node< int, int > m2( g2, 1, [&](int i) -> int {
    cout << "m2\n";
    spin_for(i);
    n1.try_put(i);
    return i;
} );
make_edge( m1, m2 );


m1.try_put( 1 );


// The following call returns immediately:
g.wait_for_all();
// The following call returns after m1 & m2
g2.wait_for_all();


// we reach here before n1 & n2 are finished
// even though wait_for_all was called on both graphs
```

In the example above, m1.try_put(1) sends a message to node m1, which runs its body and then sends a message to node m2. Next, node m2 runs its body and sends a message to n1 using an explicit try_put. In turn, n1 runs its body and sends a message to n2. The runtime library does not consider m2 to be a predecessor of n1 since no edge exists.

If you want to wait until all of the tasks spawned by these graphs are done, you need to call the function wait_for_all on both graphs. However, because there is cross-graph communication, the order of the calls is important. In the (incorrect) code segment above, the first call to g.wait_for_all() returns immediately because there are no tasks yet active in g; the only tasks that have been spawned by then belong to g2. The call to g2.wait_for_all returns after both m1 and m2 are done, since they belong to g2; the call does not however wait for n1 and n2, since they belong to g. The end of this code segment is therefore reached before n1 and n2 are done.

If the calls to wait_for_all are swapped, the code works as expected:

```
g2.wait_for_all();
g.wait_for_all();


// all tasks are done
```

While it is not too difficult to reason about how these two very small graphs interact, the interaction of two larger graphs, perhaps with cycles, will be more difficult to understand. Therefore, communication between nodes in different graphs should be done with caution.

Using input_node

By default, an `input_node` is constructed in the inactive state:

```
template< typename Body > input_node( graph &g, Body body, bool is_active=true )
```

To activate an inactive `input_node`, you call the node’s function activate:

```
input_node< int > src( g, src_body(10), false );
// use it in calls to make_edge…
src.activate();
```

All `input_node` objects are constructed in the inactive state and usually activated after the entire flow graph is constructed.

For example, you can use the code in Data Flow Graph. In that implementation, the `input_node` is constructed in the inactive state and activated after all other edges are made:

```
make_edge( squarer, summer );
make_edge( cuber, summer );
input_node< int > src( g, src_body(10), false );
make_edge( src, squarer );
make_edge( src, cuber );
src.activate();
g.wait_for_all();
```

In this example, if the `input_node` was toggled to the active state at the beginning, it might send a message to squarer immediately after the edge to squarer is connected. Later, when the edge to cuber is connected, cuber will receive all future messages, but may have already missed some.

In general it is safest to create your `input_node` objects in the inactive state and then activate them after the whole graph is constructed. However, this approach serializes graph construction and graph execution.

Some graphs can be constructed safely with `input_node``s active, allowing the overlap of construction and execution. If your graph is a directed acyclic graph (DAG), and each ``input_node` has only one successor, you can activate your `input_node``s just after their construction if you construct the edges in reverse topological order; that is, make the edges at the largest depth in the tree first, and work back to the shallowest edges. For example, if src is an ``input_node` and `func1` and `func2` are both function nodes, the following graph would not drop messages, even though src is activated just after its construction:

```
const int limit = 10;
int count = 0;
graph g;
oneapi::tbb::flow::graph g;
oneapi::tbb::flow::input_node<int> src( g, [&]( oneapi::tbb::flow_control &fc ) -> int {
  if ( count < limit ) {
    return ++count;
  }
  fc.stop();
  return {};
});
src.activate();

oneapi::tbb::flow::function_node<int,int> func1( g, 1, []( int i ) -> int {
  std::cout << i << "\n";
  return i;
} );
oneapi::tbb::flow::function_node<int,int> func2( g, 1, []( int i ) -> int {
  std::cout << i << "\n";
  return i;
} );
```

```
make_edge( func1, func2 );
make_edge( src, func1 );


g.wait_for_all();
```

The above code is safe because the edge from `func1` to `func2` is made before the edge from src to `func1`. If the edge from src to func1 were made first, `func1` might generate a message before `func2` is attached to it; that message would be dropped. Also, src has only a single successor. If src had more than one successor, the successor that is attached first might receive messages that do not reach the successors that are attached after it.

Avoiding Data Races

The edges in a flow graph make explicit the dependence relationships that you want the library to enforce. Similarly, the concurrency limits on `function_node` and `multifunction_node` objects limit the maximum number of concurrent invocations that the runtime library will allow. These are the limits that are enforced by the library; the library does not automatically protect you from data races. You must explicitly prevent data races by using these mechanisms.

For example, the follow code has a data race because there is nothing to prevent concurrent accesses to the global count object referenced by node f:

```
graph g;
int src_count = 1;
int global_sum = 0;
int limit = 100000;

input_node< int > src( g, [&]( oneapi::tbb::flow_control& fc ) -> int {
  if ( src_count <= limit ) {
    return src_count++;
  } else {
    fc.stop();
    return int();
  }
} );
src.activate();

function_node< int, int > f( g, unlimited, [&]( int i ) -> int {
  global_sum += i;  // data race on global_sum
  return i;
} );


make_edge( src, f );
g.wait_for_all();


cout << "global sum = " << global_sum
     << " and closed form = " << limit*(limit+1)/2 << "\n";
```

If you run the above example, it will likely calculate a global sum that is a bit smaller than the expected solution due to the data race. The data race could be avoided in this simple example by changing the allowed concurrency in `f` from unlimited to 1, forcing each value to be processed sequentially by `f`. You may also note that the `input_node` also updates a global value, `src_count`. However, since an `input_node` always executes serially, there is no race possible.

Flow Graph Tips on Nested Parallelism

- Use Nested Algorithms to Increase Scalability
- Use Nested Flow Graphs

Use Nested Algorithms to Increase Scalability

One powerful way to increase the scalability of a flow graph is to nest other parallel algorithms inside of node bodies. Doing so, you can use a flow graph as a coordination language, expressing the most coarse-grained parallelism at the level of the graph, with finer grained parallelism nested within.

In the example below, five nodes are created: an `input_node`, `matrix_source`, that reads a sequence of matrices from a file, two `function_nodes`, `n1` and `n2`, that receive these matrices and generate two new matrices by applying a function to each element, and two final `function_nodes`, `n1_sink` and `n2_sink`, that process these resulting matrices. The `matrix_source` is connected to both `n1` and `n2`. The node `n1` is connected to `n1_sink`, and `n2` is connected to `n2_sink`. In the lambda expressions for `n1` and `n2`, a `parallel_for` is used to apply the functions to the elements of the matrix in parallel. The functions `read_next_matrix`, `f1`, `f2`, `consume_f1` and `consume_f2` are not provided below.

```
graph g;
input_node< double * > matrix_source( g, [&]( oneapi::tbb::flow_control &fc ) -> double* {
  double *a = read_next_matrix();
  if ( a ) {
    return a;
  } else {
    fc.stop();
    return nullptr;
  }
} );
function_node< double *, double * > n1( g, unlimited, [&]( double *a ) -> double * {
  double *b = new double[N];
  parallel_for( 0, N, [&](int i) {
    b[i] = f1(a[i]);
  } );
  return b;
} );
function_node< double *, double * > n2( g, unlimited, [&]( double *a ) -> double * {
  double *b = new double[N];
  parallel_for( 0, N, [&](int i) {
    b[i] = f2(a[i]);
  } );
  return b;
} );
function_node< double *, double * > n1_sink( g, unlimited,
  []( double *b ) -> double * {
    return consume_f1(b);
} );
function_node< double *, double * > n2_sink( g, unlimited,
  []( double *b ) -> double * {
    return consume_f2(b);
} );
make_edge( matrix_source, n1 );
make_edge( matrix_source, n2 );
make_edge( n1, n1_sink );
make_edge( n2, n2_sink );
matrix_source.activate();
g.wait_for_all();
```

Use Nested Flow Graphs

In addition to nesting algorithms within a flow graph node, it is also possible to nest flow graphs. For example, below there is a graph `g` with two nodes, `a` and `b`. When node `a` receives a message, it constructs and executes an inner dependence graph. When node `b` receives a message, it constructs and executes an inner data flow graph:

```
graph g;
  function_node< int, int > a( g, unlimited, []( int i ) -> int {
      graph h;
      node_t n1( h, [=]( msg_t ) { cout << "n1: " << i << "\n"; } );
      node_t n2( h, [=]( msg_t ) { cout << "n2: " << i << "\n"; } );
      node_t n3( h, [=]( msg_t ) { cout << "n3: " << i << "\n"; } );
      node_t n4( h, [=]( msg_t ) { cout << "n4: " << i << "\n"; } );
      make_edge( n1, n2 );
      make_edge( n1, n3 );
      make_edge( n2, n4 );
      make_edge( n3, n4 );
      n1.try_put(continue_msg());
      h.wait_for_all();
      return i;
  } );
  function_node< int, int > b( g, unlimited, []( int i ) -> int {
      graph h;
      function_node< int, int > m1( h, unlimited, []( int j ) -> int {
          cout << "m1: " << j << "\n";
          return j;
      } );
      function_node< int, int > m2( h, unlimited, []( int j ) -> int {
          cout << "m2: " << j << "\n";
          return j;
      } );
      function_node< int, int > m3( h, unlimited, []( int j ) -> int {
          cout << "m3: " << j << "\n";
          return j;
      } );
      function_node< int, int > m4( h, unlimited, []( int j ) -> int {
          cout << "m4: " << j << "\n";
          return j;
      } );
      make_edge( m1, m2 );
      make_edge( m1, m3 );
      make_edge( m2, m4 );
      make_edge( m3, m4 );
      m1.try_put(i);
      h.wait_for_all();
      return i;
  } );
  make_edge( a, b );
  for ( int i = 0; i < 3; ++i ) {
      a.try_put(i);
  }
  g.wait_for_all();
```

If the nested graph remains unchanged in structure between invocations of the node, it is redundant to construct it each time. Reconstructing the graph only adds overhead to the execution. You can modify the example above, for example, to have node `b` reuse a graph that is persistent across its invocations:

```
graph h;
  function_node< int, int > m1( h, unlimited, []( int j ) -> int {
      cout << "m1: " << j << "\n";
      return j;
  } );
  function_node< int, int > m2( h, unlimited, []( int j ) -> int {
      cout << "m2: " << j << "\n";
      return j;
  } );
  function_node< int, int > m3( h, unlimited, []( int j ) -> int {
      cout << "m3: " << j << "\n";
      return j;
  } );
  function_node< int, int > m4( h, unlimited, []( int j ) -> int {
      cout << "m4: " << j << "\n";
      return j;
  } );
  make_edge( m1, m2 );
  make_edge( m1, m3 );
  make_edge( m2, m4 );
  make_edge( m3, m4 );


  graph g;
  function_node< int, int > a( g, unlimited, []( int i ) -> int {
      graph h;
      node_t n1( h, [=]( msg_t ) { cout << "n1: " << i << "\n"; } );
      node_t n2( h, [=]( msg_t ) { cout << "n2: " << i << "\n"; } );
      node_t n3( h, [=]( msg_t ) { cout << "n3: " << i << "\n"; } );
      node_t n4( h, [=]( msg_t ) { cout << "n4: " << i << "\n"; } );
      make_edge( n1, n2 );
      make_edge( n1, n3 );
      make_edge( n2, n4 );
      make_edge( n3, n4 );
      n1.try_put(continue_msg());
      h.wait_for_all();
      return i;
  } );
  function_node< int, int > b( g, unlimited, [&]( int i ) -> int {
      m1.try_put(i);
      h.wait_for_all(); // optional since h is not destroyed
      return i;
  } );
  make_edge( a, b );
  for ( int i = 0; i < 3; ++i ) {
      a.try_put(i);
  }
  g.wait_for_all();
```

It is only necessary to call `h.wait_for_all()` at the end of each invocation of `b`'s body in our modified code, if you wish for this `b`'s body to block until the inner graph is done. In the first implementation of `b`, it was necessary to call `h.wait_for_all` at the end of each invocation since the graph was destroyed at the end of the scope. So it would be valid in the body of `b` above to call `m1.try_put(i)` and then return without waiting for `h` to become idle.

Flow Graph Tips for Limiting Resource Consumption

You may want to control the number of messages allowed to enter parts of your graph, or control the maximum number of tasks in the work pool. There are several mechanisms available for limiting resource consumption in a flow graph.

- Using limiter_node
- Use Concurrency Limits
- Create a Token-Based System
- Attach Flow Graph to an Arbitrary Task Arena
    - Guiding Task Scheduler Execution
    - Work Isolation

Using limiter_node

One way to limit resource consumption is to use a limiter_node to set a limit on the number of messages that can flow through a given point in your graph. The constructor for a limiter node takes two arguments:

```
limiter_node( graph &g, size_t threshold )
```

The first argument is a reference to the graph it belongs to. The second argument sets the maximum number of items that should be allowed to pass through before the node starts rejecting incoming messages.

A limiter_node maintains an internal count of the messages that it has allowed to pass. When a message leaves the controlled part of the graph, a message can be sent to the decrement port on the `limiter_node` to decrement the count, allowing additional messages to pass through. In the example below, an `input_node` will generate `M` big objects. But the user wants to allow at most three big objects to reach the `function_node` at a time, and to prevent the `input_node` from generating all `M` big objects at once.

```
graph g;


int src_count = 0;
int number_of_objects = 0;
int max_objects = 3;


input_node< big_object * > s( g, [&]( oneapi::tbb::flow_control& fc ) -> big_object* {
    if ( src_count < M ) {
      big_object* v = new big_object();
      ++src_count;
      return v;
    } else {
      fc.stop();
      return nullptr;
    }
} );
s.activate();

limiter_node< big_object * > l( g, max_objects );


function_node< big_object *, continue_msg > f( g, unlimited,
  []( big_object *v ) -> continue_msg {
    spin_for(1);
    delete v;
    return continue_msg();
} );
```

```
make_edge( l, f );
make_edge( f, l.decrement );
make_edge( s, l );
g.wait_for_all();
```

The example above prevents the `input_node` from generating all `M` big objects at once. The `limiter_node` has a threshold of 3, and will therefore start rejecting incoming messages after its internal count reaches 3. When the `input_node` sees its message rejected, it stops calling its body object and temporarily buffers the last generated value. The `function_node` has its output, a `continue_msg`, sent to the decrement port of the `limiter_node`. So, after it completes executing, the `limiter_node` internal count is decremented. When the internal count drops below the threshold, messages begin flowing from the `input_node` again. So in this example, at most four big objects exist at a time, the three that have passed through the `limiter_node` and the one that is buffered in the `input_node`.

Use Concurrency Limits

To control the number of instances of a single node, you can use the concurrency limit on the node. To cause it to reject messages after it reaches its concurrency limit, you construct it as a “rejecting” node.

A function node is constructed with one or more template arguments. The third argument controls the buffer policy used by the node, and is by default queueing. With a queueing policy, a `function_node` that has reached its concurrency limit still accepts incoming messages, but buffers them internally. If the policy is set to rejecting the node will instead reject the incoming messages.

```
template < typename Input,
           typename Output = continue_msg,
           graph_buffer_policy = queueing >
class function_node;
```

For example, you can control the number of big objects in flight in a graph by placing a rejecting function_node downstream of an `input_node`, as is done below:

```
graph g;


int src_count = 0;
int number_of_objects = 0;
int max_objects = 3;


input_node< big_object * > s( g, [&]( oneapi::tbb::flow_control& fc ) -> big_object* {
    if ( src_count < M ) {
      big_object* v = new big_object();
      ++src_count;
      return v;
    } else {
      fc.stop();
      return nullptr;
    }
} );
s.activate();

function_node< big_object *, continue_msg, rejecting > f( g, 3,
    []( big_object *v ) -> continue_msg {
    spin_for(1);
       delete v;
    return continue_msg();
} );
```

```
make_edge( s, f );
g.wait_for_all();
```

The `function_node` will operate on at most three big objects concurrently. The node's concurrency threshold that limits the node to three concurrent invocations. When the `function_node` is running three instances concurrently, it will start rejecting incoming messages from the `input_node`, causing the `input_node` to buffer its last created object and temporarily stop invoking its body object. Whenever the `function_node` drops below its concurrency limit, it will pull new messages from the `input_node`. At most four big objects will exist simultaneously, three in the `function_node` and one buffered in the `input_node`.

Create a Token-Based System

A more flexible solution to limit the number of messages in a flow graph is to use tokens. In a token-based system, a limited number of tokens are available in the graph and a message will not be allowed to enter the graph until it can be paired with an available token. When a message is retired from the graph, its token is released, and can be paired with a new message that will then be allowed to enter.

The `oneapi::tbb::parallel_pipeline` algorithm relies on a token-based system. In the flow graph interface, there is no explicit support for tokens, but `join_node``s can be used to create an analogous system. A ``join_node` has two template arguments, the tuple that describes the types of its inputs and a buffer policy:

```
template<typename OutputTuple, graph_buffer_policy JP = queueing>
class join_node;
```

The buffer policy can be one of the following:

- `queueing`. This type of policy causes inputs to be matched first-in-first-out; that is, the inputs are joined together to form a tuple in the order they are received.
- `tag_matching`. This type of policy joins inputs together that have matching tags.
- `reserving`. This type of policy causes the `join_node` to do no internally buffering, but instead to consume inputs only when it can first reserve an input on each port from an upstream source. If it can reserve an input at each port, it gets those inputs and joins those together to form an output tuple.

A token-based system can be created by using reserving join_nodes.

In the example below, there is an `input_node` that generates `M` big objects and a `buffer_node` that is pre-filled with three tokens. The `token_t` can be anything, for example it could be `typedef int token_t;`. The `input_node` and `buffer_node` are connected to a reserving `join_node`. The `input_node` will only generate an input when one is pulled from it by the reserving `join_node`, and the reserving `join_node` will only pull the input from the `input_node` when it knows there is also an item to pull from the `buffer_node`.

```
graph g;


int src_count = 0;
int number_of_objects = 0;
int max_objects = 3;


input_node< big_object * > s( g, [&]( oneapi::tbb::flow_control& fc ) -> big_object* {
    if ( src_count < M ) {
      big_object* v = new big_object();
      ++src_count;
      return v;
    } else {
      fc.stop();
      return nullptr;
    }
} );
s.activate();

join_node< tuple_t, reserving > j(g);


buffer_node< token_t > b(g);


function_node< tuple_t, token_t > f( g, unlimited,
  []( const tuple_t &t ) -> token_t {
      spin_for(1);
   cout << get<1>(t) << "\n";
      delete get<0>(t);
   return get<1>(t);
} );


make_edge( s, input_port<0>(j) );
make_edge( b, input_port<1>(j) );
make_edge( j, f );
make_edge( f, b );


b.try_put( 1 );
b.try_put( 2 );
b.try_put( 3 );


g.wait_for_all();
```

In the above code, you can see that the `function_node` returns the token back to the `buffer_node`. This cycle in the flow graph allows the token to be recycled and paired with another input from the `input_node`. So like in the previous sections, there will be at most four big objects in the graph. There could be three big objects in the `function_node` and one buffered in the `input_node`, awaiting a token to be paired with.

Since there is no specific `token_t` defined for the flow graph, you can use any type for a token, including objects or pointers to arrays. Therefore, unlike in the example above, the `token_t` doesn't need to be a dummy type; it could for example be a buffer or other object that is essential to the computation. We could, for example, modify the example above to use the big objects themselves as the tokens, removing the need to repeatedly allocate and deallocate them, and essentially create a free list of big objects using a cycle back to the `buffer_node`.

Also, in our example above, the `buffer_node` was prefilled by a fixed number of explicit calls to `try_put`, but there are other options. For example, an `input_node` could be attached to the input of the `buffer_node`, and it could generate the tokens. In addition, our `function_node` could be replaced by a `multifunction_node` that can optionally put 0 or more outputs to each of its output ports. Using a `multifunction_node`, you can choose to recycle or not recycle a token, or even generate more tokens, thereby increasing or decreasing the allowed concurrency in the graph.

A token based system is therefore very flexible. You are free to declare the token to be of any type and to inject or remove tokens from the system as it is executing, thereby having dynamic control of the allowed concurrency in the system. Since you can pair the token with an input at the source, this approach enables you to limit resource consumption across the entire graph.

Attach Flow Graph to an Arbitrary Task Arena

oneTBB `task_arena` interface provides mechanisms to guide tasks execution within the arena by setting the preferred computation units, restricting part of computation units, or limiting arena concurrency. In some cases, you may want to apply such mechanisms when a flow graph executes.

During its construction, a `graph` object attaches to the arena, in which the constructing thread occupies a slot.

This example shows how to set the most performant core type as the preferred one for a graph execution:

```
std::vector<tbb::core_type_id> core_types = tbb::info::core_types();
tbb::task_arena arena(
    tbb::task_arena::constraints{}.set_core_type(core_types.back())
);

arena.execute( [&]() {
    graph g;
    function_node< int > f( g, unlimited, []( int ) {
         /*the most performant core type is defined as preferred.*/
    } );
    f.try_put(1);
    g.wait_for_all();
} );
```

A `graph` object can be reattached to a different `task_arena` by calling the `graph::reset()` function. It reinitializes and reattaches the `graph` to the task arena instance, inside which the `graph::reset()` method is executed.

This example shows how to reattach existing graph to an arena with the most performant core type as the preferred one for a work execution. Whenever a task is spawned on behalf of the graph, it is spawned in the arena of a graph it is attached to, disregarding the arena of the thread that the task is spawned from:

```
graph g;
function_node< int > f( g, unlimited, []( int ) {
    /*the most performant core type is defined as preferred.*/
} );

std::vector<tbb::core_type_id> core_types = tbb::info::core_types();
tbb::task_arena arena(
    tbb::task_arena::constraints{}.set_core_type(core_types.back())
);

arena.execute( [&]() {
    g.reset();
} );
f.try_put(1);
g.wait_for_all();
```

See the following topics to learn more:

- Guiding Task Scheduler Execution
- Work Isolation

Guiding Task Scheduler Execution

By default, the task scheduler tries to use all available computing resources. In some cases, you may want to configure the task scheduler to use only some of them.

Caution Guiding the execution of the task scheduler may cause composability issues.

Intel® oneAPI Threading Building Blocks (oneTBB) provides the `task_arena` interface to guide tasks execution within the arena by:

- setting the preferred computation units;
- restricting part of computation units.

Such customizations are encapsulated within the `task_arena::constraints` structure. To set the limitation, you have to customize the `task_arena::constraints` and then pass it to the `task_arena` instance during the construction or initialization.

The structure `task_arena::constraints` allows to specify the following restrictions:

- Preferred NUMA node
- Preferred core type
- The maximum number of logical threads scheduled per single core simultaneously
- The level of `task_arena` concurrency

You may use the interfaces from `tbb::info` namespace to construct the `tbb::task_arena::constraints` instance. Interfaces from `tbb::info` namespace respect the process affinity mask. For instance, if the process affinity mask excludes execution on some of the NUMA nodes, then these NUMA nodes are not returned by `tbb::info::numa_nodes()` interface.

The following examples show how to use these interfaces:

Setting the preferred NUMA node

The execution on systems with non-uniform memory access (NUMA systems) may cause a performance penalty if threads from one NUMA node access the memory allocated on a different NUMA node. To reduce this overhead, the work may be divided among several `task_arena` instances, whose execution preference is set to different NUMA nodes. To set execution preference, assign a NUMA node identifier to the `task_arena::constraints::numa_id` field.

