Threading Building Blocks
Tecnología de la Programación Paralela (TPP)

Pedro Alonso

Máster Universitario en Computación en la Nube y de Altas Prestaciones (MUCNAP)
Depto. de Sistemas Informáticos y Computación
Universitat Politècnica de València

Motivation

- Intel Threading Building Blocks *(TBB)* offers a rich and complete approach to expressing parallelism in a C++ program.
- It is a library that helps you leverage multi-core processor performance without having to be a threading expert.
- Threading Building Blocks relies on templates and the C++ concept of generic programming.
- It does not require special languages or compilers.
- Threading Building Blocks can be used on virtually any processor or any operating system.
- TBB pursues scalability as the number of processor cores increases.

Motivation

- Threading Building Blocks uses templates for common parallel iteration patterns.
- Allows attain increased speed from multiple processor cores without being aware of synchronization, load balancing, and cache optimization.
- Fully supports nested parallelism.
- To use the library, you specify tasks, not threads, and let the library map tasks onto threads in an efficient manner.

Presentation based on the book:

Intel Threading Building Blocks.
Outfitting C++ for Multi-Core Processor Parallelism,
James Reinders.

Benefits of TBB

- *Threading Building Blocks enables to specify tasks instead of threads.*
  - Other programming tools (e.g., OpenMP) require to create, join, and manage threads.
  - Direct programming with threads forces to do the work to efficiently map logical tasks onto threads.
  - TBB runtime library automatically schedules tasks onto threads in a way that makes efficient use of processor resources.
  - Avoiding programming in threads allows better portability, easier programming, more understandable source code, and better performance and scalability.

Benefits of TBB

- *Threading Building Blocks targets threading for performance.*
  - General-purpose packages tend to be low-level tools that provide a foundation, not a solution.
  - Threading Building Blocks focuses on the particular goal of parallelizing computationally intensive work, delivering higher-level, simpler solutions.
- *Threading Building Blocks is compatible with other threading packages.*
  - It does not force to pick among Threading Building Blocks, OpenMP, or raw threads.

Benefits of TBB

- *Threading Building Blocks emphasizes scalable, data-parallel programming*.
  - Dividing the program in functional blocks and assigning them to threads usually does not scale well because typically the number of blocks is fixed.
  - Data-parallel programming scales well to larger numbers of processors by dividing a data set into smaller pieces.
- *Threading Building Blocks relies on generic programming*.
  - Threading Building Blocks uses generic programming: write the best possible algorithms with the fewest constraints.
  - In the C++ Standard Template Library (STL) the interfaces are specified by requirements on types.

Threading Building Blocks vs. OpenMP

- In OpenMP the programmer has to choose among three scheduling approaches for scheduling loop iterations.
- Threading Building Blocks does not require the programmer to worry about scheduling policies.
  - TBB uses single, automatic, and divide-and-conquer approach to scheduling.
  - It is implemented with work stealing: similar to `dynamic` or `guided` but decentralized.
  - *Divide-and-conquer* technique fits well with nested parallelism.
- Parallelism structures in TBB are not limited to built-in types, OpenMP allows reductions only in built-in types.
- Threading Building Blocks implements a subtle but critical recursive model of task-based parallelism and generic algorithms.

Important concepts

- *Recursive Splitting*: The idea is breaking problems up recursively as required to get to the right level of parallel tasks.
  - This works much better than the more obvious static division of work.
- *Task Stealing*: This is a critical design decision that avoids using a global resource as important as a task queue, which would limit scalability.
- *Algorithms*: Algorithm structures are already implemented so the programmer only needs to use them easily, even combination is very easy.

Decomposition

Learning to decompose your problem into concurrent tasks. At the highest level parallelism exists into two ways:

Data Parallelism: Take lots of data and apply the same transformation to each piece of the data.

Task Parallelism: Task parallelism means lots of different, independent tasks that are linked by sharing the data they consume.

Pipeline: Task and Data Parallelism Together. Many independent tasks need to be applied to a stream of data. Each item is processed by stages as they pass through.

Patterns (book *Patterns for Parallel Programming*, by Mattson et al.)

Design spaces:

Finding Concurrency: Threading Building Blocks simplifies finding concurrency by encouraging you to find one or more tasks without worrying about mapping them to hardware threads. The key for you is to express parallelism in a way that allows Threading Building Blocks to create many tasks.

Algorithm structures: It is the high-level strategy. TBB manages easily algorithm structures like *Pipeline*, *Task Parallelism*, *Divide and Conquer*, and *Recursive Data*.

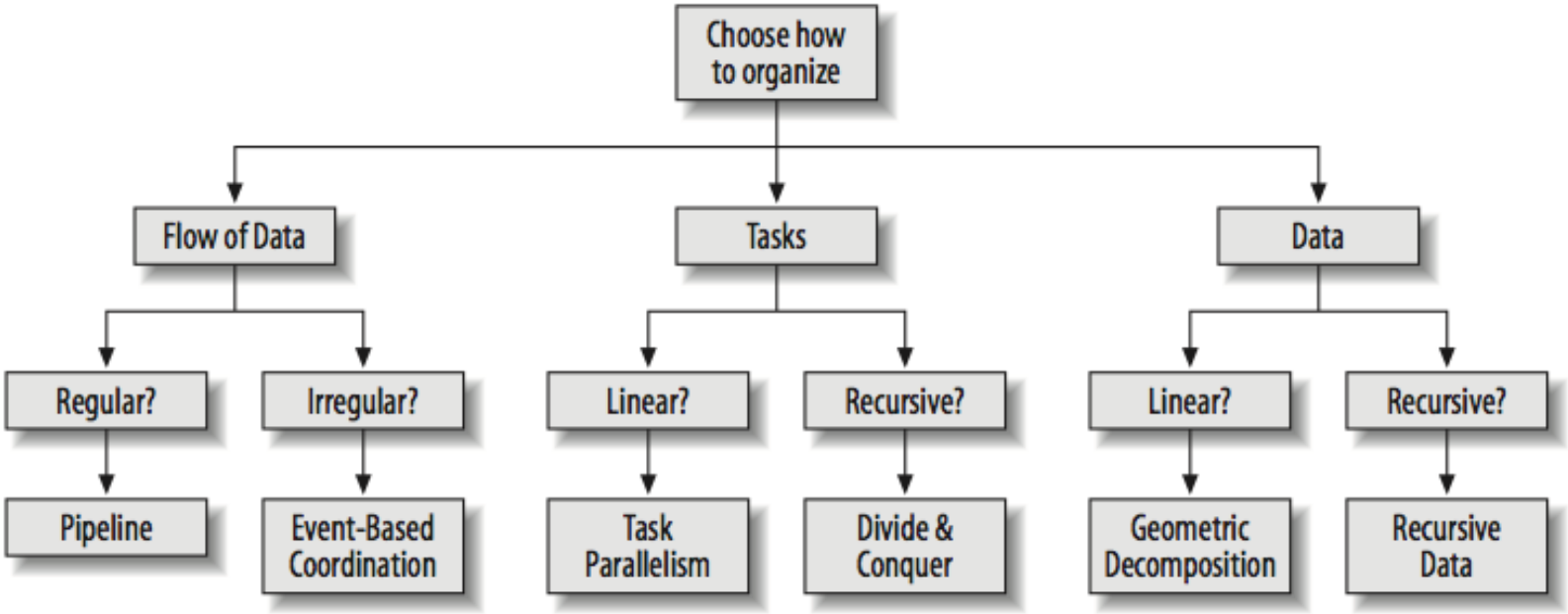
Supporting structures: This step involves the details for turning our algorithm strategy into actual code. TBB offers enough support.

Implementation mechanisms: This step includes thread management and synchronization. Threading Building Blocks does everything.

Patterns

Design Space	Key	Owner of the job
Finding concurrency	Think Parallel	programmer
Algorithm structures	Designed algorithms	programmer uses templates
Supporting structures	Designed algorithms	TBB
Implementation mechanisms	Write code to avoid the need for explicit synchronization	TBB

Organization of the work



Initializing and Terminating the Library

- Intel Threading Building Blocks components are defined in the `tbb` namespace.
- `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.

