Multicore Desktop Programming with Intel Threading Building Blocks

Wooyoung Kim and Michael Voss, Intel

// Intel Threading Building Blocks is a key component of Intel Parallel Building Blocks. This widely used C++ template library helps developers achieve well-performing modular parallel programs in multiprogrammed environments. //

THE CHARACTERISTICS OF BOTH THE APPLICATIONS and the environment make writing parallel programs that perform well on multicore desktops a challenge. Often, desktop applications are modular and depend on shared, third-party libraries and plug-ins. When modules use parallel models that don't compose well, each component might act as though it can consume all available resources, making it easy for the application to oversubscribe a system.

Desktop environments are also multiprogrammed and have independent, competing applications. To perform well, parallel applications on the desktop must be able to adapt to changes in resource availability. Writing a correct parallel program is difficult; writing a highly modular parallel program that performs well in a multiprogrammed environment is even more so.

To address many difficult issues that desktop developers face, Intel designed Intel Threading Building Blocks (Intel TBB), a portable C++ template library. Intel TBB provides a range of building blocks to help developers write efficient parallel programs (see Figure 1). Its task-based programming model and algorithms let developers express parallelism easily, leveraging its work-stealing scheduler to provide a composable execution environment that can effectively adapt to changes in resources. Moreover, its containers and synchronization constructs can be used flexibly from within tasks or from native threads to provide safety and scalability.

About Intel TBB

Intel TBB is a key component of Intel Parallel Building Blocks (Intel PBB). Intel PBB is Intel's family of complementary and compatible parallel-programming models; it also includes Intel Cilk Plus[1] and Intel Array Building Blocks.[2] (For more information on other parallel programming models, see the "Related Work in Parallel-Programming Models" sidebar.)

Intel TBB is available as a commercially supported product[3] and an open source project.[4] Intel supports the library on multiple platforms including Windows, Linux, and Mac OS. It's used by many applications, including Adobe Systems' Creative Suite 5, Autodesk's Maya, Avid's Media Composer, Epic Games' Unreal Engine 3, Firaxis Games' Civilization 5, and Creative Assembly's Napoleon: Total War.

The Tasking Interface

Intel TBB's lightweight tasks and work-stealing task scheduler are key to its performance. Users express basic units of parallel work in their applications as tasks, which are user-space C++ objects. Because a task's allocation and deallocation are much more lightweight than those of a native OS thread, developers can overdecompose their problem, creating many more tasks than hardware

0740-7459/11/$26.00 © 2011 IEEE

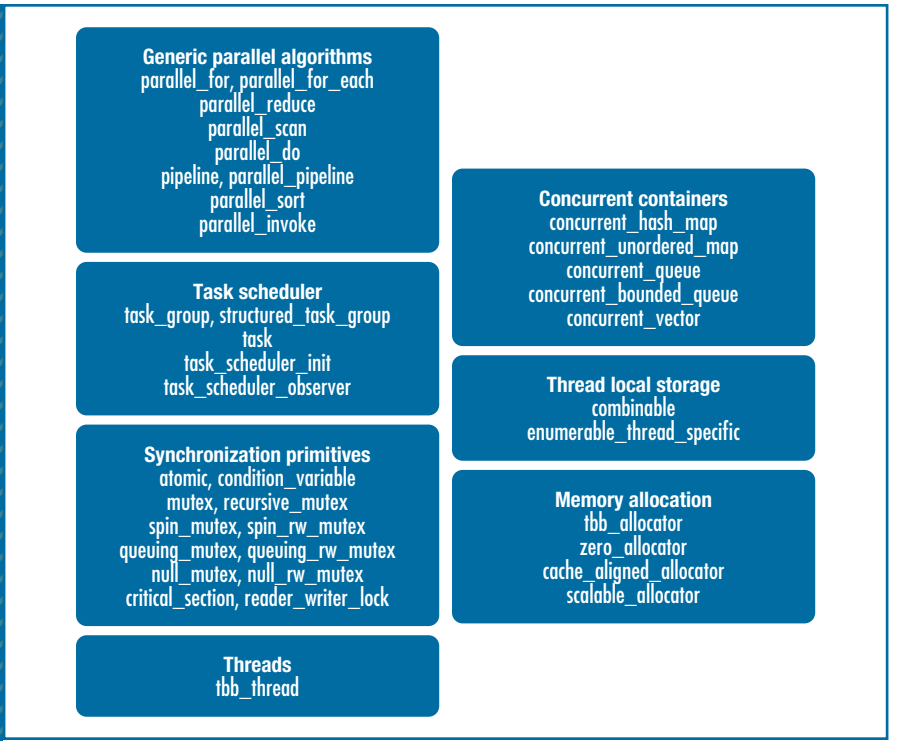

FIGURE 1. Intel Threading Building Blocks (Intel TBB) 3.0 components. This C++ template library provides a range of building blocks to help developers write efficient parallel programs.

threads. This large pool of tasks lets Intel TBB schedule work as needed to adapt to the available resources.