```
std::vector<tbb::numa_node_id> numa_indexes = tbb::info::numa_nodes();
std::vector<tbb::task_arena> arenas(numa_indexes.size());
std::vector<tbb::task_group> task_groups(numa_indexes.size());

for(unsigned j = 0; j < numa_indexes.size(); j++) {
    arenas[j].initialize(tbb::task_arena::constraints(numa_indexes[j]));
    arenas[j].execute([&task_groups, &j](){
        task_groups[j].run([](){/*some parallel stuff*/});
    });
}

for(unsigned j = 0; j < numa_indexes.size(); j++) {
    arenas[j].execute([&task_groups, &j](){ task_groups[j].wait(); });
}
```

Setting the preferred core type

The processors with Intel® Hybrid Technology contain several core types, each is suited for different purposes. For example, some applications may improve their performance by preferring execution on the most performant cores. To set execution preference, assign specific core type identifier to the `task_arena::constraints::core_type` field.

The example shows how to set the most performant core type as preferable for work execution:

```
std::vector<tbb::core_type_id> core_types = tbb::info::core_types();
tbb::task_arena arena(
    tbb::task_arena::constraints{}.set_core_type(core_types.back())
);
```

```
arena.execute( [] {
    /*the most performant core type is defined as preferred.*/
});
```

Limiting the maximum number of threads simultaneously scheduled to one core

The processors with Intel® Hyper-Threading Technology allow more than one thread to run on each core simultaneously. However, there might be situations when there is need to lower the number of simultaneously running threads per core. In such cases, assign the desired value to the `task_arena::constraints::max_threads_per_core` field.

The example shows how to allow only one thread to run on each core at a time:

```
tbb::task_arena no_ht_arena( tbb::task_arena::constraints{}.set_max_threads_per_core(1) );
no_ht_arena.execute( [] {
    /*parallel work*/
});
```

A more composable way to limit the number of threads executing on cores is by setting the maximal concurrency of the `tbb::task_arena`:

```
int no_ht_concurrency = tbb::info::default_concurrency(
    tbb::task_arena::constraints{}.set_max_threads_per_core(1)
);
tbb::task_arena arena( no_ht_concurrency );
arena.execute( [] {
    /*parallel work*/
});
```

Similarly to the previous example, the number of threads inside the arena is equal to the number of available cores. However, this one results in fewer overheads and better composability by imposing a less constrained execution.

Work Isolation

In Intel® oneAPI Threading Building Blocks (oneTBB), a thread waiting for a group of tasks to complete might execute other available tasks. In particular, when a parallel construct calls another parallel construct, a thread can obtain a task from the outer-level construct while waiting for completion of the inner-level one.

In the following example with two `parallel_for` calls, the call to the second (nested) parallel loop blocks execution of the first (outer) loop iteration:

```
// The first parallel loop.
oneapi::tbb::parallel_for( 0, N1, []( int i ) {
    // The second parallel loop.
    oneapi::tbb::parallel_for( 0, N2, []( int j ) { /* Some work */ } );
} );
```

The blocked thread is allowed to take tasks belonging to the first parallel loop. As a result, two or more iterations of the outer loop might be simultaneously assigned to the same thread. In other words, in oneTBB execution of functions constituting a parallel construct is *unsequenced* even within a single thread. In most cases, this behavior is harmless or even beneficial because it does not restrict parallelism available for the thread.

However, in some cases such unsequenced execution may result in errors. For example, a thread-local variable might unexpectedly change its value after a nested parallel construct:

```
oneapi::tbb::enumerable_thread_specific<int> ets;
oneapi::tbb::parallel_for( 0, N1, [&ets]( int i ) {
    // Set a thread specific value
    ets.local() = i;
    oneapi::tbb::parallel_for( 0, N2, []( int j ) { /* Some work */ } );
    // While executing the above parallel_for, the thread might have run iterations
    // of the outer parallel_for, and so might have changed the thread specific value.
    assert( ets.local()==i ); // The assertion may fail!
} );
```

In other scenarios, the described behavior might lead to deadlocks and other issues. In these cases, a stronger guarantee of execution being sequenced within a thread is desired. For that, oneTBB provides ways to *isolate* execution of a parallel construct, for its tasks to not interfere with other simultaneously running tasks.

One of these ways is to execute the inner level loop in a separate `task_arena`:

```
oneapi::tbb::enumerable_thread_specific<int> ets;
oneapi::tbb::task_arena nested;
oneapi::tbb::parallel_for( 0, N1, [&]( int i ) {
    // Set a thread specific value
    ets.local() = i;
    nested.execute( []{
        // Run the inner parallel_for in a separate arena to prevent the thread
        // from taking tasks of the outer parallel_for.
        oneapi::tbb::parallel_for( 0, N2, []( int j ) { /* Some work */ } );
    } );
    assert( ets.local()==i ); // Valid assertion
} );
```

However, using a separate arena for work isolation is not always convenient, and might have noticeable overheads. To address these shortcomings, oneTBB provides `this_task_arena::isolate` function which runs a user-provided functor in isolation by restricting the calling thread to process only tasks scheduled in the scope of the functor (also called the isolation region).

When entered a task waiting call or a blocking parallel construct inside an isolated region, a thread can only execute tasks spawned within the region and their child tasks spawned by other threads. The thread is prohibited from executing any outer level tasks or tasks belonging to other isolated regions.

The isolation region imposes restrictions only upon the thread that called it. Other threads running in the same task arena have no restrictions on task selection unless isolated by a distinct call to `this_task_arena::isolate`.

The following example demonstrates the use of `this_task_arena::isolate` to ensure that a thread-local variable is not changed unexpectedly during the call to a nested parallel construct.

```
#include "oneapi/tbb/task_arena.h"
#include "oneapi/tbb/parallel_for.h"
#include "oneapi/tbb/enumerable_thread_specific.h"
#include <cassert>


int main() {
    const int N1 = 1000, N2 = 1000;
    oneapi::tbb::enumerable_thread_specific<int> ets;
    oneapi::tbb::parallel_for( 0, N1, [&ets]( int i ) {
        // Set a thread specific value
        ets.local() = i;
        // Run the second parallel loop in an isolated region to prevent the current thread
        // from taking tasks related to the outer parallel loop.
        oneapi::tbb::this_task_arena::isolate( []{
            oneapi::tbb::parallel_for( 0, N2, []( int j ) { /* Some work */ } );
        } );
        assert( ets.local()==i ); // Valid assertion
    } );
    return 0;
}
```

Flow Graph Tips for Exception Handling and Cancellation

The execution of a flow graph can be canceled directly or as a result of an exception that propagates beyond a node's body. You can then optionally reset the graph so that it can be re-executed.

- Catching Exceptions Inside the Node that Throws the Exception
- Cancel a Graph Explicitly
- Use graph::reset() to Reset a Canceled Graph
- Canceling Nested Parallelism

Catching Exceptions Inside the Node that Throws the Exception

If you catch an exception within the node's body, execution continues normally, as you might expect. If an exception is thrown but is not caught before it propagates beyond the node's body, the execution of all of the graph's nodes are canceled and the exception is rethrown at the call site of graph::wait_for_all(). Take the graph below as an example:

```
graph g;


function_node< int, int > f1( g, 1, []( int i ) {  return i; } );


function_node< int, int > f2( g, 1,
    []( const int i ) -> int {
    throw i;
    return i;
} );


function_node< int, int > f3( g, 1, []( int i ) {  return i; } );


make_edge( f1, f2 );
make_edge( f2, f3 );
f1.try_put(1);
f1.try_put(2);
g.wait_for_all();
```

In the code above, the second function_node, f2, throws an exception that is not caught within the body. This will cause the execution of the graph to be canceled and the exception to be rethrown at the call to g.wait_for_all(). Since it is not handled there either, the program will terminate. If desirable, the exception could be caught and handled within the body:

```
function_node< int, int > f2( g, 1,
    []( const int i ) -> int {
        try {
            throw i;
        } catch (int j) {
            cout << "Caught " << j << "\n";
        }
        return i;
} );
```

If the exception is caught and handled in the body, then there is no effect on the overall execution of the graph. However, you could choose instead to catch the exception at the call to wait_for_all:

```
try {
    g.wait_for_all();
} catch ( int j ) {
    cout << "Caught " << j << "\n";
}
```

In this case, the execution of the graph is canceled. For our example, this means that the input 1 never reaches f3 and that input 2 never reaches either f2 or f3.

Cancel a Graph Explicitly

To cancel a graph execution without an exception, you can create the graph using an explicit task_group_context, and then call cancel_group_execution() on that object. This is done in the example below:

```
task_group_context t;
graph g(t);


function_node< int, int > f1( g, 1, []( int i ) {  return i; } );


function_node< int, int > f2( g, 1,
    []( const int i ) -> int {
        cout << "Begin " << i << "\n";
        spin_for(0.2);
        cout << "End " << i << "\n";
        return i;
} );


function_node< int, int > f3( g, 1, []( int i ) {  return i; } );


make_edge( f1, f2 );
make_edge( f2, f3 );
f1.try_put(1);
f1.try_put(2);
spin_for(0.1);
t.cancel_group_execution();
g.wait_for_all();
```

When a graph execution is canceled, any node that has already started to execute will execute to completion, but any node that has not started to execute will not start. So in the example above, f2 will print both the Begin and End message for input 1, but will not receive the input 2.

You can also get the task_group_context that a node belongs to from within the node body and use it to cancel the execution of the graph it belongs to:

```
graph g;


function_node< int, int > f1( g, 1, []( int i ) {  return i; } );


function_node< int, int > f2( g, 1,
    []( const int i ) -> int {
     cout << "Begin " << i << "\n";
     spin_for(0.2);
        cout << "End " << i << "\n";
        task::self().group()->cancel_group_execution();
        return i;
} );


function_node< int, int > f3( g, 1, []( int i ) {  return i; } );


make_edge( f1, f2 );
make_edge( f2, f3 );
f1.try_put(1);
f1.try_put(2);
g.wait_for_all();
```

You can get the task_group_context from a node's body even if the graph was not explicitly passed one at construction time.

Use graph::reset() to Reset a Canceled Graph

When a graph execution is canceled either because of an unhandled exception or because its task_group_context is canceled explicitly, the graph and its nodes may be left in an indeterminate state. For example, in the code samples shown in Cancel a Graph Explicitly the input 2 may be left in a buffer. But even beyond remnants in the buffers, there are other optimizations performed during the execution of a flow graph that can leave its nodes and edges in an indeterminate state. If you want to re-execute or restart a graph, you first need to reset the graph:

```
try {
    g.wait_for_all();
} catch ( int j ) {
    cout << "Caught " << j << "\n";
    // do something to fix the problem
    g.reset();
    f1.try_put(1);
    f1.try_put(2);
    g.wait_for_all();
}
```

Canceling Nested Parallelism

Nested parallelism is canceled if the inner context is bound to the outer context; otherwise it is not.

If the execution of a flow graph is canceled, either explicitly or due to an exception, any tasks started by parallel algorithms or flow graphs nested within the nodes of the canceled flow graph may or may not be canceled.

As with all of the library's nested parallelism, you can control cancellation relationships by use of explicit task_group_context objects. If you do not provide an explicit task_group_context to a flow graph, it is created with an isolated context by default.

Estimating Flow Graph Performance

The performance or scalability of a flow graph is not easy to predict. However there are a few key points that can guide you in estimating the limits on performance and speedup of some graphs.

The Critical Path Limits the Scalability in a Dependence Graph

A critical path is the most time consuming path from a node with no predecessors to a node with no successors. In a dependence graph, the execution of the nodes along a path cannot be overlapped since they have a strict ordering. Therefore, for a dependence graph, the critical path limits scalability.

More formally, let T be the total time consumed by all of the nodes in your graph if executed sequentially. Then let C be the time consumed along the path that takes the most time. The nodes along this path cannot be overlapped even in a parallel execution. Therefore, even if all other paths are executed in parallel with C, the wall clock time for the parallel execution is at least C, and the maximum possible speedup (ignoring microarchitectural and memory effects) is T/C.

There is Overhead in Spawning a Node's Body as a Task

The bodies of `input_nodes`, `function_nodes`, `continue_nodes` and `multifunction_nodes` execute within spawned tasks by default. This means that you need to take into account the overhead of task scheduling when estimating the time it takes for a node to execute its body. All of the rules of thumb for determining the appropriate granularity of tasks therefore also apply to node bodies as well. If you have many fine-grained nodes in your flow graph, the impact of these overheads can noticeably impact your performance. However, depending on the graph structure, you can reduce such overheads by using lightweight policy with these nodes.

Work Isolation

In Intel® oneAPI Threading Building Blocks (oneTBB), a thread waiting for a group of tasks to complete might execute other available tasks. In particular, when a parallel construct calls another parallel construct, a thread can obtain a task from the outer-level construct while waiting for completion of the inner-level one.

In the following example with two `parallel_for` calls, the call to the second (nested) parallel loop blocks execution of the first (outer) loop iteration:

```
// The first parallel loop.
oneapi::tbb::parallel_for( 0, N1, []( int i ) {
    // The second parallel loop.
    oneapi::tbb::parallel_for( 0, N2, []( int j ) { /* Some work */ } );
} );
```

The blocked thread is allowed to take tasks belonging to the first parallel loop. As a result, two or more iterations of the outer loop might be simultaneously assigned to the same thread. In other words, in oneTBB execution of functions constituting a parallel construct is *unsequenced* even within a single thread. In most cases, this behavior is harmless or even beneficial because it does not restrict parallelism available for the thread.

However, in some cases such unsequenced execution may result in errors. For example, a thread-local variable might unexpectedly change its value after a nested parallel construct:

```
oneapi::tbb::enumerable_thread_specific<int> ets;
oneapi::tbb::parallel_for( 0, N1, [&ets]( int i ) {
    // Set a thread specific value
    ets.local() = i;
    oneapi::tbb::parallel_for( 0, N2, []( int j ) { /* Some work */ } );
    // While executing the above parallel_for, the thread might have run iterations
    // of the outer parallel_for, and so might have changed the thread specific value.
    assert( ets.local()==i ); // The assertion may fail!
} );
```

In other scenarios, the described behavior might lead to deadlocks and other issues. In these cases, a stronger guarantee of execution being sequenced within a thread is desired. For that, oneTBB provides ways to *isolate* execution of a parallel construct, for its tasks to not interfere with other simultaneously running tasks.

One of these ways is to execute the inner level loop in a separate `task_arena`:

```
oneapi::tbb::enumerable_thread_specific<int> ets;
oneapi::tbb::task_arena nested;
oneapi::tbb::parallel_for( 0, N1, [&]( int i ) {
    // Set a thread specific value
    ets.local() = i;
    nested.execute( []{
        // Run the inner parallel_for in a separate arena to prevent the thread
        // from taking tasks of the outer parallel_for.
        oneapi::tbb::parallel_for( 0, N2, []( int j ) { /* Some work */ } );
    } );
    assert( ets.local()==i ); // Valid assertion
} );
```

However, using a separate arena for work isolation is not always convenient, and might have noticeable overheads. To address these shortcomings, oneTBB provides `this_task_arena::isolate` function which runs a user-provided functor in isolation by restricting the calling thread to process only tasks scheduled in the scope of the functor (also called the isolation region).

When entered a task waiting call or a blocking parallel construct inside an isolated region, a thread can only execute tasks spawned within the region and their child tasks spawned by other threads. The thread is prohibited from executing any outer level tasks or tasks belonging to other isolated regions.

The isolation region imposes restrictions only upon the thread that called it. Other threads running in the same task arena have no restrictions on task selection unless isolated by a distinct call to `this_task_arena::isolate`.

The following example demonstrates the use of `this_task_arena::isolate` to ensure that a thread-local variable is not changed unexpectedly during the call to a nested parallel construct.

```
#include "oneapi/tbb/task_arena.h"
#include "oneapi/tbb/parallel_for.h"
#include "oneapi/tbb/enumerable_thread_specific.h"
#include <cassert>


int main() {
    const int N1 = 1000, N2 = 1000;
    oneapi::tbb::enumerable_thread_specific<int> ets;
    oneapi::tbb::parallel_for( 0, N1, [&ets]( int i ) {
        // Set a thread specific value
        ets.local() = i;
        // Run the second parallel loop in an isolated region to prevent the current thread
        // from taking tasks related to the outer parallel loop.
        oneapi::tbb::this_task_arena::isolate( []{
            oneapi::tbb::parallel_for( 0, N2, []( int j ) { /* Some work */ } );
        } );
        assert( ets.local()==i ); // Valid assertion
    } );
    return 0;
}
```

Exceptions and Cancellation

Intel® oneAPI Threading Building Blocks (oneTBB) supports exceptions and cancellation. When code inside an oneTBB algorithm throws an exception, the following steps generally occur:

1. The exception is captured. Any further exceptions inside the algorithm are ignored.

2. The algorithm is cancelled. Pending iterations are not executed. If there is oneTBB parallelism nested inside, the nested parallelism may also be cancelled as explained in Cancellation and Nested Parallelism.
3. Once all parts of the algorithm stop, an exception is thrown on the thread that invoked the algorithm.

The exception thrown in step 3 might be the original exception, or might merely be a summary of type `captured_exception`. The latter usually occurs on current systems because propagating exceptions between threads requires support for the C++ `std::exception_ptr` functionality. As compilers evolve to support this functionality, future versions of oneTBB might throw the original exception. So be sure your code can catch either type of exception. The following example demonstrates exception handling.

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


using namespace oneapi::tbb;
using namespace std;


vector<int> Data;


struct Update {
    void operator()( const blocked_range<int>& r ) const {
        for( int i=r.begin(); i!=r.end(); ++i )
            Data.at(i) += 1;
    }
};


int main() {
    Data.resize(1000);
    try {
        parallel_for( blocked_range<int>(0, 2000), Update());
    } catch( out_of_range& ex ) {
       cout << "out_of_range: " << ex.what() << endl;
    }
    return 0;
}
```

The `parallel_for` attempts to iterate over 2000 elements of a vector with only 1000 elements. Hence the expression `Data.at(i)` sometimes throws an exception `std::out_of_range` during execution of the algorithm. When the exception happens, the algorithm is cancelled and an exception thrown at the call site to `parallel_for`.

- Cancellation Without An Exception
- Cancellation and Nested Parallelism

Cancellation Without An Exception

To cancel an algorithm but not throw an exception, use the expression `current_context()->cancel_group_execution()`. The part `current_context()` references the `task_group_context*` of the currently executing task if any on the current thread. Calling `cancel_group_execution()` cancels all tasks in its `task_group_context`, which is explained in more detail in Cancellation and Nested Parallelism. The method returns `true` if it actually causes cancellation, `false` if the `task_group_context` was already cancelled.

The example below shows how to use `current_context()->cancel_group_execution()`.

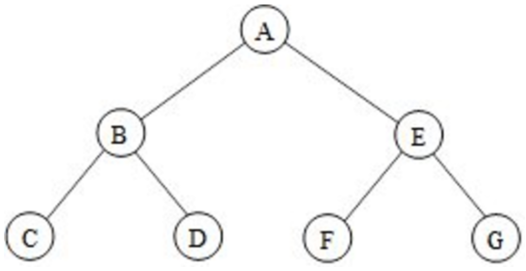
```
#include "oneapi/tbb.h"

#include <vector>
#include <iostream>

using namespace oneapi::tbb;
using namespace std;

vector<int> Data;

struct Update {
    void operator()( const blocked_range<int>& r ) const {
        for( int i=r.begin(); i!=r.end(); ++i )
            if( i<Data.size() ) {
                ++Data[i];
            } else {
                // Cancel related tasks.
                if( current_context()->cancel_group_execution() )
                    cout << "Index " << i << " caused cancellation\n";
                return;
            }
    }
};


int main() {
    Data.resize(1000);
    parallel_for( blocked_range<int>(0, 2000), Update());
    return 0;
}
```

Cancellation and Nested Parallelism

The discussion so far was simplified by assuming non-nested parallelism and skipping details of `task_group_context`. This topic explains both.

An Intel® oneAPI Threading Building Blocks (oneTBB) algorithm executes by creating `task` objects that execute the snippets of code that you supply to the algorithm template. By default, these `task` objects are associated with a `task_group_context` created by the algorithm. Nested oneTBB algorithms create a tree of these `task_group_context` objects. Cancelling a `task_group_context` cancels all of its child `task_group_context` objects, and transitively all its descendants. Hence an algorithm and all algorithms it called can be cancelled with a single request.

Exceptions propagate upwards. Cancellation propagates downwards. The opposition interplays to cleanly stop a nested computation when an exception occurs. For example, consider the tree in the following figure. Imagine that each node represents an algorithm and its `task_group_context`.



Tree of task_group_context

Suppose that the algorithm in C throws an exception and no node catches the exception. oneTBB propagates the exception upwards, cancelling related subtrees downwards, as follows:

1. Handle exception in C:
   - a. Capture exception in C.
   - b. Cancel tasks in C.
   - c. Throw exception from C to B.
2. Handle exception in B:
   - a. Capture exception in B.
   - b. Cancel tasks in B and, by downwards propagation, in D.
   - c. Throw an exception out of B to A.
3. Handle exception in A:
   - a. Capture exception in A.
   - b. Cancel tasks in A and, by downwards propagation, in E, F, and G.
   - c. Throw an exception upwards out of A.

If your code catches the exception at any level, then oneTBB does not propagate it any further. For example, an exception that does not escape outside the body of a `parallel_for` does not cause cancellation of other iterations.

To prevent downwards propagation of cancellation into an algorithm, construct an 'isolated' `task_group_context` on the stack and pass it to the algorithm explicitly. The example uses C++11 lambda expressions for brevity.

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


bool Data[1000][1000];


int main() {
    try {
        parallel_for( 0, 1000, 1,
            []( int i ) {
                task_group_context root(task_group_context::isolated);
                parallel_for( 0, 1000, 1,
                   []( int  ) {
                       Data[i][j] = true;
                   },
                   root);
                throw "oops";
            });
    } catch(...) {
    }
    return 0;
}
```

The example performs two parallel loops: an outer loop over `i` and inner loop over `j`. The creation of the isolated `task_group_contextroot` protects the inner loop from downwards propagation of cancellation from the `i` loop. When the exception propagates to the outer loop, any pending `outer` iterations are cancelled, but not inner iterations for an outer iteration that started. Hence when the program completes, each row of `Data` may be different, depending upon whether its iteration `i` ran at all, but within a row, the elements will be homogeneously `false` or `true`, not a mixture.

Removing the blue text would permit cancellation to propagate down into the inner loop. In that case, a row of `Data` might end up with both `true` and `false` values.

Containers

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

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 oneTBB offer a much higher level of concurrency, via one or both of the following methods:

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

Notice that highly-concurrent containers 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.

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


using namespace oneapi::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`.

- More on HashCompare

More on HashCompare

There are several ways to make the `HashCompare` argument for `concurrent_hash_map` work for your own types.

- Specify the `HashCompare` argument explicitly
- Let the `HashCompare` default to `tbb_hash_compare<Key>` and do one of the following:
  - Define a specialization of template `tbb_hash_compare<Key>`.

For example, if you have keys of type `Foo`, and `operator==` is defined for `Foo`, you just have to provide a definition of `tbb_hasher` as shown below:

```
size_t tbb_hasher(const Foo& f) {
    size_t h = ...compute hash code for f...
    return h;
};
```

In general, the definition of `tbb_hash_compare<Key>` or `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 go 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));
```

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 three methods that support common uses of dynamic arrays: `push_back`, `grow_by`, and `grow_to_at_least`.