Basic Algorithms

Basic algorithms for loop parallelization:

- The most visible contribution of Threading Building Blocks is the *algorithm templates*.
- Threading Building Blocks offers the following types of generic parallel algorithms:
  - `parallel_for`, `parallel_reduce` and `parallel_for_each`
    Load-balanced, parallel execution of a fixed number of independent loop iterations
  - `parallel_scan`
    A template function that computes a prefix computation (also known as a *scan*) in parallel (`y[i] = y[i-1] op x[i]`)

parallel_for

Sequential code to parallelize

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

- The *iteration space* is of type `size_t` and goes from 0 to `n-1`.
- The template function `tbb::parallel_for` breaks this iteration space into *chunks* and runs each chunk on a separate thread.
- The first step in parallelizing this loop is to convert the loop body into a form that operates on a chunk.

The body object

The form is a Standard Template Library (STL)-style function object, called the *body object*, in which `operator()` processes a *chunk*.

parallel_for

Class body for the object body

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

- A `blocked_range<T>` is a template class provided by the library. It describes a one-dimensional iteration space over type `T`.
- An instance of the body class needs member fields that remember all the local variables that were defined outside the original loop but were used inside it. Usually, the constructor will initialize these fields.
- The template function `parallel_for` requires that the body object have a copy constructor, which is invoked to create a separate copy (or copies) for each worker thread.

parallel_for

Class body for the object body

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

- Because the body object might be copied, its `operator()` should not modify the body. The method must be declared as `const`.
- The `operator()` function should not modify data member `my_a`. It can modify what `my_a` points to.
- This distinction is emphasized by declaring `my_a` as `const` and what it points to as (non-const) float.
- Otherwise, the modification might or might not become visible to the thread that invoked `parallel_for`, depending upon whether `operator()` is acting on the original or a copy.

parallel_for

- After written the *loop body* as a *body object*, invoke the template function `parallel_for`

Calling the parallel_for algorithm

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

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

- The function `parallel_for` receives two arguments:
  1. A range of type `blocked_range`.
  2. An instance of the class body.
- Function `parallel_for` is overloaded and can have more arguments.

parallel_for

- After written the *loop body* as a *body object*, invoke the template function `parallel_for`

Calling the parallel_for algorithm

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

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

`blocked_range` is a class representing the entire iteration space from 0 to `n-1`, which is divided 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 as a half-open interval [`begin`,`end`).
- `GrainSize` suggests the smallest chunk size of the iteration space.

parallel_for

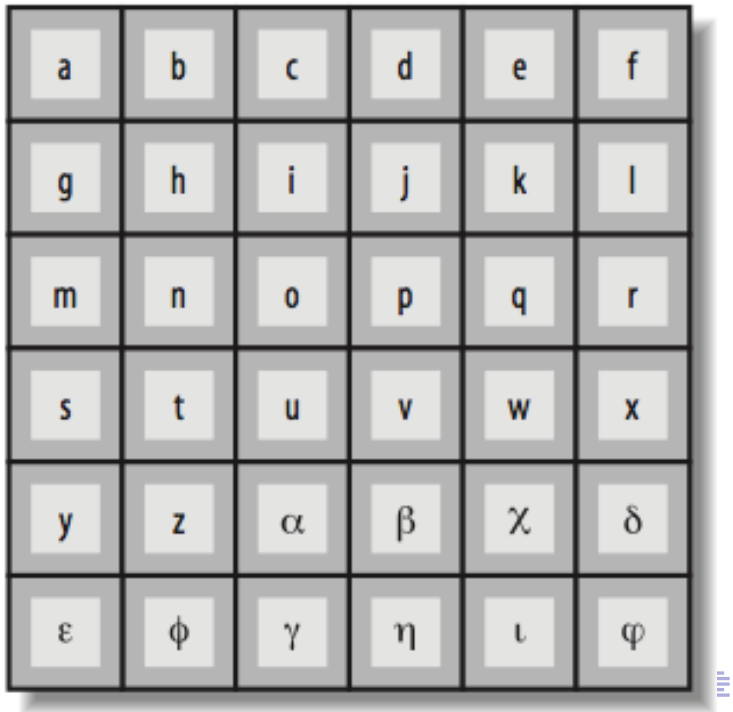
Example: Average

```
#include "tbb/parallel_for.h"
#include "tbb/blocked_range.h"
using namespace tbb;

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

void ParallelAverage( float* output, float* input, size_t n ) {
  Average avg;
  avg.input = input;
  avg.output = output;
  parallel_for( blocked_range<int>( 0, n, 1000 ), avg );
}
```

parallel_for

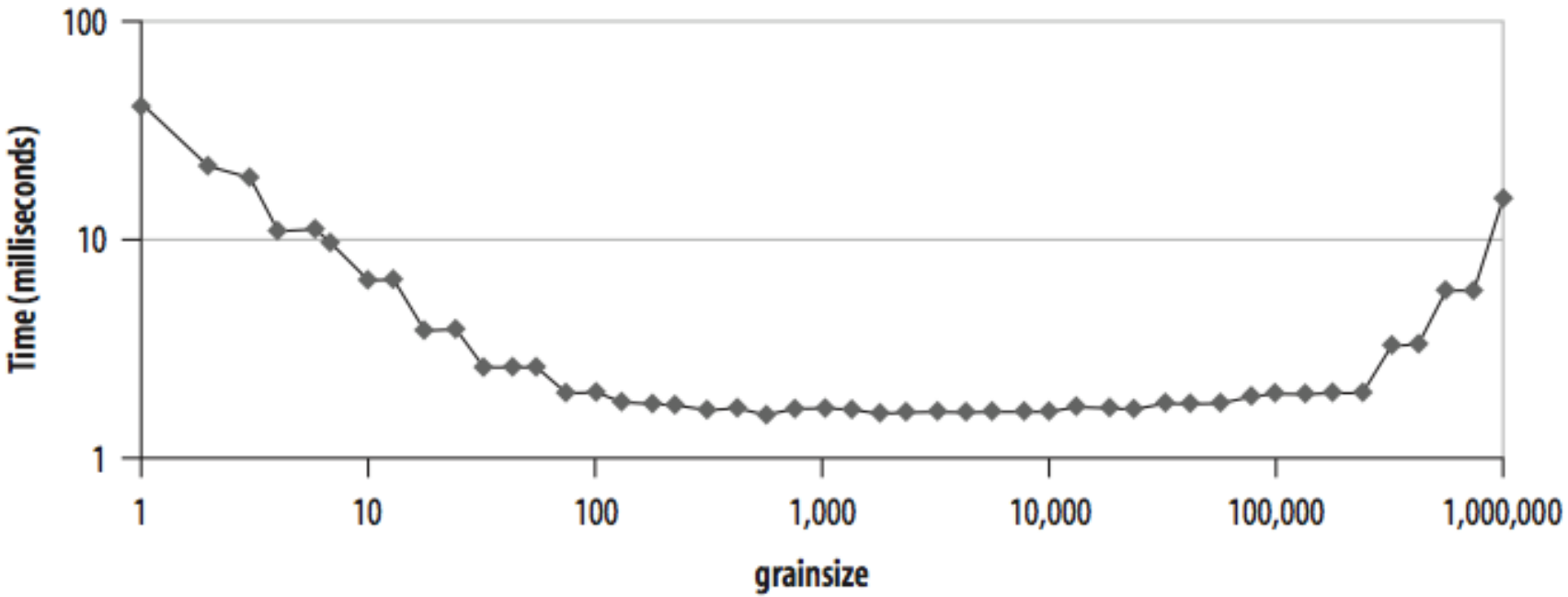
GrainSize parameter

- Implicitly specifies the number of iterations for a “reasonable” size chunk (GrainSize) to deal out to a processor.
- If the iteration space has more than GrainSize iterations, parallel_for splits it into separate subranges.
- The GrainSize amortizes parallel scheduling overhead.
- The GrainSize sets a minimum threshold for parallelization.