The main application, all libraries, and all plug-ins share a process-wide set of worker threads. Sharing worker threads allows tasks spawned from all sources to be lumped together for scheduling (which avoids oversubscription and undersubscription of resources). It also provides greater flexibility in balancing a load across hardware threads. These features are critical for highly modular applications executing on desktop systems.

Tasks and Work Stealing

The Intel TBB runtime library uses work-stealing task schedulers (inspired by Cilk[5]) to distribute tasks and balance a load among worker threads. By default, Intel TBB creates one worker thread per hardware thread. Each worker maintains a double-ended queue (deque) of tasks (see Figure 2). As the worker spawns new tasks, Intel TBB puts those tasks at the back of its deque. When a worker finishes executing a task, it takes the most recently spawned task from the back of its deque, exploiting temporal locality. This strategy unfolds recursively generated task trees in a depth-first manner, minimizing memory use.

If a worker thread doesn't find tasks in its own deque (as is the case for Worker 1 in Figure 2), it steals a task from another random worker, if possible. Stolen tasks are taken from the front of the victims' deques. For recursive algorithms, these oldest tasks are high in the task tree, represent large chunks of work, and are often cold in the victims' caches. Because tasks can execute on any worker thread, Intel TBB is free to shrink or grow the thread pool, enabling the library to react to changes in available resources.

Using Tasks

Intel TBB provides two interfaces for using tasks: `task_group` and `task`. The `task_group` class is easier to use but less flexible. The example in Figure 3 uses `task_group` to find the minimum value in a binary tree. The class `task_group` defines a `run` function. Each call to `run` creates and spawns a task that will execute the function object passed in as an argument. The example in Figure 3 also uses lambda expressions to create simple function objects at each call to `run`. (The C++ 201x language proposal introduces lambda expressions to C++.[6] They provide a concise way to create function objects and are therefore useful in libraries that frequently use function objects as arguments.)

In this example, when the number of nodes below the current node exceeds the given threshold (1,000), the program creates two child tasks to traverse the left and right subtrees. The call to `g.wait()` blocks until all the tasks in the group are complete. Once both subtrees have finished, the code obtains the final result by finding the minimum of the values computed for each subtree. The recursion continues until it reaches a node with 1,000 or fewer nodes below it. For these small subtrees, we use a serial algorithm to find the tree minimum.

When you need more control, you can use `task` instead of `task_group`. The `task` API lets users control low-level behaviors such as task-to-thread affinity, task cancellation, and exception propagation. It also can express more complex dependencies between tasks than the simple parent-child relationships that can be expressed using `task_group`. However, the resulting code is less concise than that for `task_group` and requires management of low-level details, such as reference counts. A detailed description of both APIs is in the Intel TBB reference manual.[7]

Generic Parallel Algorithms

Intel TBB also provides prepackaged, generic algorithms built on top of tasks (see Figure 1).

Iteration over a Range or Collection

Intel TBB provides algorithms that iterate over ranges or collections. The user

RELATED WORK IN PARALLEL-PROGRAMMING MODELS

Intel Cilk Plus is a C/C++ extension consisting of three tasking keywords inherited from Cilk[1] and a new array notation for vector computation. Its structured fork/join parallelism allows for features (such as hyperobjects) that Intel Threading Building Blocks doesn't.[2,3] In addition, its tasking overheads are lower because of its efficient compiler support. However, Intel Cilk Plus tasks aren't first-class objects, which makes Cilk less flexible than Intel TBB.

Intel Array Building Blocks (ArBB) provides a generalized vector-parallel-programming solution for data-intensive mathematical computation.[4,5] Users express computations as operations on arrays and vectors. A just-in-time compiler supplied with the library translates the operations into target-dependent code, in which a target could be the host CPU or an attached GPU. Intel ArBB can run data-parallel vector computations on a possibly heterogeneous system, whereas Intel TBB focuses on task-based fork/join parallelism on a homogeneous system.

The OpenMP API is a pragma-based extension to C/C++ primarily for high-performance computing. It provides high-level parallel constructs built around the thread teams concept and is extended to support tasks.[6] This reliance on thread teams that work on an identical piece of code could easily complicate managing nested parallelism and resource sharing on desktop computers. For example, many OpenMP implementations by default create additional threads at each nested parallel region.

The Microsoft Parallel Patterns Library (PPL) is a C++ template library that's similar to Intel TBB. For instance, some high-level Intel TBB algorithms and containers have corresponding abstractions in PPL. PPL uses the Concurrency RunTime (ConcRT) for task scheduling and load balancing. Intel TBB is both a commercial and open source project. It uses a task scheduler that can run on Microsoft Windows with or without the ConcRT, and it's supported on other platforms including Linux and Mac OS.

Kronos OpenCL (Open Computing Language), Microsoft DirectCompute, and Nvidia's CUDA (Compute Unified Device Architecture) target heterogeneous systems typically consisting of a host machine and remote computing engines (such as attached GPUs). They share a programming model similar to that of Intel ArBB, in which host threads offload data-parallel work onto remote computing engines. However, they require programmers' direct control at every level. In contrast, Intel TBB focuses on homogeneous systems with transparent load balancing through work stealing.