Method `push_back(x)` safely appends x to the array. Method `grow_by(n)` safely appends `n` consecutive elements initialized with `T()`. Both methods return an iterator pointing to the first appended element. Each element is initialized with `T()`. 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.grow_by(n) );
}
```

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

Method `size()` returns the number of elements in the vector, which may include elements that are still undergoing concurrent construction by methods `push_back`, `grow_by`, or `grow_to_at_least`. The 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.

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`.

Concurrent Queue Classes

Template class `concurrent_queue<T,Alloc>` implements a concurrent queue with values of type `T`. Multiple threads may simultaneously push and pop elements from the queue. The queue is unbounded and has no blocking operations. The fundamental operations on it are `push` and `try_pop`. The `push` operation works just like `push` for a std::queue. The operation `try_pop` pops an item if it is available. The check and popping have to be done in a single operation for sake of thread safety.

For example, consider the following serial code:

```
extern std::queue<T> MySerialQueue;
T item;
if( !MySerialQueue.empty() ) {
    item = MySerialQueue.front();
    MySerialQueue.pop_front();
    ... process item...
}
```

Even if each std::queue method were implemented in a thread-safe manner, the composition of those methods as shown in the example would not be thread safe if there were other threads also popping from the same queue. For example, `MySerialQueue.empty()` might return true just before another thread snatches the last item from `MySerialQueue`.

The equivalent thread-safe Intel® oneAPI Threading Building Blocks (oneTBB) code is:

```
extern concurrent_queue<T> MyQueue;
T item;
if( MyQueue.try_pop(item) ) {
    ...process item...
}
```

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 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.

Template class `concurrent_queue` is unbounded and has no methods that wait. It is up to the user to provide synchronization to avoid overflow, or to wait for the queue to become non-empty. Typically this is appropriate when the synchronization has to be done at a higher level.

Template class `concurrent_bounded_queue<T,Alloc>` is a variant that adds blocking operations and the ability to specify a capacity. The methods of particular interest on it are:

- `pop(item)` waits until it can succeed.
- `push(item)` waits until it can succeed without exceeding the queue’s capacity.
- `try_push(item)` pushes `item` only if it would not exceed the queue’s capacity.
- size() returns a *signed* integer.

The value of 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_bounded_queue` 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. If you do not need the bounds or the blocking pop, consider using `concurrent_queue` instead.

- Iterating Over a Concurrent Queue for Debugging
- When Not to Use Queues

Iterating Over a Concurrent Queue for Debugging

The template classes `concurrent_queue` and `concurrent_bounded_queue` support 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;
...
typedef concurrent_queue<Foo>::const_iterator iter;
for(iter i(q.unsafe_begin()); i!=q.unsafe_end(); ++i ) {
    cout << *i;
}
```

The prefix `unsafe_` on the methods is a reminder that they are not concurrency safe.

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_for_eachparallel_pipeline` instead. These is 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_pipeline` avoids these bottlenecks. Because its threading is implicit, it optimizes use of worker threads so that they do other work until a value shows up. It also tries to keep items hot in cache.

Summary of Containers

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

Mutual Exclusion

Mutual exclusion controls how many threads can simultaneously run a region of code. In Intel® oneAPI Threading Building Blocks (oneTBB), 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.

```
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.

The following shows an alternative way to write `AllocateNode`:

```
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 oneTBB 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.

- Mutex Flavors
- Reader Writer Mutexes
- Upgrade/Downgrade
- Lock Pathologies

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 the table below.

- **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.
- **Yield or Block**. This is an implementation detail that impacts performance. On long waits, an Intel® oneAPI Threading Building Blocks (oneTBB) mutex either *yields* or *blocks*. Here *yields* means to repeatedly poll whether progress can be made, and if not, temporarily yield [1] the processor. To *block* means to yield the processor until the mutex permits progress. Use the yielding 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.
- `mutex` has behavior similar to the `spin_mutex`. However, the `mutex`*blocks* on long waits that makes it resistant to high contention.
- `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.
- `rw_mutex` is similar to `mutex`, but additionally support *reader* locks.
- `speculative_spin_mutex` and `speculative_spin_rw_mutex` are similar to `spin_mutex` and `spin_rw_mutex`, but additionally provide *speculative locking* on processors that support hardware transaction memory. Speculative locking allows multiple threads acquire the same lock, as long as there are no “conflicts” that may generate different results than non-speculative locking. These mutexes are *scalable* when work with low conflict rate, i.e. mostly in speculative locking mode.

- 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 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.

Mutex	Scalable	Fair	Recursive	Long Wait	Size
spin_mutex	no	no	no	yields	1 byte
mutex	✓	no	no	blocks	1 byte
speculative_spin_mutex	HW dependent	no	no	yields	2 cache lines
queuing_mutex	✓	✓	no	yields	1 word
spin_rw_mutex	no	no	no	yields	1 word
spin_rw_mutex	✓	no	no	blocks	1 word
speculative_spin_rw_mutex	HW dependent	no	no	yields	3 cache lines
queuing_rw_mutex	✓	✓	no	yields	1 word
null_mutex[2]	moot	✓	✓	never	empty
null_rw_mutex	moot	✓	✓	never	empty

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

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

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.

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( !lock.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 **Lock Pathologies**. 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.

Lock Pathologies

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

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.

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.

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® oneAPI Threading Building Blocks (oneTBB) 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.

NOTE On Linux* OS, you may need to add -lrt to the linker command when you use oneapi::tbb::tick_count class. For more information, see http://fedoraproject.org/wiki/Features/ChangeInImplicitDSOLinking.

Memory Allocation

Intel® oneAPI Threading Building Blocks (oneTBB) provides several memory allocator templates that are similar to the STL template class std::allocator. 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 `scalable_allocator<T>` template to avoid 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 the same cache line, the 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 `cache_aligned_allocator<T>` template to always allocate on a separate cache line. Two objects allocated by `cache_aligned_allocator` are guaranteed to not have false sharing. However, if an object is allocated by `cache_aligned_allocator` and another object is allocated some other way, there is no guarantee.

You can use these allocator templates 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 also provides a set of functions equivalent to the C standard library memory management routines but has the `scalable_` prefix in their names, as well as the way to easily redirect the standard routines to these functions.

- Which Dynamic Libraries to Use
- Configuring the Memory Allocator
- Automatically Replacing `malloc` and Other C/C++ Functions for Dynamic Memory Allocation
  - Windows* OS C/C++ Dynamic Memory Interface Replacement
  - Linux* OS C/C++ Dynamic Memory Interface Replacement

Which Dynamic Libraries to Use

The template `scalable_allocator<T>` requires the Intel® oneAPI Threading Building Blocks (oneTBB) scalable memory allocator library as described in **Scalable Memory Allocator**. It does not require the oneTBB general library, and can be used independently of the rest of oneTBB.

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® oneAPI Threading Building Blocks (oneTBB) can be used with or without the oneTBB scalable memory allocator library.

Template	Requirements	Notes
`scalable_allocator<T>`	Intel® oneAPI Threading Building Blocks (oneTBB) scalable memory allocator library. See **Scalable Memory Allocator**.	
`tbb_allocator<T>cache_aligned_allocator<T>`		Uses the scalable allocator library if it is present, otherwise it reverts to using `malloc` and `free`.

Configuring the Memory Allocator

The oneTBB memory allocator provides the following API functions and environment variables to configure its behavior:

- the `scalable_allocation_command` function instructs the allocator to perform a certain action, such as cleaning up its internal memory buffers.

- the `scalable_allocation_mode` function allows an application to set certain parameters for the memory allocator, such as an option to map memory in huge pages or define a recommended heap size. These settings take effect until modified by another call to `scalable_allocation_mode`.

Some of the memory allocator parameters can also be set via system environment variables. It can be useful to adjust the behavior without modifying application source code, to ensure that a setting takes effect as early as possible, or to avoid explicit dependency on the oneTBB allocator binaries. The following environment variables are recognized:

- `TBB_MALLOC_USE_HUGE_PAGES` controls usage of huge pages for memory mapping.
- `TBB_MALLOC_SET_HUGE_OBJECT_THRESHOLD` defines the lower bound for the size (bytes), that is interpreted as huge and not released during regular cleanup operations.

These variables only take effect at the time the memory manager is initialized; later environment changes are ignored. A call to `scalable_allocation_mode` overrides the effect of the corresponding environment variable.

Automatically Replacing malloc and Other C/C++ Functions for Dynamic Memory Allocation

On Windows*, Linux* operating systems, it is possible to automatically replace all calls to standard functions for dynamic memory allocation (such as `malloc`) with the Intel® oneAPI Threading Building Blocks (oneTBB) scalable equivalents. Doing so can sometimes improve application performance.

Replacements are provided by the proxy library (the library names can be found in platform-specific sections below). A proxy library and a scalable memory allocator library should be taken from the same release of oneTBB, otherwise the libraries may be mutually incompatible.

- Windows* OS C/C++ Dynamic Memory Interface Replacement
- Linux* OS C/C++ Dynamic Memory Interface Replacement

Windows* OS C/C++ Dynamic Memory Interface Replacement

Release version of the proxy library is `tbbmalloc_proxy.dll`, debug version is `tbbmalloc_proxy_debug.dll`.

The following dynamic memory functions are replaced:

- Standard C library functions: `malloc`, `calloc`, `realloc`, `free`
- Replaceable global C++ operators `new` and `delete`
- Microsoft* C run-time library functions: `_msize`, `_aligned_malloc`, `_aligned_realloc`, `_aligned_free`, `_aligned_msize`

NOTE Replacement of memory allocation functions is not supported for Universal Windows Platform applications.

To do the replacement use one of the following methods:

- Add the following header to a source code of any binary which is loaded during application startup.

```
#include "oneapi/tbb/tbbmalloc_proxy.h"
```

- Alternatively, add the following parameters to the linker options for the .exe or .dll file that is loaded during application startup.

  For 32-bit code (note the triple underscore):

```
tbbmalloc_proxy.lib /INCLUDE:"___TBB_malloc_proxy"
```

  For 64-bit code (note the double underscore):

```
tbbmalloc_proxy.lib /INCLUDE:"__TBB_malloc_proxy"
```

The OS program loader must be able to find the proxy library and the scalable memory allocator library at program load time. For that you may include the directory containing the libraries in the `PATH` environment variable.

The replacement uses in-memory binary instrumentation of Visual C++* runtime libraries. To ensure correctness, it must first recognize a subset of dynamic memory functions in these libraries. If a problem occurs, the replacement is skipped, and the program continues to use the standard memory allocation functions. You can use the `TBB_malloc_replacement_log` function to check if the replacement has succeeded and to get additional information.

Set the `TBB_MALLOC_DISABLE_REPLACEMENT` environment variable to 1 to disable replacement for a specific program invocation. In this case, the program will use standard dynamic memory allocation functions. Note that the oneTBB memory allocation libraries are still required for the program to start even if their usage is disabled.

Linux* OS C/C++ Dynamic Memory Interface Replacement

Release version of the proxy library is `libtbbmalloc_proxy.so`, debug version is `libtbbmalloc_proxy_debug.so`.

The following dynamic memory functions are replaced:

- Standard C library functions: `malloc`, `calloc`, `realloc`, `free`, (added in C11) `aligned_alloc`
- Standard POSIX* function: `posix_memalign`
- Obsolete functions: `valloc`, `memalign`, `pvalloc`, `mallopt`
- Replaceable global C++ operators `new` and `delete`
- GNU C library (glibc) specific functions: `malloc_usable_size`, `__libc_malloc`, `__libc_calloc`, `__libc_memalign`, `__libc_free`, `__libc_realloc`, `__libc_pvalloc`, `__libc_valloc`

You can do the replacement either by loading the proxy library at program load time using the `LD_PRELOAD` environment variable (without changing the executable file), or by linking the main executable file with the proxy library.

The OS program loader must be able to find the proxy library and the scalable memory allocator library at program load time. For that you may include the directory containing the libraries in the `LD_LIBRARY_PATH` environment variable or add it to `/etc/ld.so.conf`.

There are limitations for dynamic memory replacement:

- glibc memory allocation hooks, such as `__malloc_hook`, are not supported.
- Mono is not supported.

Examples

These examples show how to set `LD_PRELOAD` and how to link a program to use the memory allocation replacements.

```
# Set LD_PRELOAD to load the release version of the proxy library
LD_PRELOAD=libtbbmalloc_proxy.so
# Link with the release version of the proxy library
g++ foo.o bar.o -ltbbmalloc_proxy -o a.out
```

To use the debug version of the library, replace *tbbmalloc_proxy* with *tbbmalloc_proxy_debug* in the above examples.

The Task Scheduler

This section introduces the Intel® oneAPI Threading Building Blocks (oneTBB) *task scheduler*. The task scheduler is the engine that powers the loop templates. When practical, 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.

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® oneAPI Threading Building Blocks (oneTBB), 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 oneTBB 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.

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.

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.

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 on the logical dependences between tasks, and leave the efficient scheduling to the scheduler.

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® oneAPI Threading Building Blocks (oneTBB) tasks.

How Task Scheduler Works

While the task scheduler is not bound to any particular type of parallelism, it was designed to work efficiently for fork-join parallelism with lots of forks. This type of parallelism is typical for parallel algorithms such as oneapi::tbb::parallel_for.

Let’s consider the mapping of fork-join parallelism on the task scheduler in more detail.

The scheduler runs tasks in a way that tries to achieve several targets simultaneously:	• Enable as many threads as possible, by creating enough job, to achieve actual parallelism • Preserve data locality to make a single thread execution more efficient • Minimize both memory demands and cross-thread communication to reduce an overhead

To achieve this, a balance between depth-first and breadth-first execution strategies must be reached. Assuming that the task graph is finite, depth-first is better for a sequential execution because:

- **Strike when the cache is hot**. The deepest tasks are the most recently created tasks and therefore are the hottest in the cache. Also, if they can be completed, tasks that depend on it can continue executing, and though not the hottest in a cache, they are still warmer than the older tasks deeper in the dequeue.
- **Minimize space**. Execution of the shallowest task leads to the breadth-first unfolding of a graph. It creates an exponential number of nodes that co-exist simultaneously. In contrast, depth-first execution creates the same number of nodes, but only a linear number can exists at the same time, since it creates a stack of other ready tasks.

Each thread has its deque of tasks that are ready to run. When a thread spawns a task, it pushes it onto the bottom of its deque.

When a thread participates in the evaluation of tasks, it constantly executes a task obtained by the first rule that applies from the roughly equivalent ruleset:

- Get the task returned by the previous one, if any.
- Take a task from the bottom of its deque, if any.
- Steal a task from the top of another randomly chosen deque. If the selected deque is empty, the thread tries again to execute this rule until it succeeds.

Rule 1 is described in Task Scheduler Bypass. The overall effect of rule 2 is to execute the *youngest* task spawned by the thread, which causes the depth-first execution until the thread runs out of work. Then rule 3 applies. It steals the *oldest* task spawned by another thread, which causes temporary breadth-first execution that converts potential parallelism into actual parallelism.

Task Scheduler Bypass

Scheduler bypass is an optimization where you directly specify the next task to run. According to the rules of execution described in How Task Scheduler Works, the spawning of the new task to be executed by the current thread involves the next steps:

- Push a new task onto the thread’s deque.
- Continue to execute the current task until it is completed.
- Take a task from the thread’s deque, unless it is stolen by another thread.

Steps 1 and 3 introduce unnecessary deque operations or, even worse, allow stealing that can hurt locality without adding significant parallelism. These problems can be avoided by using “Task Scheduler Bypass” technique to directly point the preferable task to be executed next instead of spawning it. When, as described in How Task Scheduler Works, the returned task becomes the first candidate for the next task to be executed by the thread. Furthermore, this approach almost guarantees that the task is executed by the current thread and not by any other thread.

Please note that at the moment the only way to use this optimization is to use **preview feature of ``onepai::tbb::task_group`**

Guiding Task Scheduler Execution

By default, the task scheduler tries to use all available computing resources. In some cases, you may want to configure the task scheduler to use only some of them.

Caution Guiding the execution of the task scheduler may cause composability issues.

Intel® oneAPI Threading Building Blocks (oneTBB) provides the `task_arena` interface to guide tasks execution within the arena by:

- setting the preferred computation units;
- restricting part of computation units.

Such customizations are encapsulated within the `task_arena::constraints` structure. To set the limitation, you have to customize the `task_arena::constraints` and then pass it to the `task_arena` instance during the construction or initialization.

The structure `task_arena::constraints` allows to specify the following restrictions:

- Preferred NUMA node
- Preferred core type
- The maximum number of logical threads scheduled per single core simultaneously
- The level of `task_arena` concurrency

You may use the interfaces from `tbb::info` namespace to construct the `tbb::task_arena::constraints` instance. Interfaces from `tbb::info` namespace respect the process affinity mask. For instance, if the process affinity mask excludes execution on some of the NUMA nodes, then these NUMA nodes are not returned by `tbb::info::numa_nodes()` interface.

The following examples show how to use these interfaces:

Setting the preferred NUMA node

The execution on systems with non-uniform memory access (NUMA systems) may cause a performance penalty if threads from one NUMA node access the memory allocated on a different NUMA node. To reduce this overhead, the work may be divided among several `task_arena` instances, whose execution preference is set to different NUMA nodes. To set execution preference, assign a NUMA node identifier to the `task_arena::constraints::numa_id` field.

```
std::vector<tbb::numa_node_id> numa_indexes = tbb::info::numa_nodes();
std::vector<tbb::task_arena> arenas(numa_indexes.size());
std::vector<tbb::task_group> task_groups(numa_indexes.size());

for(unsigned j = 0; j < numa_indexes.size(); j++) {
    arenas[j].initialize(tbb::task_arena::constraints(numa_indexes[j]));
    arenas[j].execute([&task_groups, &j](){
        task_groups[j].run([](){/*some parallel stuff*/});
    });
}
```

```
for(unsigned j = 0; j < numa_indexes.size(); j++) {
    arenas[j].execute([&task_groups, &j](){ task_groups[j].wait(); });
}
```

Setting the preferred core type

The processors with Intel® Hybrid Technology contain several core types, each is suited for different purposes. For example, some applications may improve their performance by preferring execution on the most performant cores. To set execution preference, assign specific core type identifier to the `task_arena::constraints::core_type` field.

The example shows how to set the most performant core type as preferable for work execution:

```
std::vector<tbb::core_type_id> core_types = tbb::info::core_types();
tbb::task_arena arena(
    tbb::task_arena::constraints{}.set_core_type(core_types.back())
);

arena.execute( [] {
    /*the most performant core type is defined as preferred.*/
});
```

Limiting the maximum number of threads simultaneously scheduled to one core

The processors with Intel® Hyper-Threading Technology allow more than one thread to run on each core simultaneously. However, there might be situations when there is need to lower the number of simultaneously running threads per core. In such cases, assign the desired value to the `task_arena::constraints::max_threads_per_core` field.

The example shows how to allow only one thread to run on each core at a time:

```
tbb::task_arena no_ht_arena( tbb::task_arena::constraints{}.set_max_threads_per_core(1) );
no_ht_arena.execute( [] {
    /*parallel work*/
});
```

A more composable way to limit the number of threads executing on cores is by setting the maximal concurrency of the `tbb::task_arena`:

```
int no_ht_concurrency = tbb::info::default_concurrency(
    tbb::task_arena::constraints{}.set_max_threads_per_core(1)
);
tbb::task_arena arena( no_ht_concurrency );
arena.execute( [] {
    /*parallel work*/
});
```

Similarly to the previous example, the number of threads inside the arena is equal to the number of available cores. However, this one results in fewer overheads and better composability by imposing a less constrained execution.

Design Patterns

This section provides some common parallel programming patterns and how to implement them in Intel® oneAPI Threading Building Blocks (oneTBB).

The description of each pattern has the following format:

- **Problem** – describes the problem to be solved.
- **Context** – describes contexts in which the problem arises.
- **Forces** - considerations that drive use of the pattern.
- **Solution** - describes how to implement the pattern.

- **Example** – presents an example implementation.

Variations and examples are sometimes discussed. The code examples are intended to emphasize key points and are not full-fledged code. Examples may omit obvious const overloads of non-const methods.

Much of the nomenclature and examples are adapted from Web pages created by Eun-Gyu and Marc Snir, and the Berkeley parallel patterns wiki. See links in the **General References** section.

For brevity, some of the code examples use C++11 lambda expressions. It is straightforward, albeit sometimes tedious, to translate such lambda expressions into equivalent C++03 code.

- Agglomeration
- Elementwise
- Odd-Even Communication
- Wavefront
- Reduction
- Divide and Conquer
- GUI Thread
- Non-Preemptive Priorities
- Lazy Initialization
- Local Serializer
- Fenced Data Transfer
- Reference Counting
- General References

Agglomeration

Problem

Parallelism is so fine grained that overhead of parallel scheduling or communication swamps the useful work.

Context

Many algorithms permit parallelism at a very fine grain, on the order of a few instructions per task. But synchronization between threads usually requires orders of magnitude more cycles. For example, elementwise addition of two arrays can be done fully in parallel, but if each scalar addition is scheduled as a separate task, most of the time will be spent doing synchronization instead of useful addition.

Forces

- Individual computations can be done in parallel, but are small. For practical use of Intel® oneAPI Threading Building Blocks (oneTBB), “small” here means less than 10,000 clock cycles.
- The parallelism is for sake of performance and not required for semantic reasons.

Solution

Group the computations into blocks. Evaluate computations within a block serially.

The block size should be chosen to be large enough to amortize parallel overhead. Too large a block size may limit parallelism or load balancing because the number of blocks becomes too small to distribute work evenly across processors.

The choice of block topology is typically driven by two concerns:

- Minimizing synchronization between blocks.
- Minimizing cache traffic between blocks.

If the computations are completely independent, then the blocks will be independent too, and then only cache traffic issues must be considered.

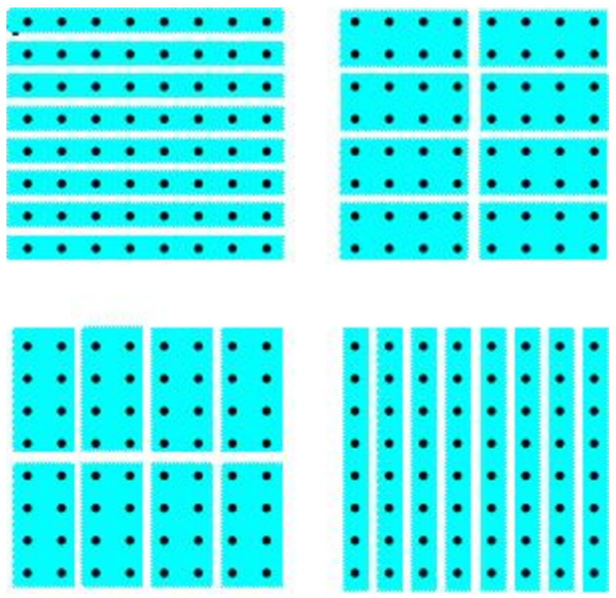
If the loop is “small”, on the order of less than 10,000 clock cycles, then it may be impractical to parallelize at all, because the optimal agglomeration might be a single block,

Examples

TBB loop templates such as `oneapi::tbb::parallel_for` that take a *range* argument support automatic agglomeration.