parallel_for

Typical execution behaviour of the `GrainSize` parameter (computation of the floating-point operation `a[i]=b[i]*c` computation over 1 million indices).



parallel_for

Automatic grain size:

- When `GrainSize` is not specified, a *partitioner* should be supplied to the algorithm template.
- If both the *partitioner* and the `GrainSize` are omitted, is like `GrainSize` = 1.
- A *partitioner* is an object that guides the chunking of a range.
- In particular, class `auto_partitioner` heuristically chooses the grain size trying to limit overhead while still providing load balancing. This is the default partitioner.
- Example:

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

parallel_for

Example: Average with simple_partitioner

```
#include "tbb/parallel_for.h"
#include "tbb/blocked_range.h"
using namespace tbb;

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

void ParallelAverage( float* output, float* input, size_t n ) {
  Average avg;
  avg.input = input;
  avg.output = output;
  parallel_for( blocked_range<int>( 0, n, 1000 ), avg );
}
```

parallel_for

Example: Average with simple_partitioner

```
#include "tbb/parallel_for.h"
#include "tbb/blocked_range.h"
using namespace tbb;

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

void ParallelAverage( float* output, float* input, size_t n ) {
  Average avg;
  avg.input = input;
  avg.output = output;
  parallel_for( blocked_range<int>( 0, n ), avg, simple_partitioner());
}
```

parallel_for: The two dimensional interval algorithm blocked_range2d

- Class template that represents a recursively divisible two-dimensional half-open interval.
- A blocked_range2d represents a half-open two-dimensional range [i0,j0)×[i1,j1).
- Each axis of the range has its own splitting threshold.
- A blocked_range2d is divisible if either axis is divisible.
- Function rows() (cols()) returns a range of the first (second) dimension.
- Function begin() (end()) over a one-dimensional range r returns the first (one-plus-last) element of that range r.

parallel_for: The two dimensional interval algorithm blocked_range2d

Example: Two nested loops, sequential

```
for( int i=0; i<n; i++ ) {
  for( int j=0; j<i; j++ ) {
    do_some_work();
  }
}
```

Example: Two nested loops, parallel

```
parallel_for( blocked_range2d<int>(0,n,0,n), [=]( blocked_range2d<int> &r ) {
   for ( int i = r.rows().begin(); i < r.rows().end(); i++ ) {
     for ( int j = r.cols().begin(); j < r.cols().end(); j++ ) {
       if( i >= j )
         do_some_work();
     }
   }
 });
```

parallel_reduce

Applying a function such as sum, max, min, or logical AND across all the members of a group is called a *reduction operation*.
Example:

Serial sum

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

The TBB solution for this is to employ the algorithm parallel_reduce

parallel_reduce

Parallel sum: the body class

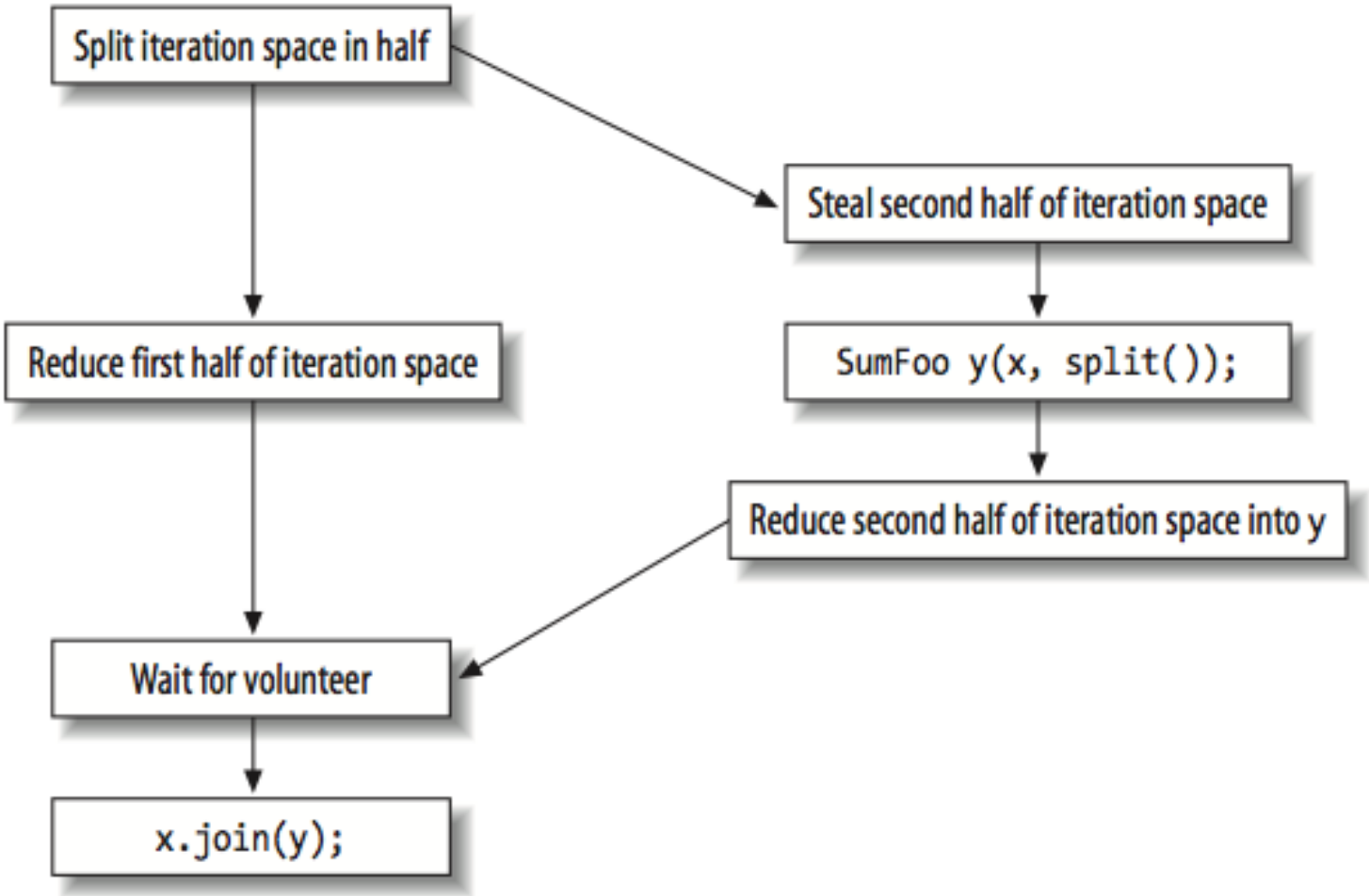
```
class SumFoo {
  float* my_a;
  public:
    float my_sum;
    void operator()( const blocked_range<size_t>& r ) {
      float *a = my_a;
      float sum = my_sum;
      size_t end = r.end();
      for( size_t i=r.begin(); i!=end; ++i )
          sum += Foo(a[i]);
      my_sum = sum;
    }

    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) { }
};
```

parallel_reduce

- Thread-private copies of the *body* must be merged at the end, and therefore the `operator()` is not `const`.
- The *splitting constructor*:
  - Arguments: a reference to the original object, and has a *dummy* argument of type `split`.
  - The dummy argument serves simply by its presence to distinguish the splitting constructor from a *copy constructor*.
- The method `join`:
  - invoked whenever a task finishes its work and needs to merge the result back with the main body of work.
  - The parameter passed to the method is the result of the work. The method repeats the same operation performed in each task on each element.