Apple's Grand Central Dispatch (GCD) is a tasking system that combines queues with closures called blocks.[7] Programmers explicitly enqueue blocks into the main queue and into local and global queues they've created. The system schedules this enqueuing on the basis of the available cores' priorities. Although GCD manages the sharing of system resources among different applications, it doesn't provide high-level algorithms or containers and requires a compiler that recognizes blocks.

References

1. M. Frigo, C.E. Leiserson, and K.H. Randall, "The Implementation of the Cilk-5 Multithreaded Language," *Proc. ACM SIGPLAN 1998 Conf. Programming Language Design and Implementation* (PLDI 99), ACM Press, 1998, pp. 212–223.
2. M. Frigo et al., "Reducers and Other Cilk++ Hyperobjects," *Proc. 21st Ann. Symp. Parallelism in Algorithms and Architectures*, ACM Press, 2009, pp. 79–90.
3. "A Quick, Easy and Reliable Way to Improve Threaded Performance: Intel Cilk Plus," Intel, 2010; http://software.intel.com/en-us/articles/intel-cilk-plus.
4. "Sophisticated Library for Vector Parallelism: Intel Array Building Blocks," Intel, 2010; http://software.intel.com/en-us/articles/intel-array-building-blocks.
5. A. Ghuloum et al., "Future-Proof Data Parallel Algorithms and Software on Intel Multi-core Architecture," *Intel Technology J.*, vol. 11, no. 4, 2007, pp. 333–347.
6. *OpenMP Application Program Interface Ver. 3.0*, OpenMP Architecture Review Board, May 2008; www.openmp.org/mp-documents/spec30.pdf.
7. "Introducing Blocks and Grand Central Dispatch," Apple, 2010; http://developer.apple.com/library/mac/#featuredarticles/BlocksGCD/.

specifies both a range and a body to apply to the elements in the range. The runtime library creates tasks from the range or collection by recursive subdivision. This process is similar to the one in the tasking example we described earlier. The library first subdivides an initial range into two tasks, each handling roughly half of the range. When the task executes, it decides whether to further subdivide its range into two additional tasks or to apply the loop's body serially to the subrange. Figure 4 shows a computation in flight, in which recursive division hasn't yet terminated for all subranges.

To control the policy for terminating recursion, users can employ partitioners. The library supports three partitioning policies. The first, `simple_partitioner`, recursively divides the range if its size is greater than a user-set threshold.

The second policy, `auto_partitioner`, is the default; it monitors stealing behavior.[8] If no stealing occurs, it divides the range into $p \times 4$ subranges, where p is the number of hardware threads. However, if a task is stolen, the thief divides that task into four additional pieces, if possible. The reasoning behind `auto_partitioner` is that stealing indicates imbalance and that finer-grained tasks should be created to enable load balancing.

The last policy is `affinity_partitioner`. It keeps a history of the threads that executed a specific subrange on previous executions of the loop (or other loops) and tries to maintain this distribution on subsequent executions.

Figure 5 shows `parallel_for` examples that don't specify a partitioner and therefore use `auto_partitioner`, the default. Both Figures 5a and 5b use a `parallel_for` loop to set `output[i]` to the average of `input[i-1]`, `input[i]`, and `input[i+1]`, for $1 \leq i < n$. In Figure 5a, a function object describes the loop's body; in Figure 5b, a lambda expression describes it. In both cases, the `parallel_for` function template concurrently applies the loop body to the range's elements.

The Intel TBB library also provides `parallel_reduce` and `parallel_scan`, which also