When agglomerating, think about cache effects. Avoid having cache lines cross between groups if possible.

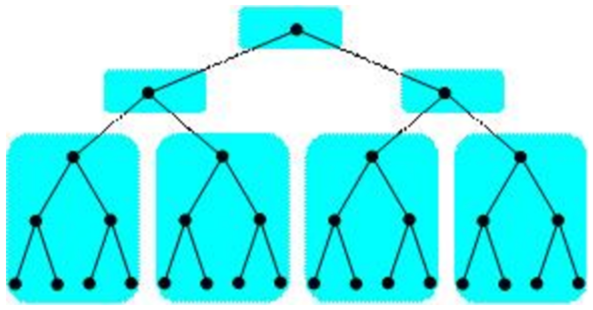
There may be boundary to interior ratio effects. For example, if the computations form a 2D grid, and communicate only with nearest neighbors, then the computation per block grows quadratically (with the block's area), but the cross-block communication grows with linearly (with the block's perimeter). The following figure shows four different ways to agglomerate an 8×8 grid. If doing such analysis, be careful to consider that information is transferred in cache line units. For a given area, the perimeter may be minimized when the block is square with respect to the underlying grid of cache lines, not square with respect to the logical grid.



Four different agglomerations of an 8×8 grid.

Also consider vectorization. Blocks that contain long contiguous subsets of data may better enable vectorization.

For recursive computations, most of the work is towards the leaves, so the solution is to treat subtrees as a groups as shown in the following figure.



Agglomeration of a recursive computation

Often such an agglomeration is achieved by recursing serially once some threshold is reached. For example, a recursive sort might solve sub-problems in parallel only if they are above a certain threshold size.

Reference

Ian Foster introduced the term "agglomeration" in his book Designing and Building Parallel Programs http://www.mcs.anl.gov/~itf/dbpp. There agglomeration is part of a four step **PCAM** design method:

1. **P**artitioning - break the program into the smallest tasks possible.
2. **C**ommunication – figure out what communication is required between tasks. When using oneTBB, communication is usually cache line transfers. Though they are automatic, understanding which ones happen between tasks helps guide the agglomeration step.
3. **A**gglomeration – combine tasks into larger tasks. His book has an extensive list of considerations that is worth reading.
4. **M**apping – map tasks onto processors. The oneTBB task scheduler does this step for you.

Elementwise

Problem

Initiate similar independent computations across items in a data set, and wait until all complete.

Context

Many serial algorithms sweep over a set of items and do an independent computation on each item. However, if some kind of summary information is collected, use the Reduction pattern instead.

Forces

No information is carried or merged between the computations.

Solution

If the number of items is known in advance, use `oneapi::tbb::parallel_for`. If not, consider using `oneapi::tbb::parallel_for_each`.

Use agglomeration if the individual computations are small relative to scheduler overheads.

If the pattern is followed by a reduction on the same data, consider doing the element-wise operation as part of the reduction, so that the combination of the two patterns is accomplished in a single sweep instead of two sweeps. Doing so may improve performance by reducing traffic through the memory hierarchy.

Example

Convolution is often used in signal processing. The convolution of a filter `c` and signal `x` is computed as:

$y_i = \sum_j c_j x_{i-j}$ Serial code for this computation might look like:

```
// Assumes c[0..clen-1] and x[1-clen..xlen-1] are defined
for( int i=0; i<xlen+clen-1; ++i ) {
   float tmp = 0;
   for( int j=0; j<clen; ++j )
       tmp += c[j]*x[i-j];
   y[i] = tmp;
}
```

For simplicity, the fragment assumes that `x` is a pointer into an array padded with zeros such that `x[k]`returns zero when `k<0` or `k≥xlen`.

The inner loop does not fit the elementwise pattern, because each iteration depends on the previous iteration. However, the outer loop fits the elementwise pattern. It is straightforward to render it using `oneapi::tbb::parallel_for` as shown:

```
oneapi::tbb::parallel_for( 0, xlen+clen-1, [=]( int i ) {
   float tmp = 0;
   for( int j=0; j<clen; ++j )
       tmp += c[j]*x[i-j];
   y[i] = tmp;
});
```

`oneapi::tbb::parallel_for` does automatic agglomeration by implicitly using `oneapi::tbb::auto_partitioner` in its underlying implementation. If there is reason to agglomerate explicitly, use the overload of `oneapi::tbb::parallel_for` that takes an explicit range argument. The following shows the example transformed to use the overload.

```
oneapi::tbb::parallel_for(
   oneapi::tbb::blocked_range<int>(0,xlen+clen-1,1000),
   [=]( oneapi::tbb::blocked_range<int> r ) {
         int end = r.end();
       for( int i=r.begin(); i!=end; ++i ) {
           float tmp = 0;
```

```
                for( int j=0; j<clen; ++j )
                    tmp += c[j]*x[i-j];
                y[i] = tmp;
            }
        }
);
```

Odd-Even Communication

Problem

Operations on data cannot be done entirely independently, but data can be partitioned into two subsets such that all operations on a subset can run in parallel.

Context

Solvers for partial differential equations can often be modified to follow this pattern. For example, for a 2D grid with only nearest-neighbor communication, it may be possible to treat the grid as a checkerboard, and alternate between updating red squares and black squares.

Another context is staggered grid (“leap frog”) Finite Difference Time Domain (FDTD solvers, which naturally fit the pattern.

Forces

- Dependencies between items form a bipartite graph.

Solution

Alternate between updating one subset and then the other subset. Apply the elementwise pattern to each subset.

References

Eun-Gyu Kim and Mark Snir, “Odd-Even Communication Group”, http://snir.cs.illinois.edu/patterns/oddeven.pdf

Wavefront

Problem

Perform computations on items in a data set, where the computation on an item uses results from computations on predecessor items.

Context

The dependences between computations form an acyclic graph.

Forces

- Dependence constraints between items form an acyclic graph.
- The number of immediate predecessors in the graph is known in advance, or can be determined some time before the last predecessor completes.

Solution

The solution is a parallel variant of topological sorting, using `oneapi::tbb::parallel_for_each` to process items. Associate an atomic counter with each item. Initialize each counter to the number of predecessors. Invoke `oneapi::tbb::parallel_for_each` to process the items that have no predessors (have counts of zero). After an item is processed, decrement the counters of its successors. If a successor’s counter reaches zero, add that successor to the `oneapi::tbb::parallel_for_each` via a “feeder”.

If the number of predecessors for an item cannot be determined in advance, treat the information “know number of predecessors” as an additional predecessor. When the number of predecessors becomes known, treat this conceptual predecessor as completed.

If the overhead of counting individual items is excessive, aggregate items into blocks, and do the wavefront over the blocks.

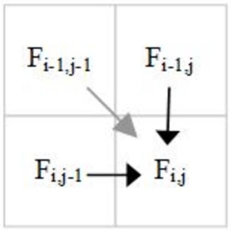
Example

Below is a serial kernel for the longest common subsequence algorithm. The parameters are strings `x` and `y` with respective lengths `xlen` and `ylen`.

```
int F[MAX_LEN+1][MAX_LEN+1];


void SerialLCS( const char* x, size_t xlen, const char* y, size_t ylen )
{
   for( size_t i=1; i<=xlen; ++i )
       for( size_t j=1; j<=ylen; ++j )
           F[i][j] = x[i-1]==y[j-1] ? F[i-1][j-1]+1:
                                      max(F[i][j-1],F[i-1][j]);
}
```

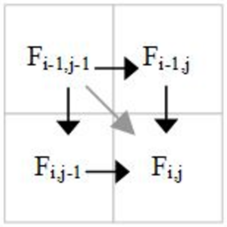
The kernel sets `F[i][j]` to the length of the longest common subsequence shared by `x[0..i-1]` and `y[0..j-1]`. It assumes that F[0][0..ylen] and `F[0..xlen][0]` have already been initialized to zero.

The following figure shows the data dependences for calculating `F[i][j]`.



Data dependences for longest common substring calculation.

The following figure shows the gray diagonal dependence is the transitive closure of other dependencies. Thus for parallelization purposes it is a redundant dependence that can be ignored.



Diagonal dependence is redundant.

It is generally good to remove redundant dependences from consideration, because the atomic counting incurs a cost for each dependence considered.

Another consideration is grain size. Scheduling each `F[i][j]` element calculation separately is prohibitively expensive. A good solution is to aggregate the elements into contiguous blocks, and process the contents of a block serially. The blocks have the same dependence pattern, but at a block scale. Hence scheduling overheads can be amortized over blocks.

The parallel code follows. Each block consists of `N×N` elements. Each block has an associated atomic counter. Array `Count` organizes these counters for easy lookup. The code initializes the counters and then rolls a wavefront using `parallel_for_each`, starting with the block at the origin since it has no predecessors.

```
const int N = 64;
std::atomic<char> Count[MAX_LEN/N+1][MAX_LEN/N+1];


void ParallelLCS( const char* x, size_t xlen, const char* y, size_t ylen ) {
   // Initialize predecessor counts for blocks.
   size_t m = (xlen+N-1)/N;
   size_t n = (ylen+N-1)/N;
   for( int i=0; i<m; ++i )
       for( int j=0; j<n; ++j )
           Count[i][j] = (i>0)+(j>0);
   // Roll the wavefront from the origin.
   typedef pair<size_t,size_t> block;
   block origin(0,0);
   oneapi::tbb::parallel_for_each( &origin, &origin+1,
       [=]( const block& b, oneapi::tbb::feeder<block>&feeder ) {
           // Extract bounds on block
           size_t bi = b.first;
           size_t bj = b.second;
           size_t xl = N*bi+1;
           size_t xu = min(xl+N,xlen+1);
           size_t yl = N*bj+1;
           size_t yu = min(yl+N,ylen+1);
           // Process the block
           for( size_t i=xl; i<xu; ++i )
               for( size_t j=yl; j<yu; ++j )
                   F[i][j] = x[i-1]==y[j-1] ? F[i-1][j-1]+1:
                                              max(F[i][j-1],F[i-1][j]);
           // Account for successors
           if( bj+1<n && --Count[bi][bj+1]==0 )
               feeder.add( block(bi,bj+1) );
           if( bi+1<m && --Count[bi+1][bj]==0 )
               feeder.add( block(bi+1,bj) );       }
   );
}
```

References

Eun-Gyu Kim and Mark Snir, “Wavefront Pattern”, http://snir.cs.illinois.edu/patterns/wavefront.pdf

Reduction

Problem

Perform an associative reduction operation across a data set.

Context

Many serial algorithms sweep over a set of items to collect summary information.

Forces

The summary can be expressed as an associative operation over the data set, or at least is close enough to associative that reassociation does not matter.

Solution

Two solutions exist in Intel® oneAPI Threading Building Blocks (oneTBB). The choice on which to use depends upon several considerations:

- Is the operation commutative as well as associative?
- Are instances of the reduction type expensive to construct and destroy. For example, a floating point number is inexpensive to construct. A sparse floating-point matrix might be very expensive to construct.

Use `oneapi::tbb::parallel_reduce` when the objects are inexpensive to construct. It works even if the reduction operation is not commutative.

Use `oneapi::tbb::parallel_for` and `oneapi::tbb::combinable` if the reduction operation is commutative and instances of the type are expensive.

If the operation is not precisely associative but a precisely deterministic result is required, use recursive reduction and parallelize it using `oneapi::tbb::parallel_invoke`.

Examples

The examples presented here illustrate the various solutions and some tradeoffs.

The first example uses `oneapi::tbb::parallel_reduce` to do a + reduction over sequence of type `T`. The sequence is defined by a half-open interval [first,last).

```
T AssociativeReduce( const T* first, const T* last, T identity ) {
   return oneapi::tbb::parallel_reduce(
       // Index range for reduction
       oneapi::tbb::blocked_range<const T*>(first,last),
       // Identity element
       identity,
       // Reduce a subrange and partial sum
       [&]( oneapi::tbb::blocked_range<const T*> r, T partial_sum )->float {
           return std::accumulate( r.begin(), r.end(), partial_sum );
       },
       // Reduce two partial sums
       std::plus<T>()
   );
}
```

The third and fourth arguments to this form of `parallel_reduce` are a built in form of the agglomeration pattern. If there is an elementwise action to be performed before the reduction, incorporating it into the third argument (reduction of a subrange) may improve performance because of better locality of reference. Note that the block size for agglomeration is not explicitly specified; `parallel_reduce` defines blocks automatically with the help of implicitly used `oneapi::tbb::auto_partitioner`.

The second example assumes the + is commutative on `T`. It is a good solution when `T` objects are expensive to construct.

```
T CombineReduce( const T* first, const T* last, T identity ) {
   oneapi::tbb::combinable<T> sum(identity);
   oneapi::tbb::parallel_for(
       oneapi::tbb::blocked_range<const T*>(first,last),
       [&]( oneapi::tbb::blocked_range<const T*> r ) {
           sum.local() += std::accumulate(r.begin(), r.end(), identity);
       }
   );
   return sum.combine( []( const T& x, const T& y ) {return x+y;} );
}
```

Sometimes it is desirable to destructively use the partial results to generate the final result. For example, if the partial results are lists, they can be spliced together to form the final result. In that case use class `oneapi::tbb::enumerable_thread_specific` instead of `combinable`. The `ParallelFindCollisions` example in Divide and Conquer demonstrates the technique.

Floating-point addition and multiplication are almost associative. Reassociation can cause changes because of rounding effects. The techniques shown so far reassociate terms non-deterministically. Fully deterministic parallel reduction for a not quite associative operation requires using deterministic reassociation. The code below demonstrates this in the form of a template that does a + reduction over a sequence of values of type `T`.

```
template<typename T>
T RepeatableReduce( const T* first, const T* last, T identity ) {
   if( last-first<=1000 ) {
       // Use serial reduction
       return std::accumulate( first, last, identity );
   } else {
       // Do parallel divide-and-conquer reduction
       const T* mid = first+(last-first)/2;
       T left, right;
       oneapi::tbb::parallel_invoke(
           [&]{left=RepeatableReduce(first,mid,identity);},
           [&]{right=RepeatableReduce(mid,last,identity);}
       );
       return left+right;
   }
}
```

The outer if-else is an instance of the agglomeration pattern for recursive computations. The reduction graph, though not a strict binary tree, is fully deterministic. Thus the result will always be the same for a given input sequence, assuming all threads do identical floating-point rounding.

`oneapi::tbb::parallel_deterministic_reduce` is a simpler and more efficient way to get reproducible non-associative reduction. It is very similar to `oneapi::tbb::parallel_reduce` but, unlike the latter, builds a deterministic reduction graph. With it, the `RepeatableReduce` sample can be almost identical to `AssociativeReduce`:

```
template<typename T>
T RepeatableReduce( const T* first, const T* last, T identity ) {
   return oneapi::tbb::parallel_deterministic_reduce(
       // Index range for reduction
       oneapi::tbb::blocked_range<const T*>(first,last,1000),
       // Identity element
       identity,
       // Reduce a subrange and partial sum
       [&]( oneapi::tbb::blocked_range<const T*> r, T partial_sum )->float {
           return std::accumulate( r.begin(), r.end(), partial_sum );
       },
       // Reduce two partial sums
       std::plus<T>()
   );
}
```

Besides the function name change, note the grain size of 1000 specified for `oneapi::tbb::blocked_range`. It defines the desired block size for agglomeration; automatic block size selection is not used due to non-determinism.

The final example shows how a problem that typically is not viewed as a reduction can be parallelized by viewing it as a reduction. The problem is retrieving floating-point exception flags for a computation across a data set. The serial code might look something like:

```
feclearexcept(FE_ALL_EXCEPT);
for( int i=0; i<N; ++i )
    C[i]=A[i]*B[i];
int flags = fetestexcept(FE_ALL_EXCEPT);
if (flags & FE_DIVBYZERO) ...;
if (flags & FE_OVERFLOW) ...;
...
```

The code can be parallelized by computing chunks of the loop separately, and merging floating-point flags from each chunk. To do this with `tbb:parallel_reduce`, first define a “body” type, as shown below.

```
struct ComputeChunk {
   int flags;          // Holds floating-point exceptions seen so far.
   void reset_fpe() {
       flags=0;
       feclearexcept(FE_ALL_EXCEPT);
   }
   ComputeChunk () {
       reset_fpe();
   }
   // "Splitting constructor"called by parallel_reduce when splitting a range into subranges.
   ComputeChunk ( const ComputeChunk&, oneapi::tbb::split ) {
       reset_fpe();
   }
   // Operates on a chunk and collects floating-point exception state into flags member.
   void operator()( oneapi::tbb::blocked_range<int> r ) {
       int end=r.end();
       for( int i=r.begin(); i!=end; ++i )
           C[i] = A[i]/B[i];
       // It is critical to do |= here, not =, because otherwise we
       // might lose earlier exceptions from the same thread.
       flags |= fetestexcept(FE_ALL_EXCEPT);
   }
   // Called by parallel_reduce when joining results from two subranges.
   void join( Body& other ) {
       flags |= other.flags;
   }
};
```

Then invoke it as follows:

```
// Construction of cc implicitly resets FP exception state.
   ComputeChunk cc;
   oneapi::tbb::parallel_reduce( oneapi::tbb::blocked_range<int>(0,N), cc );
   if (cc.flags & FE_DIVBYZERO) ...;
   if (cc.flags & FE_OVERFLOW) ...;
   ...
```

Divide and Conquer

Problem

Parallelize a divide and conquer algorithm.

Context

Divide and conquer is widely used in serial algorithms. Common examples are quicksort and mergesort.

Forces

- Problem can be transformed into subproblems that can be solved independently.
- Splitting problem or merging solutions is relatively cheap compared to cost of solving the subproblems.

Solution

There are several ways to implement divide and conquer in Intel® oneAPI Threading Building Blocks (oneTBB). The best choice depends upon circumstances.

- If division always yields the same number of subproblems, use recursion and `oneapi::tbb::parallel_invoke`.
- If the number of subproblems varies, use recursion and `oneapi::tbb::task_group`.

Example

Quicksort is a classic divide-and-conquer algorithm. It divides a sorting problem into two subsorts. A simple serial version looks like [1].

```
void SerialQuicksort( T* begin, T* end ) {
   if( end-begin>1  ) {
       using namespace std;
       T* mid = partition( begin+1, end, bind2nd(less<T>(),*begin) );
       swap( *begin, mid[-1] );
       SerialQuicksort( begin, mid-1 );
```

```
        SerialQuicksort( mid, end );
    }
}
```

The number of subsorts is fixed at two, so `oneapi::tbb::parallel_invoke` provides a simple way to parallelize it. The parallel code is shown below:

```
void ParallelQuicksort( T* begin, T* end ) {
    if( end-begin>1 ) {
        using namespace std;
        T* mid = partition( begin+1, end, bind2nd(less<T>(),*begin) );
        swap( *begin, mid[-1] );
        oneapi::tbb::parallel_invoke( [=]{ParallelQuicksort( begin, mid-1 );},
                              [=]{ParallelQuicksort( mid, end );} );
    }
}
```

Eventually the subsorts become small enough that serial execution is more efficient. The following variation, does sorts of less than 500 elements using the earlier serial code.

```
void ParallelQuicksort( T* begin, T* end ) {
    if( end-begin>=500 ) {
        using namespace std;
        T* mid = partition( begin+1, end, bind2nd(less<T>(),*begin) );
        swap( *begin, mid[-1] );
        oneapi::tbb::parallel_invoke( [=]{ParallelQuicksort( begin, mid-1 );},
                              [=]{ParallelQuicksort( mid, end );} );
    } else {
        SerialQuicksort( begin, end );
    }
}
```

The change is an instance of the Agglomeration pattern.

The next example considers a problem where there are a variable number of subproblems. The problem involves a tree-like description of a mechanical assembly. There are two kinds of nodes:

- Leaf nodes represent individual parts.
- Internal nodes represent groups of parts.

The problem is to find all nodes that collide with a target node. The following code shows a serial solution that walks the tree. It records in `Hits` any nodes that collide with `Target`.

```
std::list<Node*> Hits;
Node* Target;


void SerialFindCollisions( Node& x ) {
    if( x.is_leaf() ) {
        if( x.collides_with( *Target ) )
            Hits.push_back(&x);
    } else {
        for( Node::const_iterator y=x.begin();y!=x.end(); ++y )
            SerialFindCollisions(*y);
    }
}
```

A parallel version is shown below.

```
typedef oneapi::tbb::enumerable_thread_specific<std::list<Node*> > LocalList;
LocalList LocalHits;
Node* Target;    // Target node
```

```
void ParallelWalk( Node& x ) {
   if( x.is_leaf() ) {
       if( x.collides_with( *Target ) )
           LocalHits.local().push_back(&x);
   } else {
       // Recurse on each child y of x in parallel
       oneapi::tbb::task_group g;
       for( Node::const_iterator y=x.begin(); y!=x.end(); ++y )
           g.run( [=]{ParallelWalk(*y);} );
       // Wait for recursive calls to complete
       g.wait();
   }
}


void ParallelFindCollisions( Node& x ) {
   ParallelWalk(x);
   for(LocalList::iterator i=LocalHits.begin();i!=LocalHits.end(); ++i)
       Hits.splice( Hits.end(), *i );
}
```

The recursive walk is parallelized using class `task_group` to do recursive calls in parallel.

There is another significant change because of the parallelism that is introduced. Because it would be unsafe to update `Hits` concurrently, the parallel walk uses variable `LocalHits` to accumulate results. Because it is of type `enumerable_thread_specific`, each thread accumulates its own private result. The results are spliced together into Hits after the walk completes.

The results will *not* be in the same order as the original serial code.

If parallel overhead is high, use the agglomeration pattern. For example, use the serial walk for subtrees under a certain threshold.

[1] Production quality quicksort implementations typically use more sophisticated pivot selection, explicit stacks instead of recursion, and some other sorting algorithm for small subsorts. The simple algorithm is used here to focus on exposition of the parallel pattern.

GUI Thread

Problem

A user interface thread must remain responsive to user requests, and must not get bogged down in long computations.

Context

Graphical user interfaces often have a dedicated thread (“GUI thread”) for servicing user interactions. The thread must remain responsive to user requests even while the application has long computations running. For example, the user might want to press a “cancel” button to stop the long running computation. If the GUI thread takes part in the long running computation, it will not be able to respond to user requests.

Forces

- The GUI thread services an event loop.
- The GUI thread needs to offload work onto other threads without waiting for the work to complete.
- The GUI thread must be responsive to the event loop and not become dedicated to doing the offloaded work.

Related

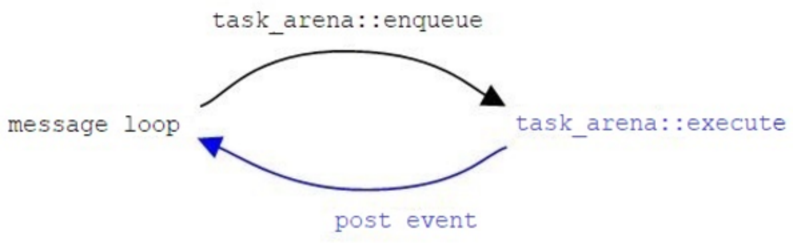
- Non-Preemptive Priorities

- Local Serializer

Solution

The GUI thread offloads the work by firing off a task to do it using method `task_arena::enqueue` of a `task_arena` instance. When finished, the task posts an event to the GUI thread to indicate that the work is done. The semantics of `enqueue` cause the task to eventually run on a worker thread distinct from the calling thread.