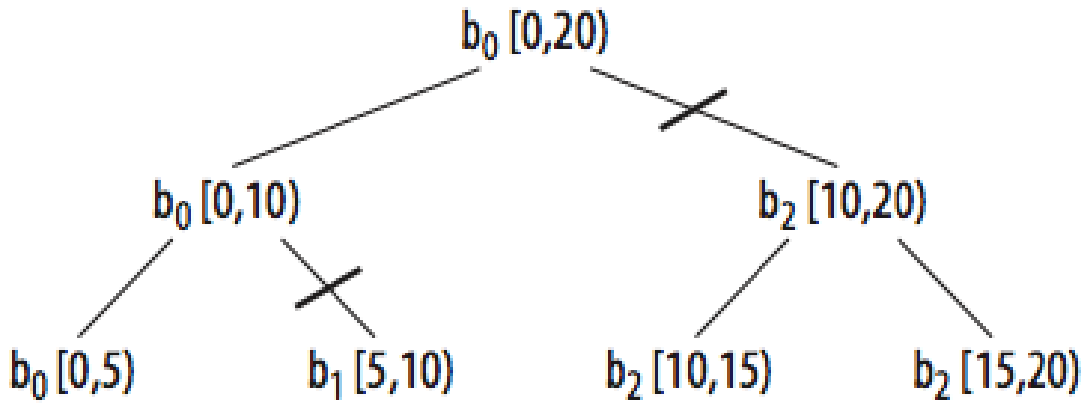
parallel_reduce

Split-join sequence of the reduce operation



parallel_reduce

- The range is recursively split at each level into two subranges.
- In the right branches the body is created by the body splitting constructor.
- On the way back up the tree, `parallel_reduce` invokes `b0.join(b1)` and `b0.join(b2)` to merge the results of the leaves.
- There exist other valid different executions.
- Left-to-right property: A given body always evaluates one or more consecutive subranges in left to right order.



parallel_reduce over blocked_range<int>(0,20,5)

parallel_reduce

Parallel sum: the parallel reduction

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

- parallel_reduce generalizes to any associative operation.
- The splitting constructor does two things:
  - Copies read-only information necessary to run the loop body.
  - Initializes the reduction variables to the identity element of the operations.
- The join method should do the corresponding merges.
- You can do more than one reduction at the same time: for instance, you can gather the min and max with a single parallel_reduce.

parallel_reduce

Example: Find the smallest element

```
long SerialMinIndexFoo( const float a[], size_t n ) {
  float value_of_min = FLT_MAX;
  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;
}
```

parallel_reduce

Example: Parallel find the smallest element

```
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( const float a[] ) :
      my_a(a),
      value_of_min(FLT_MAX),
      index_of_min(-1), { }
    . . .
```

parallel_reduce

Example: Parallel find the smallest element

```
    . . .
    MinIndexFoo( MinIndexFoo& x, split ) :
      my_a(x.my_a),
      value_of_min(FLT_MAX),
      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;
      }
    }
  });
```

parallel_reduce

Example: Parallel find the smallest element

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

- The way to implement the former algorithm is called *imperative*.
- There exists another way called *functional* which is thought to be used with *lambda* expressions.

parallel_reduce

- The function template `parallel_reduce` has two forms:
  - The imperative form is designed to minimize copying of data.
  - The functional form is designed to be easy to use in conjunction with lambda expressions.
- The functional form

```
parallel_reduce( range, identity, func, reduction )
```

  - performs a parallel reduction by applying `func` to subranges in `range` and reducing the results with the binary operator `reduction`.
  - It returns the result of the reduction.
  - The identity parameter specifies the left identity element for func's `operator()`.
  - Parameters `func` and `reduction` can be lambda expressions.

parallel_reduce

Example: The following code sums the values in an array.

```
float ParallelSum( float array[], size_t n ) {
  return parallel_reduce(
    blocked_range<float*>( array, array+n ),
    0.f,
    [](const blocked_range<float*>& r, float init)->float {
        for( float* a=r.begin(); a!=r.end(); ++a )
            init += *a;
        return init;
    },
    []( float x, float y )->float {
        return x+y;
    }
  );
}
```

Advanced Algorithms: parallel_for_each

- When the end of the iteration space is not known in advance, or
- when the loop body may add more iterations to do before the loop exits.
- A linked list is an example of an iteration space that is not known in advance.

Serial list processing code

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

- A stream of items.
- The loop body.

Advanced Algorithms: parallel_for_each

- 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`.

Functor

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

Advanced Algorithms: parallel_for_each

Parallel form

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

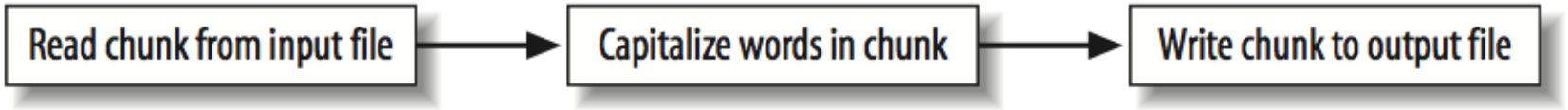
- An invocation of `parallel_for_each` never causes two threads to act on an input iterator concurrently.
- Thus typical definitions of input iterators for sequential programs work correctly.
- This convenience makes `parallel_for_each` unscalable, because the fetching of work is serial. But in many situations, you still get useful speedup over doing things sequentially.

Advanced Algorithms: parallel_for_each

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.

Advanced Algorithms: parallel_pipeline



Pipeline example

The pipeline consists of:

- The class `filter`: Represents each stage of the pipeline.
- The algorithm `parallel_pipeline`: A pipeline represents the pipelined application of a series of filters to a stream of items. The function `parallel_pipeline` gathers one filter composed by all the objects of type `filter` concatenated into one object of type `filter` to build the pipeline.

Advanced Algorithms: parallel_pipeline

`filter<InputType, OutputType>`: Represents each stage of the pipeline.

- A `filter` class template represents a strongly-typed filter in a `parallel_pipeline` algorithm, with its template parameters specifying the filter input and output types.
- A filter can be constructed from a functor or by composing two filter objects with `operator&()`. The same filter object can be reused in multiple `&` expressions.
- The `filter` class should only be used in conjunction with `parallel_pipeline` functions.
- A `filter_mode` enumeration represents an execution mode of a filter in a `parallel_pipeline` algorithm.
  - A `parallel` filter can process multiple items in parallel and without a particular order.
  - A `serial_out_of_order` filter processes items one at a time and without a particular order.
  - A `serial_in_order` filter processes items one at a time. The order in which items are processed is implicitly set by the first `serial_in_order` filter and respected by all other such filters in the pipeline.

Advanced Algorithms: parallel_pipeline

Class filter

```
template<typename InputType, typename OutputType>
      class filter { ... }
```

Funciton parallel_pipeline

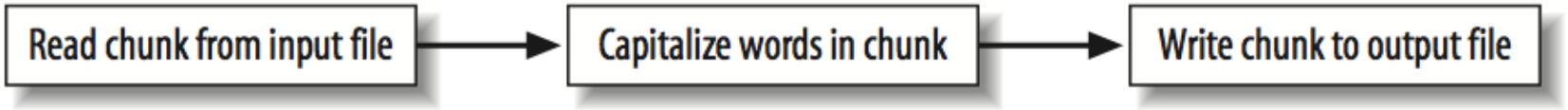
```
parallel_pipeline( max_number_of_live_tokens,
     make_filter<void,I1>(mode0,g0) &
     make_filter<I1,I2>(mode1,g1) &
     make_filter<I2,I3>(mode2,g2) &
     ...
     make_filter<In,void>(moden,gn) );
```

Funciton make_filter

```
filter<T, U> make_filter(filter::mode mode, const Func &f)
    Returns filter<T, U>(mode, f).
```

Advanced Algorithms: parallel_pipeline

Example:



- The file I/O is sequential.
- The capitalization stage can be done in parallel.
- The chunks (streams of characters) are written in the proper order to the output file.
- To decide whether to capitalize a letter, inspect whether the preceding character is a blank. The solution is to have each chunk also store the last character of the preceding chunk, i.e., the chunks overlap by one character.