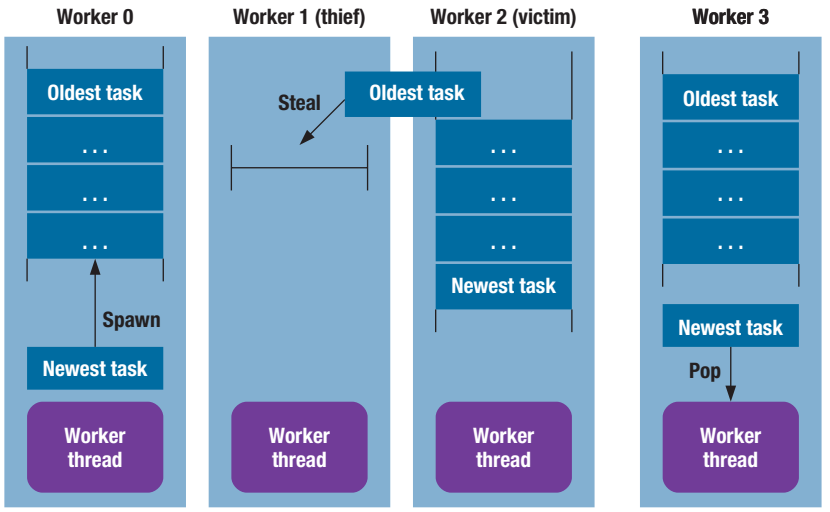

FIGURE 2. Obtaining tasks from the local double-ended queues (deques) maintained by the worker threads. This strategy unfolds recursively generated task trees in a depth-first manner, minimizing memory use.

```
float find_tree_min( tree_node *my_node ) {
  float my_min = FLT_MAX;
  if ( my_node->num_nodes_below > 1000 ) {
    tbb::task_group g;
    float min_left = FLT_MAX, min_right = FLT_MAX;
    if ( my_node->left_child )
      g.run( [&] { min_left = find_tree_min( my_node->left_child ); } );
    if ( my_node->right_child )
      g.run( [&] { min_right = find_tree_min( my_node->right_child ); } );
    g.wait();
    my_min = std::min( my_node->value, std::min( min_left, min_right ) );
  } else {
    my_min = serial_tree_min( my_node );
  }
  return my_min;
}
```

FIGURE 3. Using **task_group** to find the minimum value in a binary tree. This example also uses lambda expressions to create anonymous function objects.

apply a body to a range of elements concurrently. They support calculation of a reduction and a parallel prefix, respectively.

Adding New Items While Iterating

The algorithms we just described assume that the complete iteration space is known a priori. However, for many applications, the iteration space's end isn't known in advance. To handle this case, Intel TBB provides the **parallel_do** template function.

To use **parallel_do**, users provide **begin** and **end** iterators and a function object (or lambda expression). If the user doesn't provide random-access iterators, the algorithm ensures that no more than one thread will ever act on the iterators concurrently. This respects the definition of input iterators for sequential programs. However, if the input iterators are random-access, this constraint is relaxed, which lets the algorithm be more scalable.

The **parallel_do** body argument might take a **parallel_do** feeder as its second argument. In this case, the body might add new items to the iteration space by calling the feeder's **add** function. Figure 6 shows an example of **parallel_do**.

Execution of Pipelined Computations

The **parallel_pipeline** function is an interface that applies a series of filters to a stream of items in a pipelined fashion. Each filter operates in a particular mode: parallel, serial in-order, or serial out-of-order.[9] Parallel filters process items as they arrive and might do so concurrently. Serial in-order filters process one item at a time, in the order in which those items entered the pipeline. Serial out-of-order filters process one item at a time, but in the order in which those items arrive at the filter.

Figure 7 shows an example of **parallel_pipeline**; it's a syntactic demonstration only. It's not a practical way to calculate a root mean square because overheads would likely dominate the calculation. However, when there is sufficient work per item and sufficient processors and items, the throughput of **parallel_pipeline** is limited only by the slowest serial stage.

Concurrent Containers

The prepackaged parallel algorithms we described earlier are useful in expressing parallelism in applications. Concurrent containers are equally (if not more so) useful in developing parallel applications for desktop computers because threads commonly use concurrent containers to communicate and synchronize with each other.

A typical C++ standard template library (STL) container isn't safe for concurrent access and can't be used for thread communication in its "naked" form. You can make it thread-safe by wrapping it in a mutex, an object on

which a thread can acquire a lock. However, this approach eliminates concurrency and creates a bottleneck. The concurrent containers in Intel TBB offer highly concurrent, scalable alternatives to mutex-wrapped, serial STL containers.

Intel TBB provides containers that yield a high level of concurrency and scalability because they use specialized container organizations, fine-grain locking, and lock-free techniques. Of course, this high level of concurrency is costly. Intel TBB containers are less effective for applications in which contention is light. A rule of thumb is to use Intel TBB containers for applications that may exhibit burst access patterns to shared data—that is, concurrent accesses to them in short time spans.

Intel TBB provides three kinds of concurrent containers: `concurrent_hash_map` or `concurrent_unordered_map`, `concurrent_vector`, and `concurrent_queue` or `concurrent_bounded_queue`. Their interfaces are modeled after their STL counterparts in the C++ 201x standard proposal.[6] However, some STL methods are absent from Intel TBB containers, and others have different semantics in Intel TBB containers.[10] Developers can use these containers in conjunction with task-based programming or with native Windows or Linux threads.