The following figure sketches the communication paths. Items in black are executed by the GUI thread; items in blue are executed by another thread.



Example

The example is for the Microsoft Windows* operating systems, though similar principles apply to any GUI using an event loop idiom. For each event, the GUI thread calls a user-defined function `WndProc` to process an event.

```
// Event posted from enqueued task when it finishes its work.
const UINT WM_POP_FOO = WM_USER+0;


// Queue for transmitting results from enqueued task to GUI thread.
oneapi::tbb::concurrent_queue<Foo>ResultQueue;


// GUI thread's private copy of most recently computed result.
Foo CurrentResult;


LRESULT CALLBACK WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam) {
   switch(msg) {
       case WM_COMMAND:
           switch (LOWORD(wParam)) {
               case IDM_LONGRUNNINGWORK:
                   // User requested a long computation. Delegate it to another thread.
                   LaunchLongRunningWork(hWnd);
                   break;
               case IDM_EXIT:
                   DestroyWindow(hWnd);
                   break;
               default:
                   return DefWindowProc(hWnd, msg, wParam, lParam);
           }
           break;
       case WM_POP_FOO:
           // There is another result in ResultQueue for me to grab.
           ResultQueue.try_pop(CurrentResult);
           // Update the window with the latest result.
           RedrawWindow( hWnd, NULL, NULL, RDW_ERASE|RDW_INVALIDATE );
           break;
```

```
        case WM_PAINT:
            Repaint the window using CurrentResult
            break;
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        default:
            return DefWindowProc( hWnd, msg, wParam, lParam );
    }
    return 0;
}
```

The GUI thread processes long computations as follows:

1. The GUI thread calls `LongRunningWork`, which hands off the work to a worker thread and immediately returns.
2. The GUI thread continues servicing the event loop. If it has to repaint the window, it uses the value of`CurrentResult`, which is the most recent `Foo` that it has seen.

When a worker finishes the long computation, it pushes the result into ResultQueue, and sends a message WM_POP_FOO to the GUI thread.

1. The GUI thread services a `WM_POP_FOO` message by popping an item from ResultQueue into CurrentResult. The `try_pop` always succeeds because there is exactly one `WM_POP_FOO` message for each item in `ResultQueue`.

Routine `LaunchLongRunningWork` creates a function task and launches it using method `task_arena::enqueue`.

```
class LongTask {
    HWND hWnd;
    void operator()() {
        Do long computation
        Foo x = result of long computation
        ResultQueue.push( x );
        // Notify GUI thread that result is available.
        PostMessage(hWnd,WM_POP_FOO,0,0);
    }
public:
    LongTask( HWND hWnd_ ) : hWnd(hWnd_) {}
};

void LaunchLongRunningWork( HWND hWnd ) {
    oneapi::tbb::task_arena a;
    a.enqueue(LongTask(hWnd));
}
```

It is essential to use method `task_arena::enqueue` here. Even though, an explicit `task_arena` instance is created, the method `enqueue` ensures that the function task eventually executes when resources permit, even if no thread explicitly waits on the task. In contrast, `oneapi::tbb::task_group::run` may postpone execution of the function task until it is explicitly waited upon with the `oneapi::tbb::task_group::wait`.

The example uses a `concurrent_queue` for workers to communicate results back to the GUI thread. Since only the most recent result matters in the example, and alternative would be to use a shared variable protected by a mutex. However, doing so would block the worker while the GUI thread was holding a lock on the mutex, and vice versa. Using `concurrent_queue` provides a simple robust solution.

If two long computations are in flight, there is a chance that the first computation completes after the second one. If displaying the result of the most recently requested computation is important, then associate a request serial number with the computation. The GUI thread can pop from `ResultQueue` into a temporary variable, check the serial number, and update `CurrentResult` only if doing so advances the serial number.

Non-Preemptive Priorities

Problem

Choose the next work item to do, based on priorities.

Context

The scheduler in Intel® oneAPI Threading Building Blocks (oneTBB) chooses tasks using rules based on scalability concerns. The rules are based on the order in which tasks were spawned or enqueued, and are oblivious to the contents of tasks. However, sometimes it is best to choose work based on some kind of priority relationship.

Forces

- Given multiple work items, there is a rule for which item should be done next that is *not* the default oneTBB rule.
- Preemptive priorities are not necessary. If a higher priority item appears, it is not necessary to immediately stop lower priority items in flight. If preemptive priorities are necessary, then non-preemptive tasking is inappropriate. Use threads instead.

Solution

Put the work in a shared work pile. Decouple tasks from specific work, so that task execution chooses the actual piece of work to be selected from the pile.

Example

The following example implements three priority levels. The user interface for it and top-level implementation follow:

```
enum Priority {
   P_High,
   P_Medium,
   P_Low
};


template<typename Func>
void EnqueueWork( Priority p, Func f ) {
   WorkItem* item = new ConcreteWorkItem<Func>( p, f );
   ReadyPile.add(item);
}
```

The caller provides a priority `p` and a functor `f` to routine `EnqueueWork`. The functor may be the result of a lambda expression. `EnqueueWork` packages `f` as a `WorkItem` and adds it to global object `ReadyPile`.

Class `WorkItem` provides a uniform interface for running functors of unknown type:

```
// Abstract base class for a prioritized piece of work.
class WorkItem {
public:
   WorkItem( Priority p ) : priority(p) {}
   // Derived class defines the actual work.
   virtual void run() = 0;
   const Priority priority;
};


template<typename Func>
class ConcreteWorkItem: public WorkItem {
   Func f;
   /*override*/ void run() {
       f();
```

```
        delete this;
    }
public:
    ConcreteWorkItem( Priority p, const Func& f_ ) :
        WorkItem(p), f(f_)
    {}
};
```

Class `ReadyPile` contains the core pattern. It maintains a collection of work and fires off tasks through the `oneapi::tbb::task_group::run` interface and then choose a work from the collection:

```
class ReadyPileType {
    // One queue for each priority level
    oneapi::tbb::concurrent_queue<WorkItem*> level[P_Low+1];
    oneapi::tbb::task_group tg;
public:
    void add( WorkItem* item ) {
        level[item->priority].push(item);
        tg.run(RunWorkItem());
    }
    void runNextWorkItem() {
        // Scan queues in priority order for an item.
        WorkItem* item=NULL;
        for( int i=P_High; i<=P_Low; ++i )
            if( level[i].try_pop(item) )
                break;
        assert(item);
        item->run();
    }
};


ReadyPileType ReadyPile;
```

The task added by `add(item)` does *not* necessarily execute that item. The task itself executes `runNextWorkItem()`, which may find a higher priority item. There is one task for each item, but the mapping resolves when the task actually executes, not when it is created.

Here are the details of class `RunWorkItem`:

```
class RunWorkItem {
    void operator()() {
        ReadyPile.runNextWorkItem();
    };
};
```

`RunWorkItem` objects are fungible. They enable the oneTBB scheduler to choose when to do a work item, not which work item to do.

Other priority schemes can be implemented by changing the internals for `ReadyPileType`. A priority queue could be used to implement very fine grained priorities.

The scalability of the pattern is limited by the scalability of `ReadyPileType`. Ideally scalable concurrent containers should be used for it.

Lazy Initialization

Problem

Delay the creation of an object, potentially expensive, until it is accessed. In parallel programming, initialization must also be guarded against race conditions.

Context

The cost of operations that take place during the initialization of the object may be considerably high. In that case, the object should be initialized only when needed. Lazy initialization is the common tactic that allows implementing such an approach.

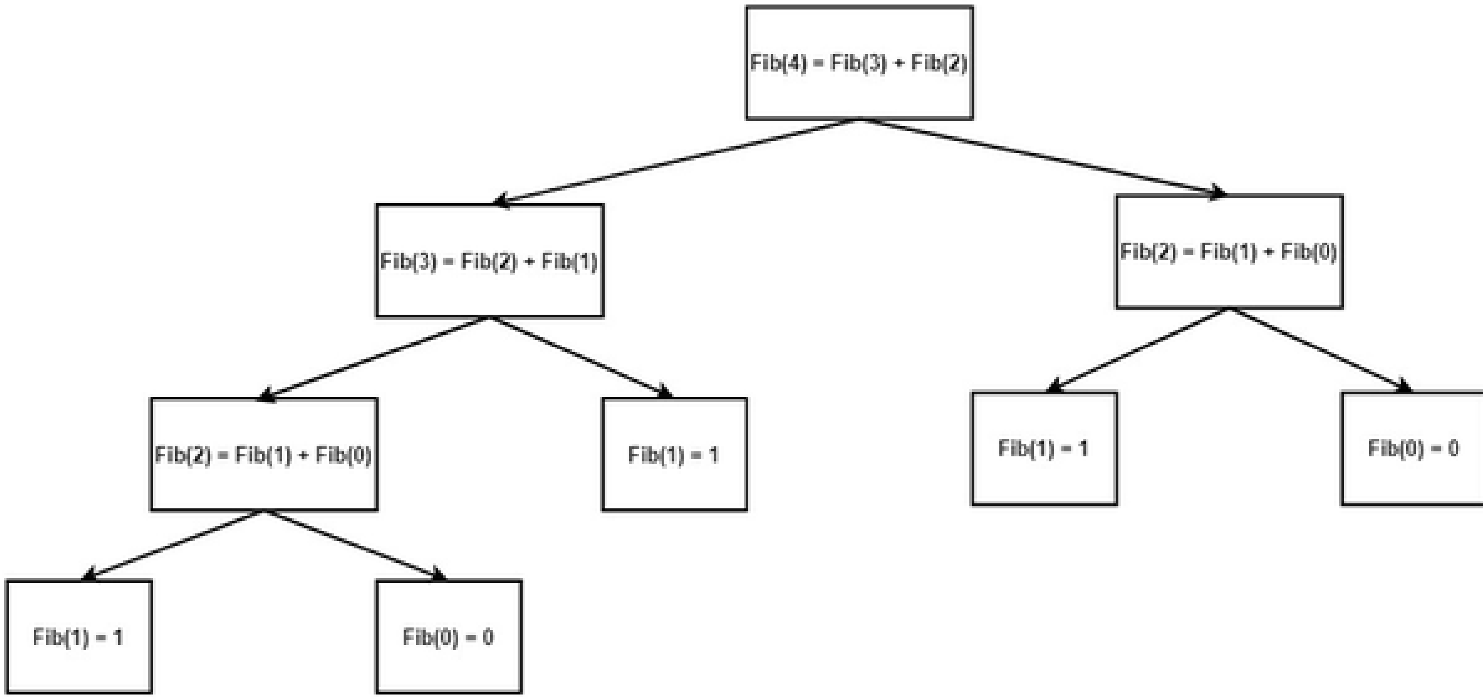
Solution

Using `oneapi::tbb::collaborative_call_once` with `oneapi::tbb::collaborative_once_flag` helps to implement thread-safe lazy initialization for a user object.

In addition, `collaborative_call_once` allows other thread blocked on the same `collaborative_once_flag` to join other oneTBB parallel constructions called within the initializing function.

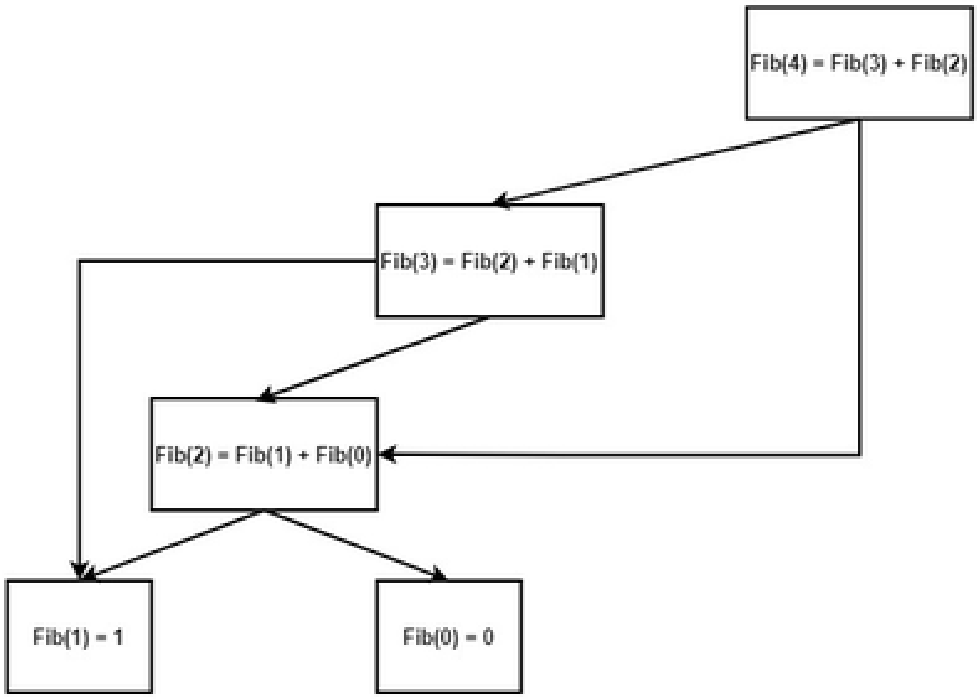
Example

This example illustrates the implementation of lazy initialization for the calculation of the Fibonacci numbers. Here is a graphical representation of the Fibonacci recursion tree for N=4.



As seen in the diagram, some elements are recalculated more than once. These operations are redundant, so the "lazy initialized" Fibonacci numbers are relevant here.

An implementation without the use of lazy initialization would have *O(2^N)* time complexity due to the full recursion tree traversal and recalculation of values. Since all the nodes are traversed once, the tree becomes a list, making the time complexity *O(N)*.



Here you can see the code for the implementation. Already calculated values are stored in a buffer paired with `collaborative_once_flag` and will not be recalculated when `collaborative_call_once` is invoked when initialization has already been done.

```
using FibBuffer = std::vector<std::pair<oneapi::tbb::collaborative_once_flag, std::uint64_t>>;

std::uint64_t LazyFibHelper(int n, FibBuffer& buffer) {
   // Base case
   if (n <= 1) {
      return n;
   }
   // Calculate nth value only once and store it in the buffer.
   // Other threads won't be blocked on already taken collaborative_once_flag
   // but join parallelism inside functor
   oneapi::tbb::collaborative_call_once(buffer[n].first, [&]() {
      std::uint64_t a, b;
      oneapi::tbb::parallel_invoke([&] { a = LazyFibHelper(n - 2, buffer); },
                                   [&] { b = LazyFibHelper(n - 1, buffer); });
      buffer[n].second = a + b;
   });

   return buffer[n].second;
}

std::uint64_t Fib(int n) {
   FibBuffer buffer(n+1);
   return LazyFibHelper(n, buffer);
}
```

Local Serializer

Context

Consider an interactive program. To maximize concurrency and responsiveness, operations requested by the user can be implemented as tasks. The order of operations can be important. For example, suppose the program presents editable text to the user. There might be operations to select text and delete selected text. Reversing the order of "select" and "delete" operations on the same buffer would be bad. However, commuting operations on different buffers might be okay. Hence the goal is to establish serial ordering of tasks associated with a given object, but not constrain ordering of tasks between different objects.

Forces

- Operations associated with a certain object must be performed in serial order.
- Serializing with a lock would be wasteful because threads would be waiting at the lock when they could be doing useful work elsewhere.

Solution

Sequence the work items using a FIFO (first-in first-out structure). Always keep an item in flight if possible. If no item is in flight when a work item appears, put the item in flight. Otherwise, push the item onto the FIFO. When the current item in flight completes, pop another item from the FIFO and put it in flight.

The logic can be implemented without mutexes, by using `concurrent_queue` for the FIFO and `atomic<int>` to count the number of items waiting and in flight. The example explains the accounting in detail.

Example

The following example builds on the Non-Preemptive Priorities example to implement local serialization in addition to priorities. It implements three priority levels and local serializers. The user interface for it follows:

```
enum Priority {
   P_High,
   P_Medium,
   P_Low
};


template<typename Func>
void EnqueueWork( Priority p, Func f, Serializer* s=NULL );
```

Template function `EnqueueWork` causes functor `f` to run when the three constraints in the following table are met.

Constraint	Resolved by class...
Any prior work for the `Serializer` has completed.	`Serializer`
A thread is available.	`RunWorkItem`
No higher priority work is ready to run.	`ReadyPileType`

Constraints on a given functor are resolved from top to bottom in the table. The first constraint does not exist when s is NULL. The implementation of `EnqueueWork` packages the functor in a `SerializedWorkItem` and routes it to the class that enforces the first relevant constraint between pieces of work.

```
template<typename Func>
void EnqueueWork( Priority p, Func f, Serializer* s=NULL ) {
   WorkItem* item = new SerializedWorkItem<Func>( p, f, s );
   if( s )
       s->add(item);
   else
       ReadyPile.add(item);
}
```

A `SerializedWorkItem` is derived from a `WorkItem`, which serves as a way to pass around a prioritized piece of work without knowing further details of the work.

```
// Abstract base class for a prioritized piece of work.
class WorkItem {
public:
   WorkItem( Priority p ) : priority(p) {}
   // Derived class defines the actual work.
   virtual void run() = 0;
   const Priority priority;
};


template<typename Func>
class SerializedWorkItem: public WorkItem {
   Serializer* serializer;
   Func f;
   /*override*/ void run() {
       f();
       Serializer* s = serializer;
       // Destroy f before running Serializer's next functor.
       delete this;
       if( s )
           s->noteCompletion();
   }
public:
   SerializedWorkItem( Priority p, const Func& f_, Serializer* s ) :
       WorkItem(p), serializer(s), f(f_)
   {}
};
```

Base class `WorkItem` is the same as class WorkItem in the example for Non-Preemptive Priorities. The notion of serial constraints is completely hidden from the base class, thus permitting the framework to extend other kinds of constraints or lack of constraints. Class `SerializedWorkItem` is essentially `ConcreteWorkItem` from the example for Non-Preemptive Priorities, extended with a `Serializer` aspect.

Virtual method `run()` is invoked when it becomes time to run the functor. It performs three steps:

1. Run the functor.
2. Destroy the functor.
3. Notify the `Serializer` that the functor completed, and thus unconstraining the next waiting functor.

Step 3 is the difference from the operation of ConcreteWorkItem::run. Step 2 could be done after step 3 in some contexts to increase concurrency slightly. However, the presented order is recommended because if step 2 takes non-trivial time, it likely has side effects that should complete before the next functor runs.

Class `Serializer` implements the core of the Local Serializer pattern:

```
class Serializer {
   oneapi::tbb::concurrent_queue<WorkItem*> queue;
   std::atomic<int> count;         // Count of queued items and in-flight item
   void moveOneItemToReadyPile() { // Transfer item from queue to ReadyPile
       WorkItem* item;
       queue.try_pop(item);
       ReadyPile.add(item);
   }
public:
   void add( WorkItem* item ) {
       queue.push(item);
       if( ++count==1 )
           moveOneItemToReadyPile();
   }
   void noteCompletion() {        // Called when WorkItem completes.
       if( --count!=0 )
           moveOneItemToReadyPile();
   }
};
```

The class maintains two members:

- A queue of WorkItem waiting for prior work to complete.
- A count of queued or in-flight work.

Mutexes are avoided by using `concurrent_queue<WorkItem*>` and `atomic<int>` along with careful ordering of operations. The transitions of count are the key understanding how class `Serializer` works.

- If method `add` increments `count` from 0 to 1, this indicates that no other work is in flight and thus the work should be moved to the `ReadyPile`.
- If method `noteCompletion` decrements count and it is *not* from 1 to 0, then the queue is non-empty and another item in the queue should be moved to `ReadyPile`.

Class `ReadyPile` is explained in the example for Non-Preemptive Priorities.

If priorities are not necessary, there are two variations on method `moveOneItemToReadyPile`, with different implications.

- Method `moveOneItemToReadyPile` could directly invoke`item->run()`. This approach has relatively low overhead and high thread locality for a given `Serializer`. But it is unfair. If the `Serializer` has a continual stream of tasks, the thread operating on it will keep servicing those tasks to the exclusion of others.
- Method `moveOneItemToReadyPile` could invoke `task::enqueue` to enqueue a task that invokes `item->run()`. Doing so introduces higher overhead and less locality than the first approach, but avoids starvation.

The conflict between fairness and maximum locality is fundamental. The best resolution depends upon circumstance.

The pattern generalizes to constraints on work items more general than those maintained by class Serializer. A generalized `Serializer::add` determines if a work item is unconstrained, and if so, runs it immediately. A generalized `Serializer::noteCompletion` runs all previously constrained items that have become unconstrained by the completion of the current work item. The term "run" means to run work immediately, or if there are more constraints, forwarding the work to the next constraint resolver.

Fenced Data Transfer

Problem

Write a message to memory and have another processor read it on hardware that does not have a sequentially consistent memory model.

Context

The problem normally arises only when unsynchronized threads concurrently act on a memory location, or are using reads and writes to create synchronization. High level synchronization constructs normally include mechanisms that prevent unwanted reordering.

Modern hardware and compilers can reorder memory operations in a way that preserves the order of a thread's operation from its viewpoint, but not as observed by other threads. A serial common idiom is to write a message and mark it as ready to ready as shown in the following code:

```
bool Ready;
std::string Message;


void Send( const std::string& src ) {. // Executed by thread 1
   Message=src;
   Ready = true;
}


bool Receive( std::string& dst ) {    // Executed by thread 2
   bool result = Ready;
   if( result ) dst=Message;
   return result;             // Return true if message was received.
}
```

Two key assumptions of the code are:

1. `Ready` does not become true until `Message` is written.
2. `Message` is not read until `Ready` becomes true.

These assumptions are trivially true on uniprocessor hardware. However, they may break on multiprocessor hardware. Reordering by the hardware or compiler can cause the sender's writes to appear out of order to the receiver (thus breaking condition a) or the receiver's reads to appear out of order (thus breaking condition b).

Forces

- Creating synchronization via raw reads and writes.

Solution

Change the flag from `bool` to `std::atomic<bool>` for the flag that indicates when the message is ready. Here is the previous example with modifications.

```
std::atomic<bool> Ready;
std::string Message;


void Send( const std::string& src ) {. // Executed by thread 1
   Message=src;
   Ready.store(true, std::memory_order_release);
}


bool Receive( std::string& dst ) {    // Executed by thread 2
   bool result = Ready.load(std::memory_order_acquire);
   if( result ) dst=Message;
   return result;             // Return true if message was received.
}
```

A write to a `std::atomic` value has *release* semantics, which means that all of its prior writes will be seen before the releasing write. A read from `std::atomic` value has *acquire* semantics, which means that all of its subsequent reads will happen after the acquiring read. The implementation of `std::atomic` ensures that both the compiler and the hardware observe these ordering constraints.

Variations

Higher level synchronization constructs normally include the necessary *acquire* and *release* fences. For example, mutexes are normally implemented such that acquisition of a lock has *acquire* semantics and release of a lock has *release* semantics. Thus a thread that acquires a lock on a mutex always sees any memory writes done by another thread before it released a lock on that mutex.

Non Solutions

Mistaken solutions are so often proposed that it is worth understanding why they are wrong.

One common mistake is to assume that declaring the flag with the `volatile` keyword solves the problem. Though the `volatile` keyword forces a write to happen immediately, it generally has no effect on the visible ordering of that write with respect to other memory operations.

Another mistake is to assume that conditionally executed code cannot happen before the condition is tested. However, the compiler or hardware may speculatively hoist the conditional code above the condition.