Advanced Algorithms: parallel_pipeline

Buffer for blocks of characters

```
class MyBuffer {
  private:
    static const size_t buffer_size = 10000;
    char* my_end;
    // storage[0] holds the last character of the prec. buffer
    char storage[1+buffer_size];
  public:
    // Pointer to first character in the buffer
    char* begin() {return storage+1;}
    const char* begin() const {return storage+1;}
    // Pointer to one past last character in the buffer
    char* end() const {return my_end;}
    // Set end of buffer.
    void set_end( char* new_ptr ) {my_end=new_ptr;}
    // Number of bytes a buffer can hold
    size_t max_size() const {return buffer_size;}
    // Number of bytes in buffer.
    size_t size() const {return my_end-begin();}
  };
```

Advanced Algorithms: parallel_pipeline

Input stage of the pipeline

```
auto f1 = [&]( flow_control& fc ) -> MyBuffer* {
  MyBuffer& b = buffer[next_buffer];
  next_buffer = (next_buffer+1) % n_buffer;
  size_t n = fread( b.begin(), 1, b.max_size(), input_file );
  if( !n ) { // end of file
    fc.stop();
    return NULL;
  } else {
    b.begin()[-1] = last_char_of_previous_buffer;
    last_char_of_previous_buffer = b.begin()[n-1];
    b.set_end( b.begin()+n );
    return &b;
  }
};
```

Advanced Algorithms: parallel_pipeline

Middle stage of the pipeline

```
auto f2 = [=]( MyBuffer* b ) -> MyBuffer* {
  bool prev_char_is_space = b->begin()[-1]==' ';
  for( char* s=b->begin(); s!=b->end(); ++s ) {
    if( prev_char_is_space && islower(*s) )
      *s = toupper(*s);
    prev_char_is_space = isspace(*s);
  }
  return b;
};
```

Output stage for the pipeline

```
auto f3 = [=]( MyBuffer* b ) -> void* {
  fwrite( b->begin(), 1, b->size(), output_file );
  return NULL;
};
```

Advanced Algorithms: parallel_pipeline

The main code for the pipeline

```
parallel_pipeline( /*max_number_of_live_token=*/ 2,
  make_filter<void,MyBuffer*>( filter_mode::serial_in_order, f1 ) &
  make_filter<MyBuffer*,MyBuffer*>( filter_mode::parallel, f2 ) &
  make_filter<MyBuffer*,void>( filter_mode::serial_in_order, f3 )
);
```

- Objects are passed between filters using pointers to avoid the overhead of copying a `MyBuffer`.
- The maximum number of tokens that can be in flight is **2** (first parameter).
- The second parameter specifies the sequence of filters. Each filter is constructed by function:

```
make_filter< inputType, outputType >( mode, functor ).
```

Advanced Algorithms: parallel_pipeline

```
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.

Advanced Algorithms: parallel_pipeline

The main code for the pipeline

```
parallel_pipeline( /*max_number_of_live_token=*/ 2,
  make_filter<void,MyBuffer*>( filter_mode::serial_in_order, f1 ) &
  make_filter<MyBuffer*,MyBuffer*>( filter_mode::parallel, f2 ) &
  make_filter<MyBuffer*,void>( filter_mode::serial_in_order, f3 )
);
```

- 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.

Advanced Algorithms: parallel_pipeline

The main code for the pipeline

```
parallel_pipeline( /*max_number_of_live_token=*/ 2,
  make_filter<void,MyBuffer*>( filter_mode::serial_in_order, f1 ) &
  make_filter<MyBuffer*,MyBuffer*>( filter_mode::parallel, f2 ) &
  make_filter<MyBuffer*,void>( filter_mode::serial_in_order, f3 )
);
```

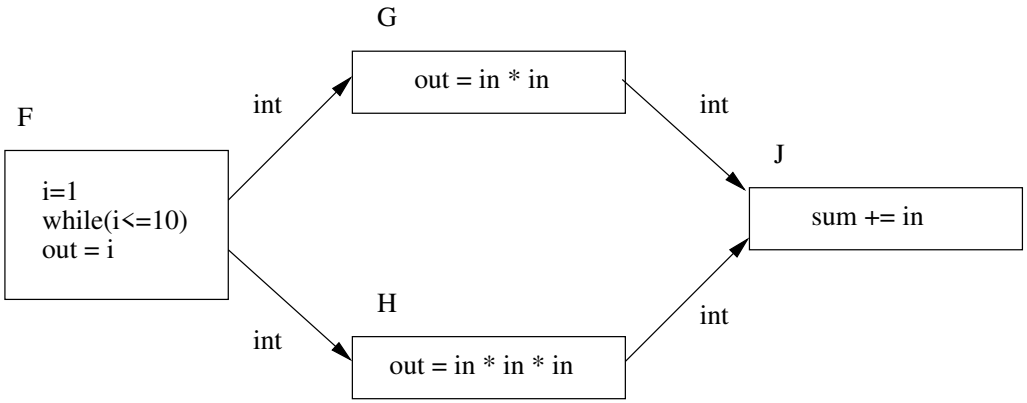
- The filters are concatenated with operator `&` to one filter of type `filter(void,void)` with `filter::operator&()`.
- When concatenating two filters, the `outputType` of the first filter must match the `inputType` of the second filter.

Graph Parallelism: `flow_graph`

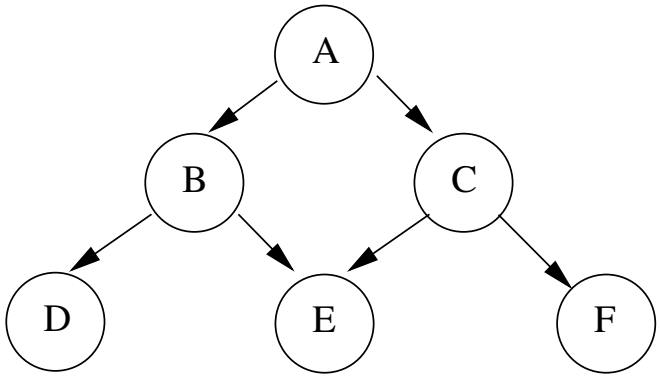
- Threading Building Blocks (oneTBB) library also supports graph parallelism.
- Using graph parallelism, computations are represented by nodes and the communication channels between these computations are represented by edges.
- When a node in the graph receives a message, a task is spawned to execute its body object on the incoming message.
- Messages flow through the graph across the edges that connect the nodes.

Graph Parallelism: `flow_graph`

Simple Data Flow Graph



Dependence Graph



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

Graph Parallelism: Graph object

- 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.
- The code below creates a graph object and then waits for all tasks spawned by the graph to complete.

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

Graph Parallelism: Nodes

- A node is a class that inherits from `graph_node` and also typically inherits from `sender<T>`, `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.
- It is typical to use predefined node types to construct a graph.
- For instance, a `function_node` is a predefined type that represents a simple function with one input and one output. The constructor has three arguments:

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

Graph Parallelism: `function_node`

```
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.

Graph Parallelism: `function_node`

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

After the node is constructed it can be passed messages to it, either by connecting it to other nodes using edges or by invoking its function `try_put`.