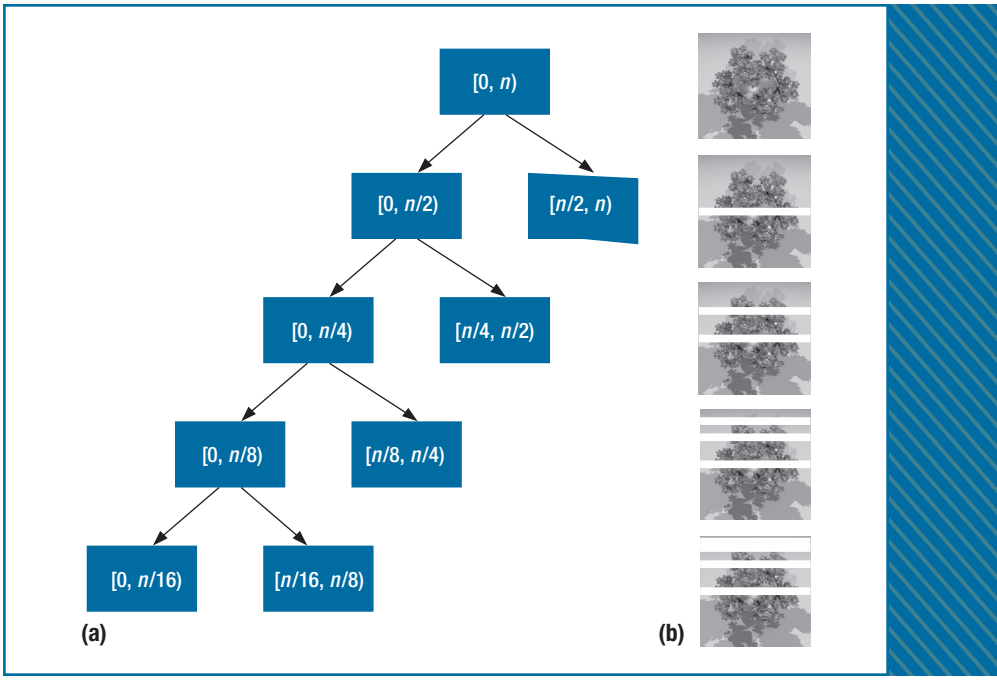

FIGURE 4. Recursive subdivision of a range to generate tasks. (a) Each box represents a task that will perform computation on a subrange. The leaves represent tasks that haven't yet executed, and the internal nodes represent tasks that have executed and chose to subdivide their range. (b) An image generated by the Tachyon ray tracer demonstrates how the range's division translates to the pixels computed in the final image.

Concurrent Associative Containers

The `concurrent_hash_map` and `concurrent_unordered_map` containers are extensions to the sequential associative containers. Because both add concurrency to a hash map, they have similar semantics. For example, both have unordered keys and at most one element for each key. However, they were developed with different design objectives and thus have different interfaces and, more importantly, have different concurrency requirements.

The `concurrent_hash_map` class allows concurrent insertion, lookup, and erasure on the same map instance. It services concurrent access using auxiliary objects called accessors. An accessor acts as a smart pointer to a pair in a `concurrent_hash_map`. It holds an implicit lock on the pair until it's destroyed or the lock is explicitly released. If the map contains a (key, value) pair corresponding to an input key, `insert` and `find` return the pair in the accessor passed to the map; `erase` deletes the pair from the map. If no pair exists for a given key, `insert` constructs a pair with the key and inserts it into the map. In addition, these operations invalidate any iterators pointing into the affected element; thus, concurrent traversal isn't allowed.

Although `concurrent_hash_map` is useful, it lacks some desirable features. In some use cases, accessor-based concurrent operations are less flexible than those that don't require accessors, and they incur unnecessary overhead. For example, every lookup on a `concurrent_hash_map` imposes a cost on the application for internal locking—even for read-only access. Additionally, the `concurrent_hash_map` interface isn't quite aligned with that of the corresponding associative container in the C++ 201x proposal.[6] The `concurrent_unordered_map` class addresses these two issues. It closely resembles `std::unordered_map` in the C++ 201x proposal,[6] although it omits methods requiring C++ 201x language features (such as rvalue references). In particular, insertion, lookup, and erasure return iterators and have no visible locking. Users must ensure race-free access to the elements in a `concurrent_unordered_map`.

Intel TBB allows concurrent insertion on the same `concurrent_unordered_map`. However, unlike `concurrent_hash_map`, insertion of new items doesn't invalidate iterators or change the order of items already in the map. Concurrent erasure isn't permitted. These changes enable concurrent insertion and traversal on the same map. Figure 8 compares operations on the two concurrent maps.

Concurrent Vectors

A `concurrent_vector` is an array of elements that permits concurrent read access and dynamic growth. Multiple threads

```
class my_body {
  float *input;
  float *output;
  const int N;

public:

my_body( float *in, float *out,
             int n ) :
  input(in), output(out), N(n) {}

  void operator()( int i ) const {
    output[i] = (input[i-1]+input[i]+
                 input[i+1])*(1/3.f);
  }
};

void DoParallelAverage(float *input,
        float *output, int N) {
  my_body b( input, output, N );
  parallel_for( 1, N-1, b );
}
```

(a)

```
void DoParallelAverage2(float *input,
  float *output, int N) {
  parallel_for( 1, N-1,
    [=]( int i ) {
      output[i] = (input[i-1]+input[i]+
                   input[i+1])*(1/3.f);
  });
}
```

(b)

FIGURE 5. Two versions of **parallel_for**, in which the body is a (a) function object or (b) lambda expression. The two are functionally equivalent. The lambda expression version is more concise, but not all compilers support lambda expressions.

```
struct Item {
   data_type data;
   std::list<Item> *sub_list;
};

void ParallelApplyFooToList( const std::list<Item> &my_list ) {
   tbb::parallel_do( my_list.begin(), my_list.end(),
    []( const Item &item, tbb::parallel_do_feeder<Item> &feeder )
    {
      Foo(item);
      if ( item.sub_list ) {
        for ( std::list<Item>::iterator i = item.sub_list->begin();
              i != item.sub_list->end(); ++i )
          feeder.add( *i );
        }
      }
   );
}
```

FIGURE 6. A **parallel_do** example. This code traverses the initial list **my_list**. Whenever it reaches an item containing a sublist, it adds all the items in the sublist to the iteration space, using calls to **feeder.add**.

can append new elements to a **concurrent_vector** and grow it concurrently without invalidating existing iterators or indices.