Similarly, it is a mistake to assume that a processor cannot read the target of a pointer before reading the pointer. A modern processor does not read individual values from main memory. It reads cache lines. The target of a pointer may be in a cache line that has already been read before the pointer was read, thus giving the appearance that the processor presciently read the pointer target.

Reference Counting

Problem

Destroy an object when it will no longer be used.

Context

Often it is desirable to destroy an object when it is known that it will not be used in the future. Reference counting is a common serial solution that extends to parallel programming if done carefully.

Forces

- If there are cycles of references, basic reference counting is insufficient unless the cycle is explicitly broken.
- Atomic counting is relatively expensive in hardware.

Solution

Thread-safe reference counting is like serial reference counting, except that the increment/decrement is done atomically, and the decrement and test "count is zero?" must act as a single atomic operation. The following example uses `std::atomic<int>` to achieve this.

```
template<typename T>
class counted {
   std::atomic<int> my_count;
   T value;
public:
   // Construct object with a single reference to it.
   counted() {my_count=1;}
   // Add reference
   void add_ref() {++my_count;}
   // Remove reference. Return true if it was the last reference.
   bool remove_ref() {return --my_count==0;}
   // Get reference to underlying object
   T& get() {
       assert(my_count>0);
       return my_value;
   }
};
```

It is incorrect to use a separate read for testing if the count is zero. The following code would be an incorrect implementation of method `remove_ref`() because two threads might both execute the decrement, and then both read `my_count` as zero. Hence two callers would both be told incorrectly that they had removed the last reference.

```
--my_count;
return my_count==0. // WRONG!
```

The decrement may need to have a *release* fence so that any pending writes complete before the object is deleted.

There is no simple way to atomically copy a pointer and increment its reference count, because there will be a timing hole between the copying and the increment where the reference count is too low, and thus another thread might decrement the count to zero and delete the object. Two ways to address the problem are "hazard pointers" and "pass the buck". See the references below for details.

Variations

Atomic increment/decrement can be more than an order of magnitude more expensive than ordinary increment/decrement. The serial optimization of eliminating redundant increment/decrement operations becomes more important with atomic reference counts.

Weighted reference counting can be used to reduce costs if the pointers are unshared but the referent is shared. Associate a *weight* with each pointer. The reference count is the sum of the weights. A pointer `x` can be copied as a pointer `x'` without updating the reference count by splitting the original weight between `x` and `x'`. If the weight of `x` is too low to split, then first add a constant W to the reference count and weight of `x`.

References

D. Bacon and V.T. Rajan, "Concurrent Cycle Collection in Reference Counted Systems" in Proc. European Conf. on Object-Oriented Programming (June 2001). Describes a garbage collector based on reference counting that does collect cycles.

M. Michael, "Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects" in IEEE Transactions on Parallel and Distributed Systems (June 2004). Describes the "hazard pointer" technique.

M. Herlihy, V. Luchangco, and M. Moir, "The Repeat Offender Problem: A Mechanism for Supporting Dynamic-Sized, Lock-Free Data Structures" in Proceedings of the 16th International Symposium on Distributed Computing (Oct. 2002). Describes the "pass the buck" technique.

General References

This section lists general references. References specific to a pattern are listed at the end of the topic for the pattern.

- **1.** Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns (1995)
- Berkeley Pattern Language for Parallel Programming
- **1.** Mattson, B. Sanders, B. Massingill. Patterns for Parallel Programming (2005)
- ParaPLoP 2009
- ParaPLoP 2010
- Eun-Gyu Kim and Marc Snir, Parallel Programming Patterns

Migrating from Threading Building Blocks (TBB)

While oneTBB is mostly source compatible with TBB, some interfaces were deprecated in TBB and redesigned or removed in oneTBB. This section considers the most difficult use cases for migrating TBB to oneTBB.

- Migrating from tbb::task_scheduler_init
    - Managing the number of threads
        - Querying the default number of threads
        - Setting the maximum concurrency
        - Examples
    - Setting thread stack size
    - Terminating oneTBB scheduler
- Migrating from low-level task API
    - Spawning of individual tasks
        - Using `oneapi::tbb::task_group`
        - Using `oneapi::tbb::parallel_invoke`
    - Adding more work during task execution
    - Task recycling
        - Recycling as child or continuation
    - Scheduler Bypass
    - Deferred task creation
- Mixing two runtimes

Migrating from tbb::task_scheduler_init

`tbb::task_scheduler_init` was a multipurpose functionality in the previous versions of Threading Building Blocks (TBB). This section considers different use cases and how they can be covered with oneTBB.

Managing the number of threads

Querying the default number of threads

- oneapi::tbb::info::default_concurrency() returns the maximum concurrency that will be created by *default* in implicit or explicit `task_arena`.
- oneapi::tbb::this_task_arena::max_concurrency() returns the maximum number of threads available for the parallel algorithms within the current context (or *default* if an implicit arena is not initialized)
- oneapi::tbb::global_control::active_value(tbb::global_control::max_allowed_parallelism) returns the current limit of the thread pool (or *default* if oneTBB scheduler is not initialized)

Setting the maximum concurrency

- task_arena(/* max_concurrency */) limits the maximum concurrency of the parallel algorithm running inside `task_arena`
- tbb::global_control(tbb::global_control::max_allowed_parallelism, /* max_concurrency */) limits the total number of oneTBB worker threads

Examples

The default parallelism:

```
#include <oneapi/tbb/info.h>
#include <oneapi/tbb/parallel_for.h>
#include <oneapi/tbb/task_arena.h>
#include <cassert>

int main() {
    // Get the default number of threads
    int num_threads = oneapi::tbb::info::default_concurrency();

    // Run the default parallelism
    oneapi::tbb::parallel_for( /* ... */ [] {
        // Assert the maximum number of threads
        assert(num_threads == oneapi::tbb::this_task_arena::max_concurrency());
    });

    // Create the default task_arena
    oneapi::tbb::task_arena arena;
    arena.execute([]{
        oneapi::tbb::parallel_for( /* ... */ [] {
            // Assert the maximum number of threads
            assert(num_threads == oneapi::tbb::this_task_arena::max_concurrency());
        });
    });

    return 0;
}
```

The limited parallelism:

```
#include <oneapi/tbb/info.h>
#include <oneapi/tbb/parallel_for.h>
#include <oneapi/tbb/task_arena.h>
#include <oneapi/tbb/global_control.h>
#include <cassert>

int main() {
    // Create the custom task_arena with four threads
    oneapi::tbb::task_arena arena(4);
    arena.execute([]{
        oneapi::tbb::parallel_for( /* ... */ [] {
            // This arena is limited with for threads
            assert(oneapi::tbb::this_task_arena::max_concurrency() == 4);
        });
    });

    // Limit the number of threads to two for all oneTBB parallel interfaces
    oneapi::tbb::global_control
global_limit(oneapi::tbb::global_control::max_allowed_parallelism, 2);

    // the default parallelism
    oneapi::tbb::parallel_for( /* ... */ [] {
        // No more than two threads is expected; however,
tbb::this_task_arena::max_concurrency() can return a bigger value
        int thread_limit =
oneapi::tbb::global_control::active_value(oneapi::tbb::global_control::max_allowed_parallelism);
        assert(thread_limit == 2);
    });

    arena.execute([]{
        oneapi::tbb::parallel_for( /* ... */ [] {
            // No more than two threads is expected; however,
tbb::this_task_arena::max_concurrency() is four
            int thread_limit =
oneapi::tbb::global_control::active_value(oneapi::tbb::global_control::max_allowed_parallelism);
            assert(thread_limit == 2);
            assert(tbb::this_task_arena::max_concurrency() == 4);
        });
    });

    return 0;
}
```

Setting thread stack size

Use oneapi::tbb::global_control(oneapi::tbb::global_control::thread_stack_size, /* stack_size */) to set the stack size for oneTBB worker threads:

```
#include <oneapi/tbb/parallel_for.h>
#include <oneapi/tbb/global_control.h>

int main() {
    // Set 16 MB of the stack size for oneTBB worker threads.
    // Note that the stack size of the main thread should be configured in accordace with the
    // system documentation, e.g. at application startup moment
    oneapi::tbb::global_control global_limit(tbb::global_control::thread_stack_size, 16 * 1024 *
1024);
```

```
    oneapi::tbb::parallel_for( /* ... */ [] {
        // Create a big array in the stack
        char big_array[10*1024*1024];
    });

    return 0;
}
```

Terminating oneTBB scheduler

task_scheduler_handle Class allows waiting for oneTBB worker threads completion:

```
#include <oneapi/tbb/global_control.h>
#include <oneapi/tbb/parallel_for.h>

int main() {
    oneapi::tbb::task_scheduler_handle handle{tbb::attach{}};
    // Do some parallel work here
    oneapi::tbb::parallel_for(/* ... */);
    oneapi::tbb::finalize(handle);
    return 0;
}
```

Migrating from low-level task API

The low-level task API of Intel(R) Threading Building Blocks (TBB) was considered complex and hence error-prone, which was the primary reason it had been removed from oneAPI Threading Building Blocks (oneTBB). This guide helps with the migration from TBB to oneTBB for the use cases where low-level task API is used.

Spawning of individual tasks

For most use cases, the spawning of individual tasks can be replaced with the use of either `oneapi::tbb::task_group` or `oneapi::tbb::parallel_invoke`.

For example, `RootTask`, `ChildTask1`, and `ChildTask2` are the user-side functors that inherit `tbb::task` and implement its interface. Then spawning of `ChildTask1` and `ChildTask2` tasks that can execute in parallel with each other and waiting on the `RootTask` is implemented as:

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

int main() {
    // Assuming RootTask, ChildTask1, and ChildTask2 are defined.
    RootTask& root = *new(tbb::task::allocate_root()) RootTask{};

    ChildTask1& child1 = *new(root.allocate_child()) ChildTask1{/*params*/};
    ChildTask2& child2 = *new(root.allocate_child()) ChildTask2{/*params*/};

    root.set_ref_count(3);

    tbb::task::spawn(child1);
    tbb::task::spawn(child2);

    root.wait_for_all();
}
```

Using `oneapi::tbb::task_group`

The code above can be rewritten using `oneapi::tbb::task_group`:

```
#include <oneapi/tbb/task_group.h>

int main() {
    // Assuming ChildTask1, and ChildTask2 are defined.
    oneapi::tbb::task_group tg;
    tg.run(ChildTask1{/*params*/});
    tg.run(ChildTask2{/*params*/});
    tg.wait();
}
```

The code looks more concise now. It also enables lambda functions and does not require you to implement `tbb::task` interface that overrides the `tbb::task* tbb::task::execute()` virtual method. With this new approach, you work with functors in a C++-standard way by implementing `void operator() const`:

```
struct Functor {
    // Member to be called when object of this type are passed into
    // oneapi::tbb::task_group::run() method
    void operator()() const {}
};
```

Using `oneapi::tbb::parallel_invoke`

It is also possible to use `oneapi::tbb::parallel_invoke` to rewrite the original code and make it even more concise:

```
#include <oneapi/tbb/parallel_invoke.h>

int main() {
    // Assuming ChildTask1, and ChildTask2 are defined.
    oneapi::tbb::parallel_invoke(
        ChildTask1{/*params*/},
        ChildTask2{/*params*/}
    );
}
```

Adding more work during task execution

`oneapi::tbb::parallel_invoke` follows a blocking style of programming, which means that it completes only when all functors passed to the parallel pattern complete their execution.

In TBB, cases when the amount of work is not known in advance and the work needs to be added during the execution of a parallel algorithm were mostly covered by `tbb::parallel_do` high-level parallel pattern. The `tbb::parallel_do` algorithm logic may be implemented using the task API as:

```
#include <cstddef>
#include <vector>
#include <tbb/task.h>

// Assuming RootTask and OtherWork are defined and implement tbb::task interface.

struct Task : public tbb::task {
    Task(tbb::task& root, int i)
        : m_root(root), m_i(i)
    {}

    tbb::task* execute() override {
        // ... do some work for item m_i ...
        if (add_more_parallel_work) {
            tbb::task& child = *new(m_root.allocate_child()) OtherWork;
            tbb::task::spawn(child);
        }
        return nullptr;
    }

    tbb::task& m_root;
    int m_i;
};

int main() {
    std::vector<int> items = { 0, 1, 2, 3, 4, 5, 6, 7 };
    RootTask& root = *new(tbb::task::allocate_root()) RootTask{/*params*/};

    root.set_ref_count(items.size() + 1);

    for (std::size_t i = 0; i < items.size(); ++i) {
        Task& task = *new(root.allocate_child()) Task(root, items[i]);
        tbb::task::spawn(task);
    }

    root.wait_for_all();
    return 0;
}
```

In oneTBB `tbb::parallel_do` interface was removed. Instead, the functionality of adding new work was included into the `oneapi::tbb::parallel_for_each` interface.

The previous use case can be rewritten in oneTBB as follows:

```
#include <vector>
#include <oneapi/tbb/parallel_for_each.h>

int main() {
    std::vector<int> items = { 0, 1, 2, 3, 4, 5, 6, 7 };

    oneapi::tbb::parallel_for_each(
        items.begin(), items.end(),
        [](int& i, tbb::feeder<int>& feeder) {

            // ... do some work for item i ...

            if (add_more_parallel_work)
                feeder.add(i);
        }
    );
}
```

Since both TBB and oneTBB support nested expressions, you can run additional functors from within an already running functor.

The previous use case can be rewritten using `oneapi::tbb::task_group` as:

```
#include <cstddef>
#include <vector>
#include <oneapi/tbb/task_group.h>

int main() {
    std::vector<int> items = { 0, 1, 2, 3, 4, 5, 6, 7 };
    oneapi::tbb::task_group tg;
    for (std::size_t i = 0; i < items.size(); ++i) {
        tg.run([&i = items[i], &tg] {

            // ... do some work for item i ...

            if (add_more_parallel_work)
                // Assuming OtherWork is defined.
                tg.run(OtherWork{});

        });
    }
    tg.wait();
}
```

Task recycling

You can re-run the functor by passing `*this` to the `oneapi::tbb::task_group::run()` method. The functor will be copied in this case. However, its state can be shared among instances:

```
#include <memory>
#include <oneapi/tbb/task_group.h>

struct SharedStateFunctor {
    std::shared_ptr<Data> m_shared_data;
    oneapi::tbb::task_group& m_task_group;

    void operator()() const {
        // do some work processing m_shared_data

        if (has_more_work)
            m_task_group.run(*this);

        // Note that this might be concurrently accessing m_shared_data already
    }
};

int main() {
    // Assuming Data is defined.
    std::shared_ptr<Data> data = std::make_shared<Data>(/*params*/);
    oneapi::tbb::task_group tg;
    tg.run(SharedStateFunctor{data, tg});
    tg.wait();
}
```

Such patterns are particularly useful when the work within a functor is not completed but there is a need for the task scheduler to react to outer circumstances, such as cancellation of group execution. To avoid issues with concurrent access, it is recommended to submit it for re-execution as the last step:

```
#include <memory>
#include <oneapi/tbb/task_group.h>

struct SharedStateFunctor {
    std::shared_ptr<Data> m_shared_data;
    oneapi::tbb::task_group& m_task_group;

    void operator()() const {
        // do some work processing m_shared_data
        if (need_to_yield) {
            m_task_group.run(*this);
            return;
        }
    }
};

int main() {
    // Assuming Data is defined.
    std::shared_ptr<Data> data = std::make_shared<Data>(/*params*/);
    oneapi::tbb::task_group tg;
    tg.run(SharedStateFunctor{data, tg});
    tg.wait();
}
```

Recycling as child or continuation

In oneTBB this kind of recycling is done manually. You have to track when it is time to run the task:

```
#include <cstddef>
#include <vector>
#include <atomic>
#include <cassert>
#include <oneapi/tbb/task_group.h>

struct ContinuationTask {
    ContinuationTask(std::vector<int>& data, int& result)
        : m_data(data), m_result(result)
    {}

    void operator()() const {
        for (const auto& item : m_data)
            m_result += item;
    }

    std::vector<int>& m_data;
    int& m_result;
};

struct ChildTask {
    ChildTask(std::vector<int>& data, int& result,
              std::atomic<std::size_t>& tasks_left, std::atomic<std::size_t>& tasks_done,
              oneapi::tbb::task_group& tg)
        : m_data(data), m_result(result), m_tasks_left(tasks_left), m_tasks_done(tasks_done),
m_tg(tg)
    {}

    void operator()() const {
        std::size_t index = --m_tasks_left;
        m_data[index] = produce_item_for(index);
        std::size_t done_num = ++m_tasks_done;
        if (index % 2 != 0) {
            // Recycling as child
            m_tg.run(*this);
            return;
        } else if (done_num == m_data.size()) {
            assert(m_tasks_left == 0);
            // Spawning a continuation that does reduction
            m_tg.run(ContinuationTask(m_data, m_result));
        }
    }
    std::vector<int>& m_data;
    int& m_result;
    std::atomic<std::size_t>& m_tasks_left;
    std::atomic<std::size_t>& m_tasks_done;
    oneapi::tbb::task_group& m_tg;
};


int main() {
    int result = 0;
    std::vector<int> items(10, 0);
    std::atomic<std::size_t> tasks_left{items.size()};
    std::atomic<std::size_t> tasks_done{0};

    oneapi::tbb::task_group tg;
    for (std::size_t i = 0; i < items.size(); i+=2) {
        tg.run(ChildTask(items, result, tasks_left, tasks_done, tg));
    }
    tg.wait();
}
```

Scheduler Bypass

TBB `task::execute()` method can return a pointer to a task that can be executed next by the current thread. This might reduce scheduling overheads compared to direct `spawn`. Similar to `spawn`, the returned task is not guaranteed to be executed next by the current thread.

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

// Assuming OtherTask is defined.

struct Task : tbb::task {
    task* execute(){
        // some work to do ...

        auto* other_p = new(this->parent().allocate_child()) OtherTask{};
        this->parent().add_ref_count();

        return other_p;
    }
};

int main(){
    // Assuming RootTask is  defined.
    RootTask& root = *new(tbb::task::allocate_root()) RootTask{};

    Task& child = *new(root.allocate_child()) Task{/*params*/};

    root.add_ref_count();

    tbb::task_spawn(child);

    root.wait_for_all();;
}
```

In oneTBB, this can be done using `oneapi::tbb::task_group`.

```
#include <oneapi/tbb/task_group.h>

// Assuming OtherTask is defined.

int main(){
    oneapi::tbb::task_group tg;

    tg.run([&tg](){
        //some work to do ...

        return tg.defer(OtherTask{});
    });

    tg.wait();
}
```

Here `oneapi::tbb::task_group::defer` adds a new task into the `tg`. However, the task is not put into a queue of tasks ready for execution via `oneapi::tbb::task_group::run`, but bypassed to the executing thread directly via function return value.

Deferred task creation

The TBB low-level task API separates the task creation from the actual spawning. This separation allows to postpone the task spawning, while the parent task and final result production are blocked from premature leave. For example, `RootTask`, `ChildTask`, and `CallBackTask` are the user-side functors that inherit `tbb::task` and implement its interface. Then, blocking the `RootTask` from leaving prematurely and waiting on it is implemented as follows:

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

int main() {
    // Assuming RootTask, ChildTask, and CallBackTask are defined.
    RootTask& root = *new(tbb::task::allocate_root()) RootTask{};

    ChildTask&    child    = *new(root.allocate_child()) ChildTask{/*params*/};
    CallBackTask& cb_task  = *new(root.allocate_child()) CallBackTask{/*params*/};

    root.set_ref_count(3);

    tbb::task::spawn(child);

    register_callback([cb_task&](){
        tbb::task::enqueue(cb_task);
    });

    root.wait_for_all();
    // Control flow will reach here only after both ChildTask and CallBackTask are executed,
    // i.e. after the callback is called
}
```

In oneTBB, this can be done using `oneapi::tbb::task_group`.

```
#include <oneapi/tbb/task_group.h>

int main(){
    oneapi::tbb::task_group tg;
    oneapi::tbb::task_arena arena;
    // Assuming ChildTask and CallBackTask are defined.
    auto cb = tg.defer(CallBackTask{/*params*/});

    register_callback([&tg, c = std::move(cb), &arena]{
        arena.enqueue(c);
    });

    tg.run(ChildTask{/*params*/});


    tg.wait();
    // Control flow gets here once both ChildTask and CallBackTask are executed
    // i.e. after the callback is called
}
```

Here `oneapi::tbb::task_group::defer` adds a new task into the `tg`. However, the task is not spawned until `oneapi::tbb::task_arena::enqueue` is called.

NOTE The call to `oneapi::tbb::task_group::wait` will not return control until both `ChildTask` and `CallBackTask` are executed.

Mixing two runtimes

Threading Building Blocks (TBB) and oneAPI Threading Building Blocks (oneTBB) can be safely used in the same application. TBB and oneTBB runtimes are named differently and can be loaded safely within the same process. In addition, the ABI versioning is completely different that prevents symbols conflicts.

However, if both runtimes are loaded into the same process it can lead to oversubscription because each runtime will use its own pool of threads. It might lead to a performance penalty due to increased number of context switches. To check if both TBB and oneTBB are loaded to the application, export `TBB_VERSION=1` before the application run. If both runtimes are loaded there will be two blocks of output, for example:

oneTBB possible output:

```
oneTBB: SPECIFICATION VERSION       1.0
oneTBB: VERSION                2021.2
oneTBB: INTERFACE VERSION      12020
oneTBB: TBB_USE_DEBUG          1
oneTBB: TBB_USE_ASSERT         1
oneTBB: TOOLS SUPPORT          disabled
```

TBB possible output:

```
TBB: VERSION                   2018.0
TBB: INTERFACE VERSION         10006
TBB: BUILD_DATE                Mon 01 Mar 2021 01:28:40 PM UTC
TBB: BUILD_HOST                localhost (x86_64)
TBB: BUILD_OS                  Fedora release 32 (Thirty Two)
TBB: BUILD_KERNEL      Linux 5.8.9-200.fc32.x86_64 #1 SMP Mon Sep 14 18:28:45 UTC 2020
TBB: BUILD_GCC                 g++ (GCC) 10.2.1 20201125 (Red Hat 10.2.1-9)
TBB: BUILD_LIBC        2.31
TBB: BUILD_LD                  GNU ld version 2.34-6.fc32
TBB: BUILD_TARGET      intel64 on cc10_libc2.31_kernel5.8.9
TBB: BUILD_COMMAND     g++ -DDO_ITT_NOTIFY -g -O2 -DUSE_PTHREAD -m64 -fPIC -D__TBB_BUILD=1 -Wall -
Wno-parentheses -Wno-non-virtual-dtor -I../../src -I../../src/rml/include -I../../include -I.
TBB: TBB_USE_DEBUG  0
TBB: TBB_USE_ASSERT 0
```

```
TBB: DO_ITT_NOTIFY  1
TBB: RML    private
TBB: Tools support  disabled
```

Constrained APIs

Starting from C++20, most of Intel® oneAPI Threading Building Blocks (oneTBB) APIs are constrained to enforce named requirements on template arguments types.

The violations of these requirements are detected at a compile time during the template instantiation.

Example

```
// Call for body(oneapi::tbb::blocked_range) is ill-formed
// oneapi::tbb::parallel_for call results in constraint failure
auto body = [](const int& r) { /*...*/ };
oneapi::tbb::parallel_for(oneapi::tbb::blocked_range{1, 10}, body);

// Error example:
// error: no matching function to call to oneapi::tbb::parallel_for
// note: constraints not satisfied
// note: the required expression 'body(range)' is invalid
        body(range);
```

Caution The code that violates named requirements but compiles successfully until C++20, may not compile in C++20 mode due to early and strict constraints diagnostics.

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 it 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® oneAPI Threading Building Blocks (oneTBB) can be mixed with other threading packages. No special effort is required to use any part of oneTBB with other threading packages.

Here is an example that parallelizes an outer loop with OpenMP and an inner loop with oneTBB.