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

Now it can wait for the messages to be processed by calling `wait_for_all` on the graph object:

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

Graph Parallelism: `function_node`

Important remarks:

- 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.
- Constructing the node with a different concurrency limit, parallelism can be achieved:

```
graph g;
function_node< int, int > n( g, unlimited, []( int v ) -> int {
    ... some code ...
    return v;
} );
```

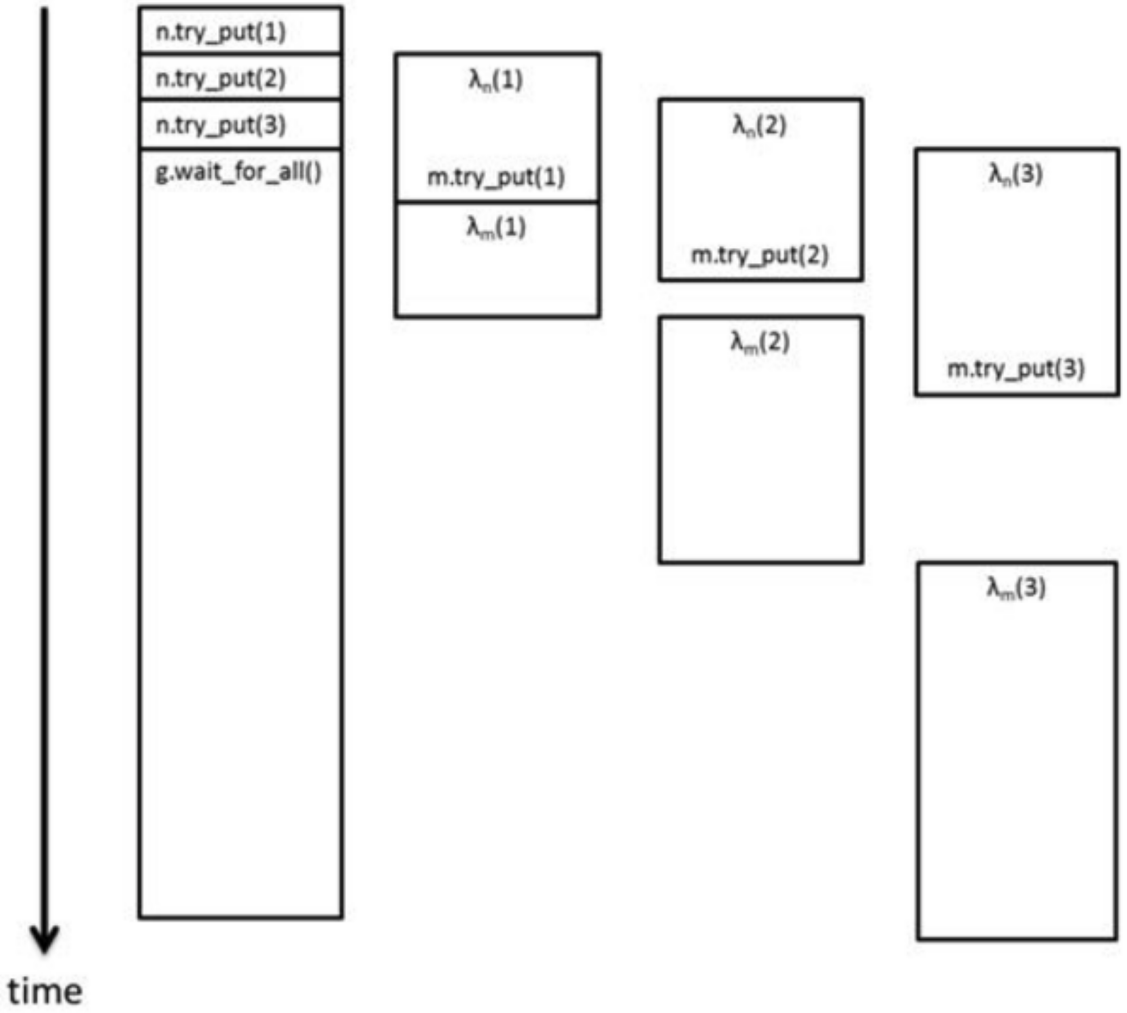
Graph Parallelism: Edges

- 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.

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

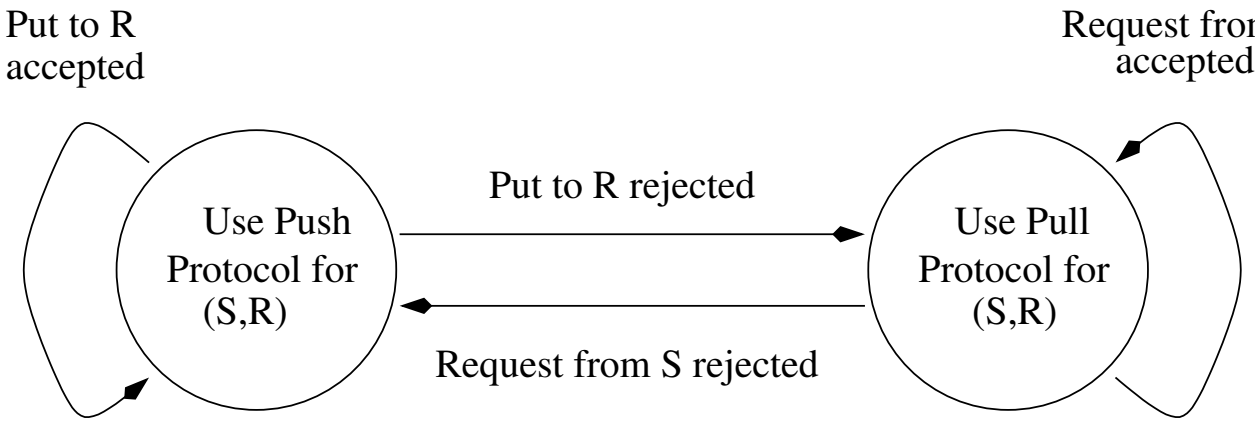
Graph Parallelism: Mapping Nodes to Tasks

- Execution timeline of the two node Graph.
- The bodies of `n` and `m` will be referred to as λ_n and λ_m, respectively.



Graph Parallelism: Message Passing Protocol

- 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.
- This protocol allows for avoiding 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.
  - If a predecessor has a message, the successor will process it and the edge will remain in pull mode.
  - If the predecessor has no message, the edge between the nodes will switch from pull to push mode.



Graph Parallelism: Single-push vs. Broadcast-push

- Nodes 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.

- For instance, `buffer_node` has a "single-push" policy for forwarding messages. Putting X messages to the `buffer_node` results in X messages being pushed. In general there is no policy for which node is pushed to if more than one successor can accept.
- A `broadcast_node` pushes any message it receives to all accepting successors. Putting X messages to the `broadcast_node` results in a total of $X \times Y$ messages pushed to the Y `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.

Graph Parallelism: Buffering and Forwarding

- 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.

Graph Parallelism: Application Categories

Most flow graphs fall into one of two categories:

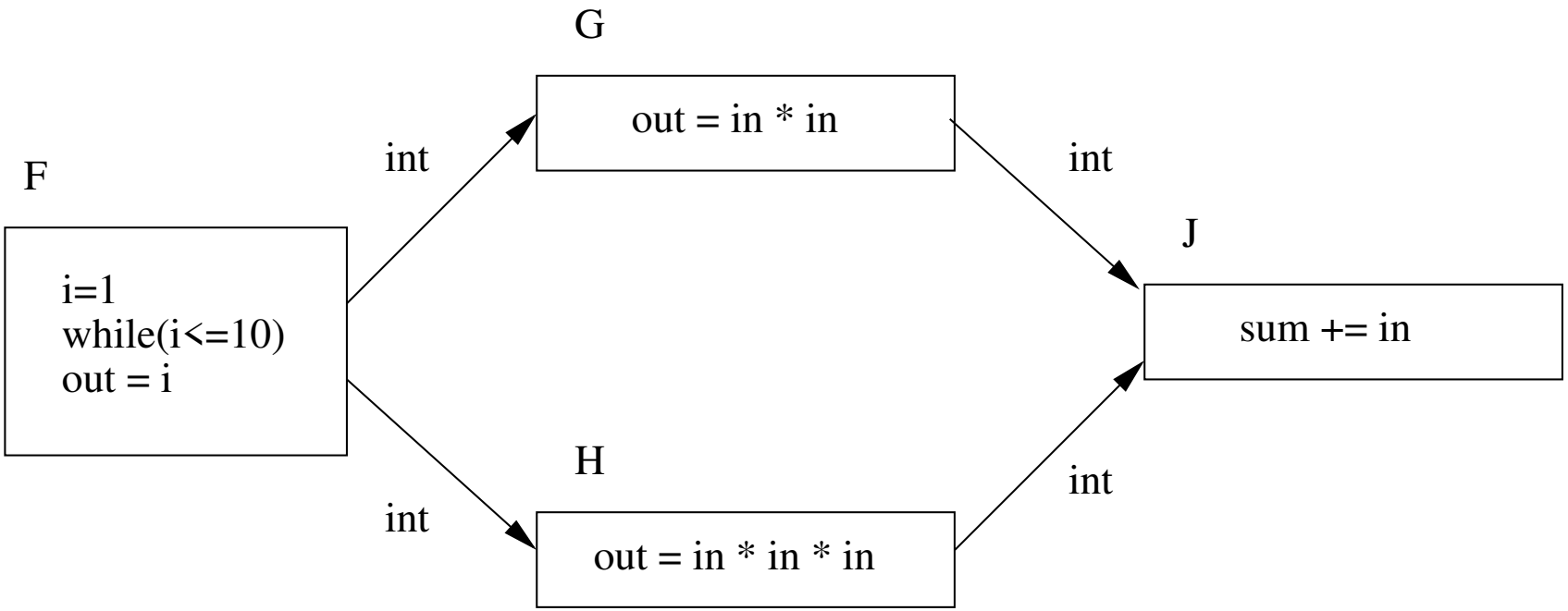
Data flow graphs. In this type of graph, data is passed along the graph's edges. The nodes receive, transform and then pass along the data messages.