This design choice has two important design and implementation consequences. First, methods that might invalidate existing iterators and indices aren't allowed—this includes **erase()** and **insert()**. Second, existing elements can't relocate when a concurrent vector grows, which means elements might not be stored contiguously, as they are in STL vectors.[6] For example, the routine in Figure 9 safely appends a C string to a shared vector. Note the use of **std::copy** and iterators.

Concurrent Queues

A **concurrent_queue** is a first-in, first-out data structure that permits multiple threads to concurrently add and remove items. Its capacity is unbounded, subject only to the target machine's memory limitations. It doesn't have methods that block, which makes it appropriate when synchronization must occur at a high level. (The **concurrent_bounded_queue** class is a bounded variant with finite capacity and blocking semantics.)

The fundamental operations on a **concurrent_queue** are the **push** and **try_pop** methods. Because the concurrent queue's capacity is unbounded, **push** always succeeds (provided that the target machine has available memory). The **try_pop** method pops an item if it's available; the check and popping occur atomically.

Other Building Blocks

In addition to high-level parallel algorithms and concurrent containers, Intel TBB provides other low-level building blocks such as mutexes and atomic operations.

Mutexes and Locks

All Intel TBB mutexes have a similar interface; this makes them easier to learn and enables generic programming. For example, all Intel TBB mutexes have a nested **scoped_lock** type, which implements the **acquire** and **release** methods.

The simplest mutex class is **spin_mutex**. As its name implies, it requires that threads spin-wait until acquiring the lock. Figure 10 gives an example; the constructor for **scoped_lock** waits until no other locks are on **spin_mutex**, and the destructor releases the lock.

The **spin_mutex** class is unfair and nonrecursive, and it spin-waits in user space. However, it's fast in lightly contended situations and is the mutex of choice when a design spreads contention among many **spin_mutex** objects.

The **queuing_mutex** class is fair and also nonrecursive. Although it also spin-waits in user space, it spins each thread

```
float RootMeanSquare( float* first, float* last ) {
  float sum=0;
  parallel_pipeline( /*max_number_of_live_token=*/16,
    make_filter<void,float*>(
      filter::serial,
      [&](flow_control& fc)-> float*{
        if( first<last ) {
          return first++;
        } else {
          fc.stop();
          return NULL;
        }
      }
    ) &
    make_filter<float*,float>(
      filter::parallel,
      [](float* p){return (*p)*(*p);}
    ) &
    make_filter<float,void>(
      filter::serial,
      [&](float x) {sum+=x;}
    )
  );
  return sqrt(sum);
}
```

FIGURE 7. A **parallel_pipeline** example. This code creates three filters: a serial filter iterates through the items in the list, a parallel filter squares each item, and another serial filter adds each squared value to the final sum. The overloaded **operator&** concatenates the filters.

```
using namespace tbb;
typedef
  concurrent_hash_map<string,int> StringTableH;
StringTableH htable;
StringTableH::accessor a;
for(string* p=range.begin();p!=range.end();++p)
{
  htable.insert( a, *p );
  a->second += 1;
  a.release();
}

StringTableH::const_accessor ca;
bool b = htable. find( ca, str_key )
        && htable.erase( ca );
(a)
using namespace tbb;
typedef
  concurrent_unordered_map<string,atomic<int> >
  StringTableU;
StringTableU utable;
...
for(string* p=range.begin();p!=range.end();++p)
{
    string_t::iterator i = utable.insert( *p );
    ++(*i).second;
}

string_t::iterator i = utable. find( *p );
if( i!=utable.end() ) utable.unsafe_erase(i);
(b)
```

FIGURE 8. Comparison of (a) **concurrent_hash_map** and (b) **concurrent_unordered_map**. Insertion on a **concurrent_hash_map** requires use of an accessor as well as an explicit release when accessing the element is complete, whereas only **unsafe_erase** is supported on **concurrent_unordered_map**.

on a different location (which reduces pressure on memory traffic) and is thus more scalable than **spin_mutex**.

The **null_mutex** class does nothing; it mainly enables generic programming. For example, it's useful for instantiating a thread-private container by using a container template with a mutex type as one of its template arguments.

The three previous classes have reader/writer variants (denoted by **_rw_** in the class names—for example, **spin_rw_mutex**), which allows multiple readers in the protected region. The **mutex** and **recursive_mutex** classes are wrappers around the system's "native" mutual-exclusion interfaces.