```
int M, N;


struct InnerBody {
    ...
};


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

The details of `InnerBody` are omitted for brevity. The `#pragma omp parallel` causes the OpenMP to create a team of threads, and each thread executes the block statement associated with the pragma. The `#pragma omp for` indicates that the compiler should use the previously created thread team to execute the loop in parallel.

Here is the same example written using POSIX* Threads.

```
int M, N;


struct InnerBody {
    ...
};


void* OuterLoopIteration( void* args ) {
    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.

[3] “Cilk: An Efficient Multithreaded Runtime System”, Robert Blumofe, Christopher Joerg, Bradley Kuszmaul, C. Leiserson, and Keith Randall, Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming, 1995.

oneTBB API Reference

For oneTBB API Reference, refer to oneAPI Specification. The current supported version of oneAPI Specification is 1.0.

Specification extensions

Intel® oneAPI Threading Building Blocks (oneTBB) implements the oneTBB specification. This document provides additional details or restrictions where necessary. It also describes features that are not included in the oneTBB specification.

- oneapi::tbb::info namespace
- parallel_for_each Body semantics and requirements
- parallel_sort ranges interface extension

Preview features

A preview feature is a component of oneTBB introduced to receive early feedback from users.

The key properties of a preview feature are:

- It is off by default and must be explicitly enabled.
- It is intended to have a high quality implementation.
- There is no guarantee of future existence or compatibility.
- It may have limited or no support in tools such as correctness analyzers, profilers and debuggers.

Caution A preview feature is subject to change in future. It might be removed or significantly altered in future releases. Changes to a preview feature do NOT require usual deprecation and removal process. Therefore, using preview features in production code is strongly discouraged.

- Type-specified message keys for join_node
- Scalable Memory Pools
  - memory_pool
  - fixed_pool
  - memory_pool_allocator
- Helper Functions for Expressing Graphs
  - Constructors for Flow Graph nodes
  - `follows` and `precedes` function templates
  - `make_node_set` function template
  - `make_edges` function template
- concurrent_lru_cache

- task_arena::constraints extensions
- oneapi::tbb::info namespace extensions
- task_group extensions
- The customizing mutex type for `concurrent_hash_map`

oneapi::tbb::info namespace

The `oneapi::tbb::info` namespace satisfies the corresponding oneTBB specification section.

The Intel® oneAPI Threading Building Blocks (oneTBB) implementation requires the hwloc library to query NUMA(version >= 1.11) and Hybrid CPUs(version >= 2.4) topology information.

See also:

- info namespace specification
- oneapi::tbb::task_arena specification
- oneapi::tbb::info namespace preview extensions
- task_arena::constraints class preview extensions

parallel_for_each Body semantics and requirements

- Description

Description

This page clarifies ParallelForEachBody named requirements for `tbb::parallel_for_each` algorithm specification.

```
namespace oneapi {
    namespace tbb {

        template <typaname InputIterator, typename Body>
        void parallel_for_each( InputIterator first, InputIterator last, Body body ); //
overload (1)
        template <typename InputIterator, typename Body>
        void parallel_for_each( InputIterator first, InputIterator last, Body body,
task_group_context& group ); // overload (2)

        template <typename Container, typename Body>
        void parallel_for_each( Container& c, Body body ); // overload (3)
        template <typename Container, typename Body>
        void parallel_for_each( Container& c, Body body, task_group_context& group ); //
overload (4)

        template <typename Container, typename Body>
        void parallel_for_each( const Container& c, Body body ); // overload (5)
        template <typename Container, typename Body>
        void parallel_for_each( const Container& c, Body body, task_group_context& group ); //
overload (6)

    } // namespace tbb
} // namespace oneapi
```

Terms

- `iterator` determines the type of the iterator passed into `parallel_for_each` algorithm (which is `InputIterator` for overloads **(1)** and **(2)** and `decltype(std::begin(c))` for overloads **(3) - (6)**)

- `value_type` - the type `typename std::iterator_traits<iterator>::value_type`
- `reference` - the type `typename std::iterator_traits<iterator>::reference`.

Requirements for different iterator types

If the `iterator` satisfies **Input iterator** named requirements from [input.iterators] ISO C++ Standard section and do not satisfies **Forward iterator** named requirements from [forward.iterators] ISO C++ Standard section, `tbb::parallel_for_each` requires the execution of the `body` with an object of type `const value_type&` or `value_type&&` to be well-formed. If both forms are well-formed, an overload with rvalue reference will be preferred.

> **Caution** If the `Body` only takes non-const lvalue reference to `value_type`, named requirements above are violated and the program can be ill-formed.

If the `iterator` satisfies **Forward iterator** named requirements from [forward.iterators] ISO C++ Standard section, `tbb::parallel_for_each` requires the execution of the `body` with an object of type `reference` to be well-formed.

Requirements for `Body` with `feeder` argument

Additional elements submitted into `tbb::parallel_for_each` through the `feeder::add` passes to the `Body` as rvalues and therefore the corresponding execution of the `Body` is required to be well-formed.

parallel_sort ranges interface extension

- Description
- API

Description

Intel® oneAPI Threading Building Blocks (oneTBB) implementation extends the oneapi::tbb::parallel_sort specification with overloads that takes the container by forwarding reference.

API

Header

```
#include <oneapi/tbb/parallel_sort.h>
```

Syntax

```
namespace oneapi {
    namespace tbb {

        template <typename Container>
        void parallel_sort( Container&& c );
        template <typename Container, typename Compare>
        void parallel_sort( Container&& c, const Compare& comp );

    } // namespace tbb
} // namespace oneapi
```

Functions

***template<typename*Container>voidparallel_sort(Container&&c);**

Equivalent to `parallel_sort( std::begin(c), std::end(c), comp )`, where **comp** uses **operator<** to determine relative orderings.

***template<typename*Container,*typename*Compare>voidparallel_sort(Container&&c, *const*Compare&comp);**

Equivalent to `parallel_sort( std::begin(c), std::end(c), comp )`.

Example

This interface may be used for sorting rvalue or constant views:

```
#include <array>
#include <span> // requires C++20
#include <oneapi/tbb/parallel_sort.h>

std::span<int> get_span() {
    static std::array<int, 3> arr = {3, 2, 1};
    return std::span<int>(arr);
}

int main() {
    tbb::parallel_sort(get_span());
}
```

Type-specified message keys for join_node

NOTE To enable this feature, define the `TBB_PREVIEW_FLOW_GRAPH_FEATURES` macro to 1.

- Description
- API
- See Also

Description

The extension allows a key matching `join_node` to obtain keys via functions associated with its input types. The extension simplifies the existing approach by removing the need to provide a function object for each input port of `join_node`.

API

Header

```
#include "oneapi/tbb/flow_graph.h"
```

Syntax

The extension adds a special constructor to the `join_node` interface when the `key_matching<typename K, class KHash=tbb_hash_compare>` policy is used. The constructor has the following signature:

```
join_node( graph &g )
```

When constructed this way, a `join_node` calls the `key_from_message` function for each incoming message to obtain the key associated with it. The default implementation of `key_from_message` is the following

```
namespace oneapi {
    namespace tbb {
        namespace flow {
            template <typename K, typename T>
            K key_from_message( const T &t ) {
                return t.key();
            }
        }
    }
}
```

`T` is one of the user-provided types in `OutputTuple` and is used to construct the `join_node`, and `K` is the key type of the node. By default, the `key()` method defined in the message class will be called. Alternatively, the user can define its own `key_from_message` function in the same namespace with the message type. This function will be found via C++ argument-dependent lookup and used in place of the default implementation.

See Also

join_node Specification

Scalable Memory Pools

NOTE To enable this feature, set the `TBB_PREVIEW_MEMORY_POOL` macro to 1.

Memory pools allocate and free memory from a specified region or an underlying allocator using thread-safe, scalable operations. The following table summarizes the Memory Pool named requirement. Here, `P` represents an instance of the memory pool class.

Pseudo-Signature	Semantics
`~P() throw();`	Destructor. Frees all the allocated memory.
`void P::recycle();`	Frees all the allocated memory.
`void* P::malloc(size_t n);`	Returns a pointer to `n` bytes allocated from the memory pool.
`void P::free(void* ptr);`	Frees the memory object specified via `ptr` pointer.
`void* P::realloc(void* ptr, size_t n);`	Reallocates the memory object pointed by `ptr` to `n` bytes.

Model Types

The `memory_pool` template class and the `fixed_pool` class meet the Memory Pool named requirement.

- memory_pool
- fixed_pool
- memory_pool_allocator

memory_pool

NOTE To enable this feature, set the `TBB_PREVIEW_MEMORY_POOL` macro to 1.

A class template for scalable memory allocation from memory blocks provided by an underlying allocator.

- Description
- API
- Examples

Description

A `memory_pool` allocates and frees memory in a way that scales with the number of processors. The memory is obtained as big chunks from an underlying allocator specified by the template argument. The latter must satisfy the subset of the allocator requirements from the [allocator.requirements] ISO C++ Standard section. A `memory_pool` meet the Memory Pool named requirement.

Caution If the underlying allocator refers to another scalable memory pool, the inner pool (or pools) must be destroyed before the outer pool is destroyed or recycled.

API

Header

```
#include "oneapi/tbb/memory_pool.h"
```

Synopsis

```
namespace oneapi {
    namespace tbb {
        template <typename Alloc>
        class memory_pool {
        public:
            explicit memory_pool(const Alloc &src = Alloc());
            memory_pool(const memory_pool& other) = delete;
            memory_pool& operator=(const memory_pool& other) = delete;
            ~memory_pool();
            void recycle();
            void *malloc(size_t size);
            void free(void* ptr);
           void *realloc(void* ptr, size_t size);
        };
    }
}
```

Member Functions

***explicit*memory_pool(*const*Alloc&src=Alloc())**

Effects: Constructs a memory pool with an instance of underlying memory allocator of type `Alloc` copied from `src`. Throws the `bad_alloc` exception if runtime fails to construct an instance of the class.

Examples

The code below provides a simple example of allocation from an extensible memory pool.

```
#define TBB_PREVIEW_MEMORY_POOL 1
#include "oneapi/tbb/memory_pool.h"
...
```

```
oneapi::tbb::memory_pool<std::allocator<char> > my_pool;
void* my_ptr = my_pool.malloc(10);
my_pool.free(my_ptr);
```

fixed_pool

NOTE To enable this feature, set the `TBB_PREVIEW_MEMORY_POOL` macro to 1.

A class for scalable memory allocation from a buffer of fixed size.

- Description
- API
- Examples

Description

`fixed_pool` allocates and frees memory in a way that scales with the number of processors. All the memory available for the allocation is initially passed through arguments of the constructor. `fixed_pool` meet the Memory Pool named requirement.

API

Header

```
#include "oneapi/tbb/memory_pool.h"
```

Synopsis

```
namespace oneapi {
    namespace tbb {
        class fixed_pool {
        public:
            fixed_pool(void *buffer, size_t size);
            fixed_pool(const fixed_pool& other) = delete;
            fixed_pool& operator=(const fixed_pool& other) = delete;
            ~fixed_pool();

            void recycle();
            void* malloc(size_t size);
            void free(void* ptr);
            void* realloc(void* ptr, size_t size);
        };
    } // namespace tbb
} // namespace oneapi
```

Member Functions

fixed_pool(void*buffer, size_tsize)

Effects: Constructs a memory pool to manage the memory of size `size` pointed to by `buffer`. Throws the `bad_alloc` exception if the library fails to construct an instance of the class.

Examples

The code below provides a simple example of allocation from a fixed pool.

```
#define TBB_PREVIEW_MEMORY_POOL 1
#include "oneapi/tbb/memory_pool.h"
...
char buf[1024*1024];
oneapi::tbb::fixed_pool my_pool(buf, 1024*1024);
void* my_ptr = my_pool.malloc(10);
my_pool.free(my_ptr);}
```

memory_pool_allocator

NOTE To enable this feature, set the `TBB_PREVIEW_MEMORY_POOL` macro to 1.

A class template that provides a memory pool with a C++ allocator interface.

- Description
- API
- Examples

Description

`memory_pool_allocator` meets the allocator requirements from the [allocator.requirements] ISO C++ Standard section It also provides a constructor to allocate and deallocate memory. This constructor is linked with an instance of either the `memory_pool` or the `fixed_pool` class. The class is mainly intended for enabling memory pools within STL containers.

API

Header

```
#include "oneapi/tbb/memory_pool.h"
```

Synopsis

```
namespace oneapi {
    namespace tbb {
        template<typename T>
        class memory_pool_allocator {
        public:
            using value_type = T;
            using pointer = value_type*;
            using const_pointer = const value_type*;
            using reference = value_type&;
            using const_reference = const value_type&;
            using size_type = size_t;
            using difference_type = ptrdiff_t;
            template<typename U> struct rebind {
                using other = memory_pool_allocator<U>;
            };
            explicit memory_pool_allocator(memory_pool &pool) throw();
            explicit memory_pool_allocator(fixed_pool &pool) throw();
            memory_pool_allocator(const memory_pool_allocator& src) throw();
            template<typename U>
            memory_pool_allocator(const memory_pool_allocator<U,P>& src) throw();
            pointer address(reference x) const;
            const_pointer address(const_reference x) const;
            pointer allocate(size_type n, const void* hint=0);
            void deallocate(pointer p, size_type);
            size_type max_size() const throw();
            void construct(pointer p, const T& value);
            void destroy(pointer p);
        };

        template<>
        class memory_pool_allocator<void> {
        public:
            using pointer = void*;
            using const_pointer = const void*;
            using value_type = void;
            template<typename U> struct rebind {
                using other = memory_pool_allocator<U>;
            };
            memory_pool_allocator(memory_pool &pool) throw();
            memory_pool_allocator(fixed_pool &pool) throw();
            memory_pool_allocator(const memory_pool_allocator& src) throw();
            template<typename U>
            memory_pool_allocator(const memory_pool_allocator<U>& src) throw();
        };
    } // namespace tbb
} // namespace oneapi

template<typename T, typename U>
inline bool operator==( const memory_pool_allocator<T>& a,
                       const memory_pool_allocator<U>& b);
template<typename T, typename U>
inline bool operator!=( const memory_pool_allocator<T>& a,
                        const memory_pool_allocator<U>& b);
```

Member Functions

***explicit*memory_pool_allocator(memory_pool&pool)**

Effects: Constructs a memory pool allocator serviced by `memory_pool` instance pool.

***explicit*memory_pool_allocator(fixed_pool&pool)**

Effects: Constructs a memory pool allocator serviced by `fixed_pool` instance pool.

Examples

The code below provides a simple example of container construction with the use of a memory pool.

```
#define TBB_PREVIEW_MEMORY_POOL 1
#include "oneapi/tbb/memory_pool.h"
...
typedef oneapi::tbb::memory_pool_allocator<int>
pool_allocator_t;
std::list<int, pool_allocator_t> my_list(pool_allocator_t( my_pool ));
```

Helper Functions for Expressing Graphs

NOTE To enable this feature, define the `TBB_PREVIEW_FLOW_GRAPH_FEATURES` macro to 1.

Helper functions are intended to make creation of the flow graphs less verbose.

- Description
- API
- Example

Description

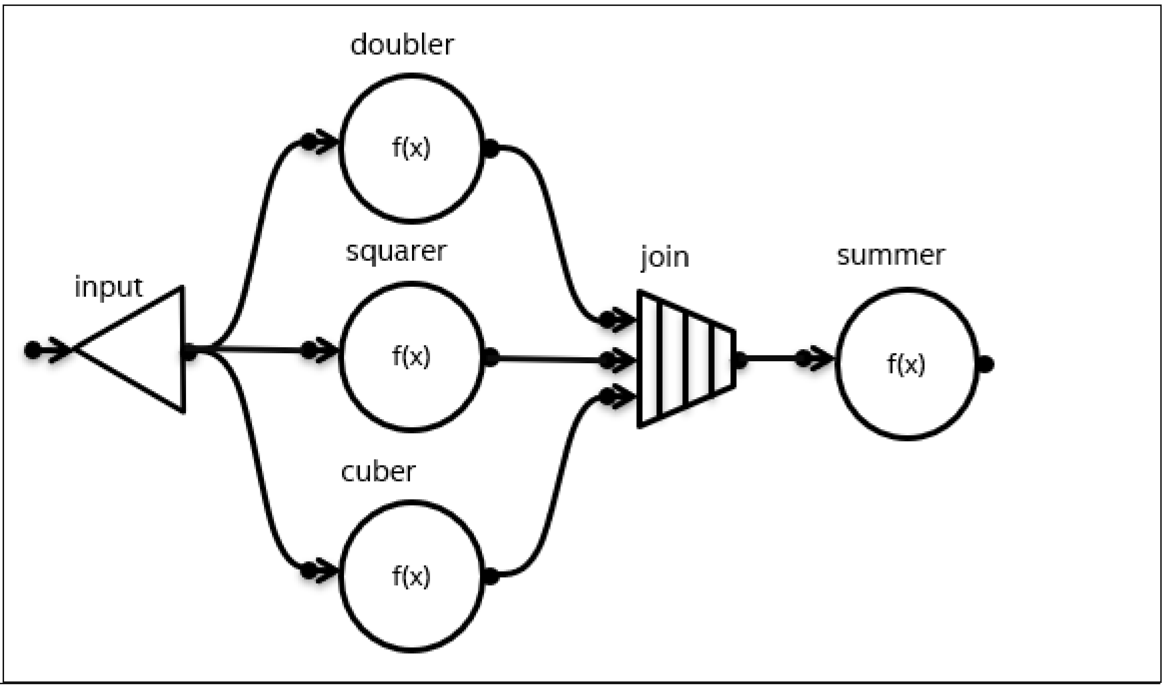
This feature adds `make_edges`, `make_node_set`, `follows` and `precedes` functions to `oneapi::tbb::flow` namespace. These functions simplify the process of building flow graphs by allowing to gather nodes into sets and connect them to other nodes in the graph.

API

- Constructors for Flow Graph nodes
- `follows` and `precedes` function templates
- `make_node_set` function template
- `make_edges` function template

Example

Consider the graph depicted below.



In the examples below, C++17 Class Template Argument Deduction is used to avoid template parameter specification where possible.

Regular API

```
#include <oneapi/tbb/flow_graph.h>

int main() {
    using namespace oneapi::tbb::flow;

    graph g;
    broadcast_node<int> input(g);

    function_node doubler(g, unlimited, [](const int& v) { return 2 * v; });
    function_node squarer(g, unlimited, [](const int&) { return v * v; });
    function_node cuber(g, unlimited, [](const int& v) { return v * v * v; });

    join_node<std::tuple<int, int, int>> join(g);

    int sum = 0;
    function_node summer(g, serial, [&](const std::tuple<int, int, int>& v) {
        int sub_sum = std::get<0>(v) + std::get<1>(v) + std::get<2>(v);
        sum += sub_sum;
        return sub_sum;
    });

    make_edge(input, doubler);
    make_edge(input, squarer);
    make_edge(input, cuber);
    make_edge(doubler, std::get<0>(join.input_ports()));
    make_edge(squarer, std::get<1>(join.input_ports()));
    make_edge(cuber, std::get<2>(join.input_ports()));
    make_edge(join, summer);

    for (int i = 1; i <= 10; ++i) {
        input.try_put(i);
    }
    g.wait_for_all();
}
```

Preview API

```
#define TBB_PREVIEW_FLOW_GRAPH_FEATURES 1
#include <oneapi/tbb/flow_graph.h>

int main() {
    using namespace oneapi::tbb::flow;

    graph g;

    function_node doubler(g, unlimited, [](const int& v) { return 2 * v; });
    function_node squarer(g, unlimited, [](const int&) { return v * v; });
    function_node cuber(g, unlimited, [](const int& v) { return v * v * v; });

    auto handlers = make_node_set(doubler, squarer, cuber);

    broadcast_node input(precedes(handlers));
    join_node join(follows(handlers));

    int sum = 0;
    function_node summer(follows(join), serial,
                         [&](const std::tuple<int, int, int>& v) {
                             int sub_sum = std::get<0>(v) + std::get<1>(v) + std::get<2>(v);
                             sum += sub_sum;
                             return sub_sum;
                         });

    for (int i = 1; i <= 10; ++i) {
        input.try_put(i);
    }
    g.wait_for_all();
}
```

Constructors for Flow Graph nodes

NOTE To enable this feature, define the `TBB_PREVIEW_FLOW_GRAPH_FEATURES` macro to 1.

Description

The “Helper Functions for Expressing Graphs” feature adds a set of new constructors that can be used to construct a node that `follows` or `precedes` a set of nodes.

Where possible, the constructors support Class Template Argument Deduction (since C++17).

API

Header

```
#include <oneapi/tbb/flow_graph.h>
```

Syntax

```
// continue_node
continue_node(follows(...), Body body, Policy = Policy());
continue_node(precedes(...), Body body, Policy = Policy());

continue_node(follows(...), int number_of_predecessors, Body body, Policy = Policy());
continue_node(precedes(...), int number_of_predecessors, Body body, Policy = Policy());

// function_node
function_node(follows(...), std::size_t concurrency, Policy = Policy());
function_node(precedes(...), std::size_t concurrency, Policy = Policy());

// input_node
input_node(precedes(...), body);

// multifunction_node
multifunction_node(follows(...), std::size_t concurrency, Body body);
multifunction_node(precedes(...), std::size_t concurrency, Body body);

// async_node
async_node(follows(...), std::size_t concurrency, Body body);
async_node(precedes(...), std::size_t concurrency, Body body);

// overwrite_node
explicit overwrite_node(follows(...));
explicit overwrite_node(precedes(...));

// write_once_node
explicit write_once_node(follows(...));
explicit write_once_node(precedes(...));
```

```
// buffer_node
explicit buffer_node(follows(...));
explicit buffer_node(precedes(...));

// queue_node
explicit queue_node(follows(...));
explicit queue_node(precedes(...));

// priority_queue_node
explicit priority_queue_node(follows(...), const Compare& comp = Compare());
explicit priority_queue_node(precedes(...), const Compare& compare = Compare());

// sequencer_node
sequencer_node(follows(...), const Sequencer& s);
sequencer_node(precedes(...), const Sequencer& s);

// limiter_node
limiter_node(follows(...), std::size_t threshold);
limiter_node(precedes(...), std::size_t threshold);

// broadcast_node
explicit broadcast_node(follows(...));
explicit broadcast_node(precedes(...));

// join_node
explicit join_node(follows(...), Policy = Policy());
explicit join_node(precedes(...), Policy = Policy());

// split_node
explicit split_node(follows(...));
explicit split_node(precedes(...));