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.

Graph Parallelism: Data flow graphs

- In this type of 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.

Example of Data Flow Graph:



Graph Parallelism: Data flow graphs

```
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 << "La suma es " << sum << "\n";
```

Graph Parallelism: Data flow graphs

- 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`. 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();
```

Graph Parallelism: Data flow graphs

- A better option is to introduce an `input_node` in the graph,

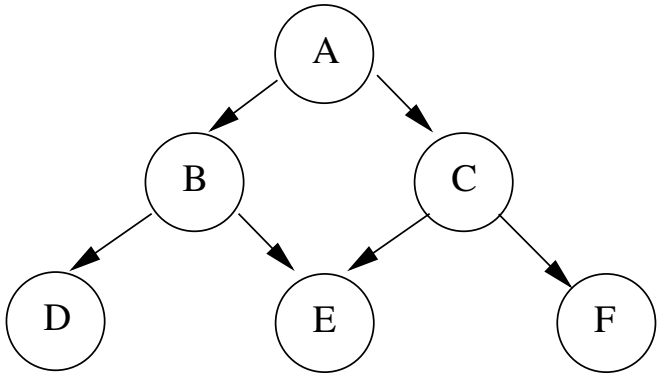
```
input_node< int > src( g, src_body(10) );
make_edge( src, squarer );
make_edge( src, cuber );
src.activate();
g.wait_for_all();
```

- where the body object si implemented as:

```
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()( flow_control& fc ) {
        if ( my_next_value <= my_limit ) {
            return my_next_value++;
        } else {
            fc.stop();
            return int();
        }
    }
};
```

Graph Parallelism: Dependency graphs

- 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.
- Dependence graphs are a special case of data flow graphs, where the data passed between nodes are of type `continue_msg`.
- 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.



- To implement this as a flow graph, `continue_node` objects are used for the nodes and `continue_msg` objects as the messages.

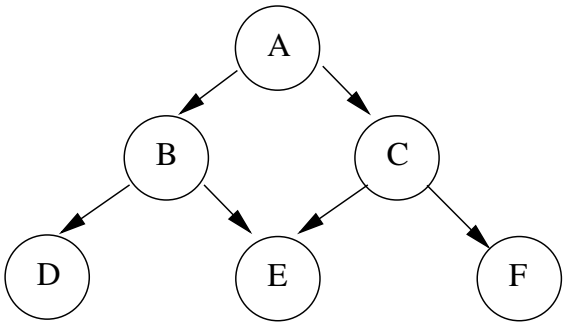
```
template< typename Body > continue_node( graph &g, Body body)
```

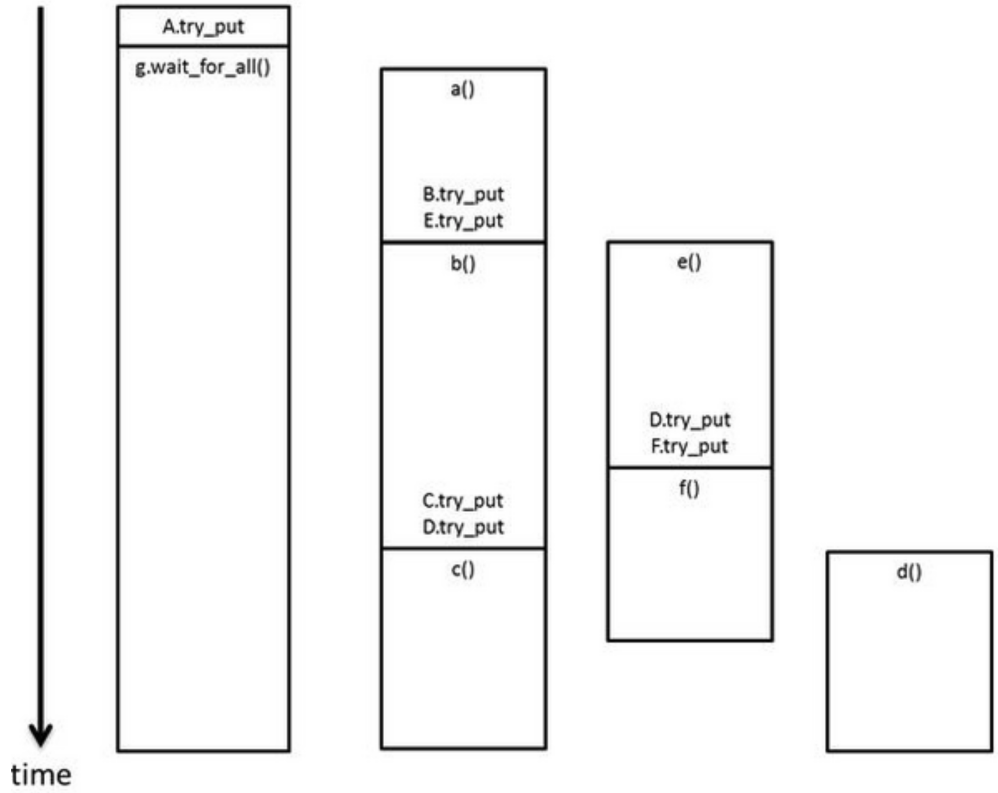
Graph Parallelism: Dependency graphs

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

int main() {
  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;
}
```

Graph Parallelism: Dependency graphs

- Execution timeline of the Dependency Graph.





- 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. Only the call to `wait_for_all` blocks.

Graph Parallelism: Predefined Node Types

`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 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`, `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.

Task Algorithms

- *oneAPI Threading Building Blocks (oneTBB)* provides a task scheduler, which is the engine that drives the algorithm templates and task groups. The exact tasking API depends on the implementation.
- The tasks are quanta of computation.
  - The scheduler implements worker thread pool and maps tasks onto these threads.
  - The mapping is non-preemptive. Once a thread starts running a task, the task is bound to that thread until completion. During that time, the thread services other tasks only when it waits for completion of nested parallel constructs. While waiting, either user or worker thread may run any available task, including unrelated tasks created by this or other threads.
- The task scheduler is intended for parallelizing computationally intensive work. Because task objects are not scheduled preemptively, they should generally avoid making calls that might block a thread for long periods during which the thread cannot service other tasks.

Task Algorithms

- Intel Threading Building Blocks provides low level algorithms to access concurrency in a more simple way.
- This low level programming is useful, e.g. for *recursive algorithms*.
- We'll see the next two tools in this context:
  - The `parallel_invoke` algorithm, and
  - The `taskgroup` class.