Atomic Operations

These operations, which are low-level and hardware-dependent, appear to occur instantaneously. They're quick compared to locks and never suffer from lock pathologies. However, they do only a limited set of operations. So, developers often use them as building blocks for more complicated operations.

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

FIGURE 9. An example of **concurrent_vector**. This routine safely appends a C string to a shared vector.

Intel TBB makes these operations portable by hiding them under the C++ template class **atomic<T>**. Intel TBB supports the five fundamental atomic operations in Table 1. For syntactic convenience, it provides additional interfaces in the form of overloaded operators.

Some architectures have weak memory consistency, which means the hardware might reorder memory operations on different addresses for efficiency.[11,12] To account for this, **atomic<T>** lets programmers enforce certain ordering of memory operations as they like (see Table 2). In Table 2, the column on

TABLE 1

Fundamental operations on a variable x of Atomic<T>.

Operation	Description
= x	Read *x*.
x =	Write to *x* and return it.
x.fetch_and_store(y)	Do *y* = *x*, and return the old value of *x*.
x.fetch_and_add(y)	Do *x* += *y*, and return the old value of *x*.
x.compare_and_swap(y,z)	If *x* = *z*, then do *x* = *y*; in either case, return the old value of *x*.

the right lists operations that default to a particular constraint. If desired, users employ variants taking a template argument, **acquire** or **release**, to relax these defaults. For example, **release** in **refcount.fetch_and_add<release>(-1);** guarantees that stores before the decrement are visible before **refcount** is decremented. However, it also allows loads after the decrement to occur before the decrement.

Thread-Local Storage

Intel TBB provides two template classes for making thread-local storage available to programmers: **combinable** and **enumerable_thread_specific**. Both provide a **local** function that returns (or lazily creates) one thread-local element per thread and a **combine** function that reduces these thread-local elements to a single value. However, the **enumerable_thread_specific** class also acts like an STL container and permits iteration over the elements using the usual STL iteration idioms.

Performance

Figure 11 shows the performance of three sample Intel TBB applications, which executed using one through 32 threads on a system with 32 cores. The Intel TBB 3.0 distribution includes all three applications as examples.

The polygon overlay application is an implementation of Polygon Overlay from the Cowichan Problems.[13] It divides two maps of equal extent into nonoverlapping polygons and generates a resulting map overlaying those polygons. It uses **parallel_for** to iterate over the submaps and **enumerable_thread_specific** to store polygons generated by intersecting the maps.

Tachyon is based on John Stone's 2D ray tracer and renderer.[14] It uses **parallel_for** and **blocked_range2d** to parallelize over

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

FIGURE 10. A **spin_mutex** example. The **spin_mutex** class requires that threads spin-wait until acquiring the lock. In this example, the constructor for **scoped_lock** waits until no other locks are on **spin_mutex**, and the destructor releases the lock.

FIGURE 11. The performance of Intel TBB 3.0 for the Polygon Overlay, Tachyon, and Primes applications.

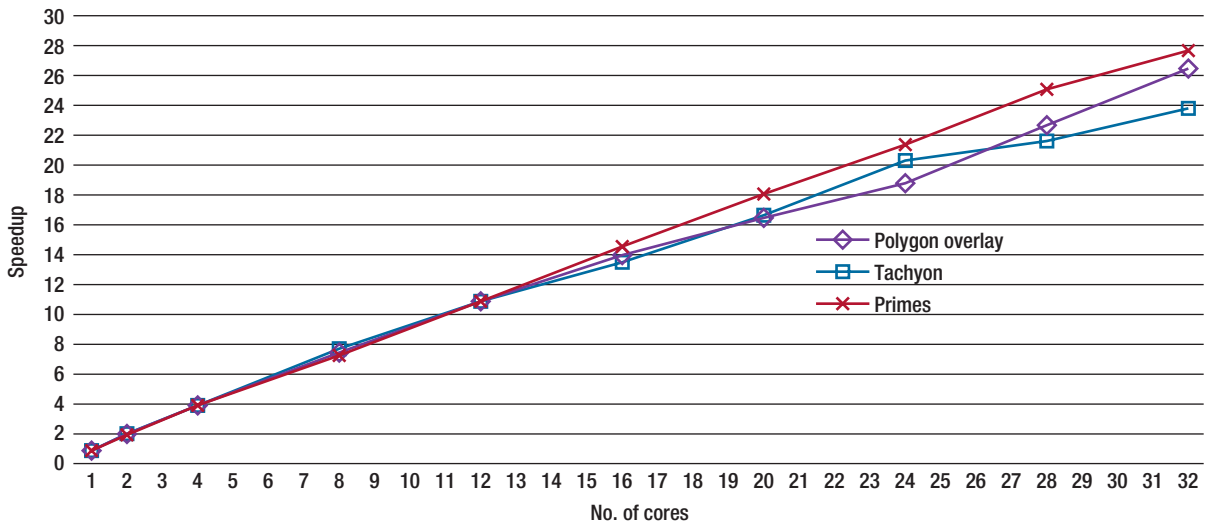

TABLE 2

Ordering constraints.