// indexer_node
indexer_node(follows(...));
indexer_node(precedes(...));
```

See Also

follows and precedes function templates

follows and precedes function templates

NOTE To enable this feature, define the `TBB_PREVIEW_FLOW_GRAPH_FEATURES` macro to 1.

The `follows` and `precedes` helper functions aid in expressing dependencies between nodes when building oneTBB flow graphs. These helper functions can only be used while constructing the node.

- Description
- API

Description

The `follows` helper function specifies that the node being constructed is the successor of the set of nodes passed as an argument.

The `precedes` helper function specifies that the node being constructed is the predecessor of the set of nodes passed as an argument.

Functions `follows` and `precedes` are meant to replace the graph argument, which is passed as the first argument to the constructor of the node. The graph argument for the node being constructed is obtained either from the specified node set or the sequence of nodes passed to `follows` or `precedes`.

If the nodes passed to `follows` or `precedes` belong to different graphs, the behavior is undefined.

API

Header

```
#include <oneapi/tbb/flow_graph.h>
```

Syntax

```
// node_set is an exposition-only name for the type returned from make_node_set function

template <typename NodeType, typename... NodeTypes>
/*unspecified*/ follows( node_set<NodeType, NodeTypes...>& set );

template <typename NodeType, typename... NodeTypes>
/*unspecified*/ follows( NodeType& node, NodeTypes&... nodes );

template <typename NodeType, typename... NodeTypes>
/*unspecified*/ precedes( node_set<NodeType, NodeTypes...>& set );

template <typename NodeType, typename... NodeTypes>
/*unspecified*/ precedes( NodeType& node, NodeTypes&... nodes );
```

Input Parameters

Either a set or a sequence of nodes can be used as arguments for `follows` and `precedes`. The following expressions are equivalent:

```
auto handlers = make_node_set(n1, n2, n3);
broadcast_node<int> input(precedes(handlers));
```

```
broadcast_node<int> input(precedes(n1, n2, n3));
```

make_node_set function template

NOTE To enable this feature, define the `TBB_PREVIEW_FLOW_GRAPH_FEATURES` macro to 1.

- Description
- API
- See Also

Description

The `make_node_set` function template creates a set of nodes that can be passed as arguments to `make_edges`, `follows` and `precedes` functions.

API

Header

```
#include <oneapi/tbb/flow_graph.h>
```

Syntax

```
template <typename Node, typename... Nodes>
/*unspecified*/ make_node_set( Node& node, Nodes&... nodes );
```

See Also

make_edges function template

follows and precedes function templates

make_edges function template

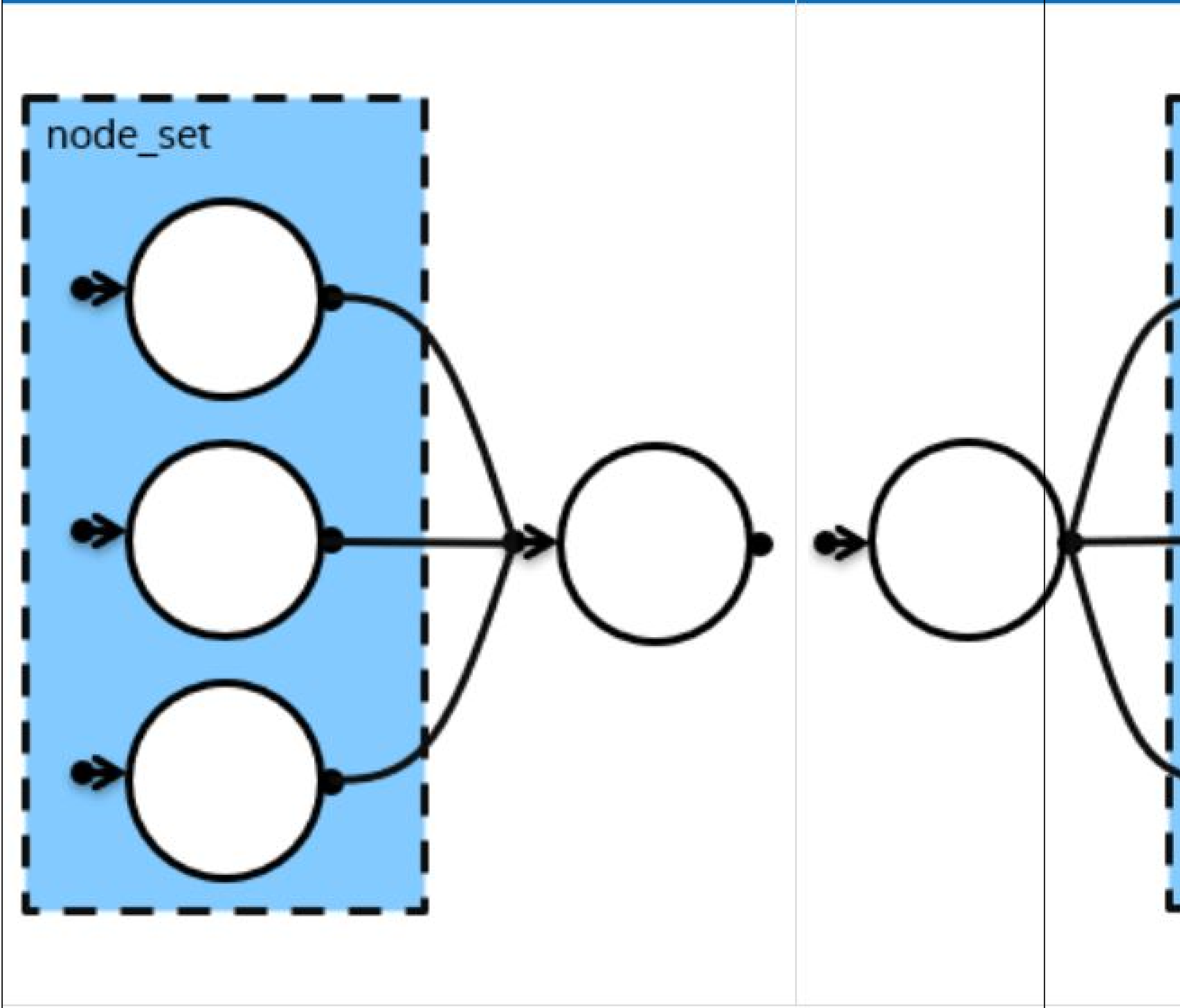
NOTE To enable this feature, define the `TBB_PREVIEW_FLOW_GRAPH_FEATURES` macro to 1.

- Description
- API

Description

The `make_edges` function template creates edges between a single node and each node in a set of nodes.

There are two ways to connect nodes in a set and a single node using `make_edges`:

make_edges(node_set, node)	make_edges(node,
	

API

Header

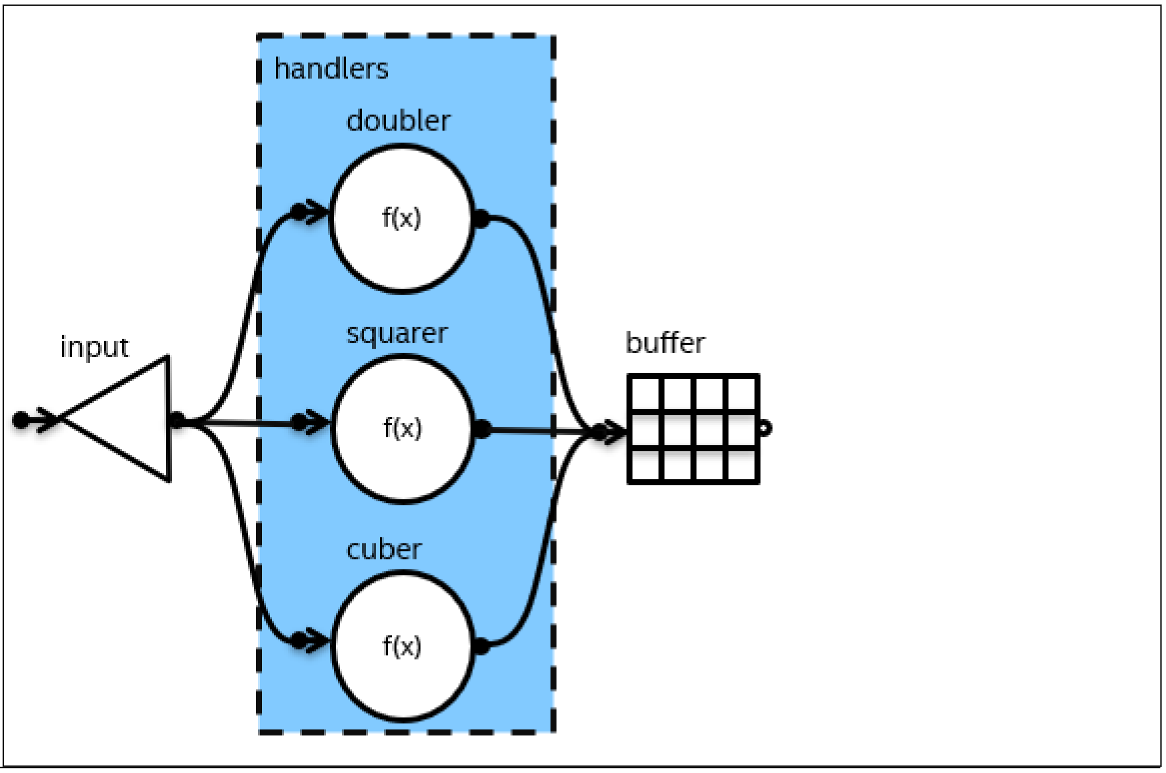
```
#include <oneapi/tbb/flow_graph.h>
```

Syntax

```
// node_set is an exposition-only name for the type returned from make_node_set function

template <typename NodeType, typename Node, typename... Nodes>
void make_edges(node_set<Node, Nodes...>& set, NodeType& node);

template <typename NodeType, typename Node, typename... Nodes>
void make_edges(NodeType& node, node_set<Node, Nodes...>& set);
```

Example

The example implements the graph structure in the picture below.



```
#define TBB_PREVIEW_FLOW_GRAPH_FEATURES 1
#include <oneapi/tbb/flow_graph.h>

int main() {
    using namespace oneapi::tbb::flow;

    graph g;
    broadcast_node<int> input(g);

    function_node doubler(g, unlimited, [](const int& i) { return 2 * i; });
    function_node squarer(g, unlimited, [](const int& i) { return i * i; });
    function_node cuber(g, unlimited, [](const int& i) { return i * i * i; });

    buffer_node<int> buffer(g);

    auto handlers = make_node_set(doubler, squarer, cuber);
    make_edges(input, handlers);
    make_edges(handlers, buffer);

    for (int i = 1; i <= 10; ++i) {
        input.try_put(i);
    }
    g.wait_for_all();
}
```

concurrent_lru_cache

NOTE To enable this feature, define the `TBB_PREVIEW_CONCURRENT_LRU_CACHE` macro to 1.

A Class Template for Least Recently Used cache with concurrent operations.

- Description
- API

Description

A `concurrent_lru_cache` container maps keys to values with the ability to limit the number of stored unused values. For each key, there is at most one item stored in the container.

The container permits multiple threads to concurrently retrieve items from it.

The container tracks which items are in use by returning a proxy `concurrent_lru_cache::handle` object that refers to an item instead of its value. Once there are no `handle` objects holding reference to an item, it is considered unused.

The container stores all the items that are currently in use plus a limited number of unused items. Excessive unused items are erased according to least recently used policy.

When no item is found for a given key, the container calls the user-specified `value_function_type` object to construct a value for the key, and stores that value. The `value_function_type` object must be thread-safe.

API

Header

```
#include "oneapi/tbb/concurrent_lru_cache.h"
```

Synopsis

```
namespace oneapi {
    namespace tbb {
        template <typename Key, typename Value, typename ValueFunctionType = Value (*)(Key)>
        class concurrent_lru_cache {
        public:
            using key_type = Key;
            using value_type = Value;
            using pointer = value_type*;
            using const_pointer = const value_type*;
            using reference = value_type&;
            using const_reference = const value_type&;

            using value_function_type = ValueFunctionType;

            class handle {
            public:
                handle();
                handle( handle&& other );

                ~handle();

                handle& operator=( handle&& other );

                operator bool() const;
                value_type& value();
            }; // class handle
            concurrent_lru_cache( value_function_type f, std::size_t
number_of_lru_history_items );
            ~concurrent_lru_cache();

            handle operator[]( key_type key );
        }; // class concurrent_lru_cache
    } // namespace tbb
} // namespace oneapi
```

Member Functions

concurrent_lru_cache(value_function_typef, std::size_tnumber_of_lru_history_items);

Effects: Constructs an empty cache that can keep up to `number_of_lru_history_items` unused values, with a function object `f` for constructing new values.

~concurrent_lru_cache();

Effects: Destroys the `concurrent_lru_cache`. Calls the destructors of the stored elements and deallocates the used storage.

The behavior is undefined in case of concurrent operations with `*this`.

handle*operator*[](key_typek);

Effects: Searches the container for an item that corresponds to the given key. If such an item is not found, the user-specified function object is called to construct a value that is inserted into the container.

Returns: a `handle` object holding reference to the matching value.

Member Objects

`handle` class

Member Functions

handle();

Effects: Constructs a `handle` object that does not refer to any value.

handle(handle&&other);

Effects: Transfers the reference to the value stored in `concurrent_lru_cache` from `other` to the newly constructed object. Upon completion, `other` no longer refers to any value.

~handle();

Effects: Releases the reference (if it exists) to a value stored in `concurrent_lru_cache`.

The behavior is undefined for concurrent operations with `*this`.

handle&*operator*=(handle&&other);

Effects: Transfers the reference to a value stored in `concurrent_lru_cache` from `other` to `*this`. If existed, the previous reference held by `*this` is released. Upon completion `other` no longer refers to any value.

Returns: a reference to `*this`.

***operator*bool()*const*;**

Returns: `true` if `*this` holds reference to a value, `false` otherwise.

value_type&value();

Returns: a reference to a `value_type` object stored in `concurrent_lru_cache`.

The behavior is undefined if `*this` does not refer to any value.

task_arena::constraints extensions

NOTE To enable this feature, set the `TBB_PREVIEW_TASK_ARENA_CONSTRAINTS_EXTENSION` macro to 1.

- Description
- API

Description

These extensions allow to customize `tbb::task_arena::constraints` with the following properties:

- On machines with Intel® Hybrid Technology set the preferred core type for threads working within the task arena.
- Limit the maximum number of threads that can be scheduled to one core simultaneously.

API

Header

```
#include <oneapi/tbb/task_arena.h>
```

Synopsis

```
namespace oneapi {
    namespace tbb {

        class task_arena {
        public:
            struct constraints {
                constraints& set_numa_id(numa_node_id id);
                constraints& set_max_concurrency(int maximal_concurrency);
                constraints& set_core_type(core_type_id id);
                constraints& set_max_threads_per_core(int threads_number);

                numa_node_id numa_id = task_arena::automatic;
                int max_concurrency = task_arena::automatic;
                core_type_id core_type = task_arena::automatic;
                int max_threads_per_core = task_arena::automatic;
            }; // struct constraints
        }; // class task_arena

    } // namespace tbb
} // namespace oneapi
```

Member Functions

constraints&set_numa_id(numa_node_idid)

Sets the `numa_id` field to the `id`.

Returns: Reference to `*this`.

constraints&set_max_concurrency(intmaximal_concurrency)

Sets the `max_concurrency` field to the `maximal_concurrency`.

Returns: Reference to `*this`.

constraints&set_core_type(core_type_idid)

Sets the `core_type` field to the `id`.

Returns: Reference to `*this`.

constraints&set_max_threads_per_core(intthreads_number)

Sets the `max_threads_per_core` field to the `threads_number`.

Returns: Reference to `*this`.

Member Objects

numa_node_idnuma_id

An integral logical index uniquely identifying a NUMA node. All threads joining the `task_arena` are bound to this NUMA node.

NOTE To obtain a valid NUMA node ID, call `oneapi::tbb::info::numa_nodes()`.

intmax_concurrency

The maximum number of threads that can participate in work processing within the `task_arena` at the same time.

core_type_idcore_type

An integral logical index uniquely identifying a core type. All threads joining the `task_arena` are bound to this core type.

NOTE To obtain a valid core type node ID, call `oneapi::tbb::info::core_types()`.

intmax_threads_per_core

The maximum number of threads that can be scheduled to one core simultaneously.

See also:

- oneapi::tbb::info namespace preview extensions
- oneapi::tbb::task_arena specification

oneapi::tbb::info namespace extensions

NOTE To enable this feature, set the `TBB_PREVIEW_TASK_ARENA_CONSTRAINTS_EXTENSION` macro to `1`.

- Description
- API

Description

These extensions allow to query information about execution environment.

API

Header

```
#include <oneapi/tbb/info.h>
```

Syntax

```
namespace oneapi {
    namespace tbb {
        using core_type_id = /*implementation-defined*/;
        namespace info {
            std::vector<core_type_id> core_types();
            int default_concurrency(task_arena::constraints c);
        }
    }
}
```

Types

`core_type_id` - Represents core type identifier.

Functions

std::vector<core_type_id>core_types()

Returns the vector of integral indexes that indicate available core types. The indexes are sorted from the least performant to the most performant core type.

NOTE If error occurs during system topology parsing, returns vector containing single element that equals to `task_arena::automatic`.

intdefault_concurrency(task_arena::constraintsc)

Returns concurrency level for the given constraints.

See also:

- task_arena::constraints class preview extensions
- info namespace specification

task_group extensions

NOTE To enable these extensions, set the `TBB_PREVIEW_TASK_GROUP_EXTENSIONS` macro to 1.

- Description
- API

Description

Intel® oneAPI Threading Building Blocks (oneTBB) implementation extends the tbb::task_group specification with the following members:

- requirements for a user-provided function object

API

Header

```
#include <oneapi/tbb/task_group.h>
```

Synopsis

```
namespace oneapi {
    namespace tbb {

       class task_group {
       public:

           //only the requirements for the return type of function F are changed
           template<typename F>
           task_handle defer(F&& f);

           //only the requirements for the return type of function F are changed
           template<typename F>
           task_group_status run_and_wait(const F& f);

           //only the requirements for the return type of function F are changed
           template<typename F>
           void run(F&& f);
       };

    } // namespace tbb
} // namespace oneapi
```

Member Functions

template<*typename*F>task_handledefer(F&&f)

As an optimization hint, `F` might return a `task_handle`, which task object can be executed next.

NOTE The `task_handle` returned by the function must be created using `*thistask_group`. That is, the one for which the run method is called, otherwise it is undefined behavior.

template<*typename*F>task_group_statusrun_and_wait(*const*F&f)

As an optimization hint, `F` might return a `task_handle`, which task object can be executed next.

NOTE The `task_handle` returned by the function must be created using `*thistask_group`. That is, the one for which the run method is called, otherwise it is undefined behavior.

template<*typename*F>voidrun(F&&f)

As an optimization hint, `F` might return a `task_handle`, which task object can be executed next.

NOTE The `task_handle` returned by the function must be created with `*thistask_group`. It means, with the one for which run method is called, otherwise it is an undefined behavior.

See also

- oneapi::tbb::task_group specification

- oneapi::tbb::task_group_context specification
- oneapi::tbb::task_group_status specification
- oneapi::tbb::task_handle class

The customizing mutex type for concurrent_hash_map

NOTE To enable this feature, define the `TBB_PREVIEW_CONCURRENT_HASH_MAP_EXTENSIONS` macro to 1.

- Description
- API

Description

oneTBB `concurrnent_hash_map` class uses reader-writer mutex to provide thread safety and avoid data races for insert, lookup, and erasure operations. This feature adds an extra template parameter for `concurrent_hash_map` that allows to customize the type of the reader-writer mutex.

API

Header

```
#include <oneapi/tbb/concurrent_hash_map.h>
```

Synopsis

```
namespace oneapi {
namespace tbb {

    template <typename Key, typename T,
            typename HashCompare = tbb_hash_compare<Key>,
            typename Allocator = tbb_allocator<std::pair<const Key, T>>,
            typename Mutex = spin_rw_mutex>
    class concurrent_hash_map {
        using mutex_type = Mutex;
    };

} // namespace tbb
} // namespace oneapi
```

Type requirements

The type of the mutex passed as a template argument for `concurrent_hash_map` should meet the requirements of ReaderWriterMutex. It should also provide the following API:

bool`ReaderWriterMutex::scoped_lock::`is_writer()*const*;

Returns: `true` if the `scoped_lock` object acquires the mutex as a writer, `false` otherwise.

The behavior is undefined if the `scoped_lock` object does not acquire the mutex.

`oneapi::tbb::spin_rw_mutex`, `oneapi::tbb::speculative_spin_rw_mutex`, `oneapi::tbb::queuing_rw_mutex`, `oneapi::tbb::null_rw_mutex`, and `oneapi::tbb::rw_mutex` meet the requirements above.

Example

The example below demonstrates how to wrap `std::shared_mutex` (C++17) to meet the requirements of **ReaderWriterMutex** and how to customize `concurrent_hash_map` to use this mutex.

```
#define TBB_PREVIEW_CONCURRENT_HASH_MAP_EXTENSIONS 1
#include "oneapi/tbb/concurrent_hash_map.h"
#include <shared_mutex>

class SharedMutexWrapper {
public:
    // ReaderWriterMutex requirements

    static constexpr bool is_rw_mutex = true;
    static constexpr bool is_recursive_mutex = false;
    static constexpr bool is_fair_mutex = false;

    class scoped_lock {
    public:
        scoped_lock() : my_mutex_ptr(nullptr), my_writer_flag(false) {}
        scoped_lock(SharedMutexWrapper& mutex, bool write = true)
            : my_mutex_ptr(&mutex), my_writer_flag(write)
        {
            if (my_writer_flag) {
                my_mutex_ptr->my_mutex.lock();
            } else {
                my_mutex_ptr->my_mutex.lock_shared();
            }
        }

        ~scoped_lock() {
            if (my_mutex_ptr) release();
        }

        void acquire(SharedMutexWrapper& mutex, bool write = true) {
            if (my_mutex_ptr) release();

            my_mutex_ptr = &mutex;
            my_writer_flag = write;

            if (my_writer_flag) {
                my_mutex_ptr->my_mutex.lock();
            } else {
                my_mutex_ptr->my_mutex.lock_shared();
            }
        }

        void release() {
            if (my_writer_flag) {
                my_mutex_ptr->my_mutex.unlock();
            } else {
                my_mutex_ptr->my_mutex.unlock_shared();
            }
        }

        bool upgrade_to_writer() {
            // std::shared_mutex does not have the upgrade/downgrade parallel_for_each_semantics
            if (my_writer_flag) return true; // Already a writer

            my_mutex_ptr->my_mutex.unlock_shared();
            my_mutex_ptr->my_mutex.lock();
            return false; // The lock was reacquired
        }

        bool downgrade_to_reader() {
            if (!my_writer_flag) return true; // Already a reader

            my_mutex_ptr->my_mutex.unlock();
            my_mutex_ptr->my_mutex.lock_shared();
            return false;
        }

        bool is_writer() const {
            return my_writer_flag;
        }

    private:
        SharedMutexWrapper* my_mutex_ptr;
        bool                my_writer_flag;
    };
private:
    std::shared_mutex my_mutex;
}; // struct SharedMutexWrapper

int main() {
    using map_type = oneapi::tbb::concurrent_hash_map<int, int,
                                                      oneapi::tbb::tbb_hash_compare<int>,
                                                      oneapi::tbb::tbb_allocator<std::pair<const
int, int>>,
                                                      SharedMutexWrapper>;

    map_type map; // This object will use SharedMutexWrapper for thread safety of insert/find/
erase operations
}
```

Notices and Disclaimers