Fibonacci numbers

```
size_t fib(size_t n) {
   size_t i, j;
   if (n<2)
     return n;
   else {
         i=fib(n-1);
         j=fib(n-2);
         return i+j;
   }
}
```

The parallel_invoke algorithm

- Template function that evaluates several functions in parallel.
- Header: `#include "tbb/parallel_invoke.h"`.
- The expression `parallel_invoke(f0,f1,...,fk)` evaluates `f0()`, `f1()`, ..., `fk()` possibly in parallel.
- Each argument must have a type for which `operator()` is defined.
- Typically the arguments are either function objects or pointers to functions.
- Return values are ignored.

The parallel_invoke algorithm

Example

```
void f();
extern void bar(int);

class MyFunctor {
    int arg;
public:
    MyFunctor(int a) : arg(a) {}
    void operator()() const {bar(arg);}
};

void RunFunctionsInParallel() {
    MyFunctor g(2);
    MyFunctor h(3);
    tbb::parallel_invoke(f, g, h );
}
```

The parallel_invoke algorithm

Example: Fibonacci numbers (with a functor v. 1)

```
class FibClass {
  size_t n, &m;
  public:
    FibClass( size_t n, size_t& m ) : n(n), m(m) { }
    void operator()() const {
      fib( n, m );
    }
};
void fib(size_t n, size_t& m ) {
   size_t i, j;
   if (n<2) m = n;
   else {
         FibClass f1( n-1, i );
         FibClass f2( n-2, j );
         parallel_invoke( f1, f2 );
         m = i+j;
   }
}
```

The parallel_invoke algorithm

Example: Fibonacci numbers (with a functor v. 2)

```
class FibClass {
  size_t n, &m;
  public:
    FibClass( size_t n, size_t& m ) : n(n), m(m) { }
    void operator()() const {
      fib( n, m );
    }
};
void fib(size_t n, size_t& m ) {
   size_t i, j;
   if (n<2) m = n;
   else {
        parallel_invoke( FibClass( n-1, i ),
                         FibClass( n-2, j ) );
        m = i+j;
   }
}
```

The parallel_invoke algorithm

Example: Fibonacci numbers (with a functor v. 3)

```
class FibClass {
  size_t n, &m;
  public:
    FibClass( size_t n, size_t& m ) : n(n), m(m) { }
    void operator()() const {
      fib( n, m );
    }
};
void fib(size_t n, size_t& m ) {
   size_t i, j;
   if (n<2) m = n;
   else {
    parallel_invoke( *(new FibClass( n-1, i )),
                     *(new FibClass( n-2, j )) );
         m = i+j;
   }
}
```

The parallel_invoke algorithm

Example: Fibonacci numbers (with a lambda expression)

```
size_t fib(size_t n) {
   size_t i, j;
   if (n<2)
     return n;
   else {
         parallel_invoke( [&]{i=fib(n-1);},
                          [&]{j=fib(n-2);} );
         return i+j;
   }
}
```

The task_group class

- It is a high level interface to the task scheduler.
- A task_group represents concurrent execution of a group of tasks.
- Header: `#include "tbb/task_group.h"`.
- Members:

```
task_group();
~task_group();
template<typename Func>
   void run( const Func& f );
template<typename Func>
   void run( task_handle<Func>& handle );
template<typename Func>
   void run_and_wait( const Func& f );
template<typename Func>
   void run_and_wait( task_handle<Func>& handle );
task_group_status wait();
bool is_canceling();
void cancel();
```

The task_group class

Example: Fibonacci numbers (with a functor)

```
void fib(size_t n, size_t& m ) {
   size_t i, j;
   if (n<2) m = n;
   else {
         task_group g;
         g.run( FibClass( n-1, i ) );
         g.run( FibClass( n-2, j ) );
         g.wait();
         m = i+j;
   }
}
```

The task_group class

Example: Fibonacci numbers (with a lambda expression)

```
size_t fib(size_t n) {
    size_t i, j;
    if (n<2)
      return n;
    else {
          task_group g;
          g.run([&]{i=fib(n-1);});
          g.run([&]{j=fib(n-2);});
          g.wait();
          return i+j;
    }
}
```

Containers

- Intel Threading Building Blocks provides *highly concurrent* containers that permit multiple threads to invoke a method simultaneously on the same container.
  - queue,
  - vector, and
  - hash map.
- The interfaces of the Threading Building Blocks containers are similar to STL, but they do not match completely.
- The Threading Building Blocks containers provide *fine-grained locking* or *lock-free* implementations, and sometimes both.

Containers: concurrent_queue

Template class for the concurrent queue

```
concurrent_queue<T>
```

- Implements a concurrent queue with values of type T.
- Multiple threads may simultaneously push and pop elements from the queue.
- A queue is a *first-in first-out* structure (single-threaded program).
- If multiple threads are pushing and popping concurrently, the definition of "first" is uncertain.
- The only guarantee is: if a thread pushes multiple values, and another thread pops those same values, they will be popped in the same order that they were pushed.

Containers: `concurrent_queue`

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

- `pop_if_present`:
  This method is `nonblocking`: it attempts to pop a value, and if it cannot because the queue is empty, it returns anyway.
- `pop`:
  This method `blocks` until it pops a value. It must be used if a thread must wait for an item to become available and it has nothing else to do.

Containers: `concurrent_queue`

- `concurrent_queue::size_type` is a signed integral type, not unsigned.
- `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.
- `empty()` method is defined to be true if and only if `size()` is not positive.
- By default, a `concurrent_queue<T>` is unbounded.
- Method `set_capacity` can be used to bound the queue. Setting the capacity causes push to block until there is room in the queue.

Containers: concurrent_vector

Template class for the concurrent vector

```
concurrent_vector<T>
```

- Implements a dynamically growable array with values of type `T`.
- Multiple threads may simultaneously access elements of the vector.
- Care must be taken when more than one task access an element that is under construction or is otherwise being modified.
- Two methods for resizing:
  - `grow_by`: enables to safely append `n` consecutive elements to a vector, and returns the index of the first appended element; and
  - `grow_to_at_least`: grows a vector to size `n` if it is shorter.
- The `size()` method returns the number of elements in the vector, which may include elements that are still undergoing concurrent construction.

Containers: concurrent_vector

In this example, it is safely appended a C string to a shared vector.

Example of concurrent_vector

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

Containers: `concurrent_vector`

Whole Vector Operations (These operations are not thread–safe on the same instance):

- `concurrent_vector()`: Constructs an empty vector.
- `concurrent_vector( const concurrent_vector& src )`: Constructs a copy of `src`.
- `concurrent_vector& operator=( const concurrent_vector& src )`: Assigns contents of `src` to `*this`. Returns a reference to lefthand side.
- `~concurrent_vector()`: Erases all elements and destroys the vector.
- `void clear()`: Erases all elements. Afterward, `size()==0`.

Containers: concurrent_vector

Concurrent Operations: These methods safely execute on the same instance of a concurrent_vector<T>:

- `size_t grow_by( size_t delta )`: Appends `delta` elements to the end of the vector. The new elements are initialized with `T()`. Returns the old size of the vector.
- `void grow_to_at_least( size_t n )`: Grows the vector until it has at least `n` elements. The new elements are initialized with `T()`.
- `size_t push_back( const T& value )`: Appends a copy of value to the end of the vector. Returns the index of the copy.
- `&T operator[]( size_t index )`: Returns a reference to the element with the specified index.
- `const &T operator[]( size_t index ) const`: `const` reference to the element with the specified index.

Containers: `concurrent_vector`

Parallel iteration

Concurrent Operations: These methods safely execute on the same instance of a `concurrent_vector<T>`:

- `size_t grow_by( size_t delta )`: Appends `delta` elements to the end of the vector. The new elements are initialized with `T()`. Returns the old size of the vector.
- `void grow_to_at_least( size_t n )`: Grows the vector until it has at least `n` elements. The new elements are initialized with `T()`.
- `size_t push_back( const T& value )`: Appends a copy of value to the end of the vector. Returns the index of the copy.
- `&T operator[]( size_t index )`: Returns a reference to the element with the specified index.
- `const &T operator[]( size_t index ) const`: `const` reference to the element with the specified index.