Constraint	Description	Default for
acquire	Operations after the atomic operation never move over the atomic operation.	read
release	Operations before the atomic operation never move over the atomic operation.	write
Sequentially consistent	Operations on either side never move over the atomic operation; the sequentially consistent atomic operations have a global order.	fetch_and_store fetch_and_add compare_and_swap

tasks that are rectangular subareas of the image. Figure 4b shows sample images from Tachyon.

Primes is a parallel version of the Sieve of Eratosthenes.[15] It uses **parallel_reduce** to compute all prime numbers up to a given integer.

For more information regarding performance and optimization choices in Intel software products, see http://software.intel.com/en-us/articles/optimization-notice.

Intel TBB lets developers build well-performing applications by expressing parallelism in their applications using high-level constructs and by relegating details of scheduling and synchronization to the runtime library. In addition, Intel TBB provides a set of low-level constructs that developers can use directly, adding flexibility that many other competing parallel programming models lack. Intel TBB is continually evolving to meet the demands of the emerging parallel desktop development community. To download the most recent release of the library, to learn about upcoming features, or to provide feedback on new features, visit the community website at www.threadingbuildingblocks.org.

References

1. "A Quick, Easy and Reliable Way to Improve Threaded Performance: Intel Cilk Plus," Intel, 2010; http://software.intel.com/en-us/articles/intel-cilk-plus.
2. "Sophisticated Library for Vector Parallelism: Intel Array Building Blocks," Intel, 2010; http://software.intel.com/en-us/articles/intel-array-building-blocks.
3. "Deliver Scalable and Portable Parallel Code: Intel Threading Building Blocks," Intel, 2010; http://software.intel.com/en-us/intel-tbb.
4. "Intel Threading Building Blocks 3.0 for Open Source," Intel, 2010; www.threadingbuildingblocks.org.
5. M. Frigo, C.E. Leiserson, and K.H. Randall, "The Implementation of the Cilk-5 Multithreaded Language," *Proc. ACM SIGPLAN 1998 Conf. Programming Language Design and Implementation* (PLDI 99), ACM Press, 1998, pp. 212–223.
6. *Programming Languages – C++*, Int'l Org. for Standardization, Mar. 2010; www.open-std.org/JTC1/SC22/WG21/docs/papers/2010/n3092.pdf
7. *Intel Threading Building Blocks Reference Manual*, Intel, 2010; http://software.intel.com/en-us/articles/intel-threading-building-blocks-reference-pdf.
8. A. Robison, M. Voss, and A. Kukanov, "Optimization via Reflection on Work Stealing in TBB," *Proc. 2008 IEEE Int'l Symp. Parallel and Distributed Processing* (IPDPS 08), IEEE Press, 2008, pp. 1–8.
9. S. MacDonald, D. Szafron, and J. Schaeffer, "Rethinking the Pipeline as Object-Oriented States with Transformations," *Proc. 9th Int'l Workshop High-Level Parallel Programming Models and Supportive Environments* (HIPS 04), IEEE CS Press, 2004, pp. 12–21.
10. A. Marochko, "TBB Containers vs. STL. Functionality Rift," blog, Intel, 13 Oct. 2008; http://software.intel.com/en-us/blogs/2008/10/13/tbb-containers-vs-stl-functionality-rift.
11. *Intel 64 and IA-32 Architectures Software Developer's Manual, Vol. 3A: System Programming Guide, Part 1*, Intel, 2010; www.intel.com/Assets/PDF/manual/253668.pdf.
12. P. Sewell et al., "x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors," *Comm. ACM*, vol. 53, no. 7, 2010, pp. 89–97.
13. G.V. Wilson, "Assessing the Usability of Parallel Programming Systems: The Cowichan Problems," *Proc. IFIP Working Conf. Programming Environments for Massively Parallel Distributed Systems*, 1994, pp. 183–193.
14. J. Stone, "Tachyon Parallel / Multiprocessor Ray Tracing System," Sept. 2010; http://jedi.ks.uiuc.edu/~johns/raytracer.
15. E.W. Weisstein, "Sieve of Eratosthenes," Wolfram Research, 2010; http://mathworld.wolfram.com/SieveofEratosthenes.html.

ABOUT THE AUTHORS

WOOYOUNG KIM is a software engineer in the Intel Software and Services Group. His interests include programming languages and runtime support for parallel, concurrent, and distributed systems (including shared-memory, multicore systems, and wireless sensor networks). Kim has a PhD in computer science from the University of Illinois at Urbana-Champaign. Contact him at wooyoungdim@intel.com.

MICHAEL VOSS is a software engineer in the Intel Software and Services Group and a developer of Intel Threading Building Blocks. His research interests include languages and compilers for parallel computing and adaptive program optimization. Voss has a PhD in electrical engineering from Purdue University. Contact him at michaelj.voss@intel.com.