A C++ Library Solution To Parallelism

Alexey Kukanov (Intel)
Artur Laksberg (Microsoft)
Arch Robison (Intel)

Abstract

This white paper describes the common subset of two C++ programming libraries: the Parallel Patterns Library (PPL) and the Threading Building Blocks library (TBB) [1]. Significant part of this subset was developed jointly by Microsoft (as part of PPL and the Visual Studio product) and Intel (as part of TBB and the Composer XE product). Our goal is to provide a high-level cross-platform parallel programming model for C++.

Motivation

Concurrent programming on modern operating systems is based on multithreading – the execution model that allows multiple threads to run at the same time, sharing one or more CPU cores.

For an application that creates a small number of threads that require little or no synchronization, threads work just fine. However, for an application with fine-grain concurrency requirements, the overhead of creation and destruction of threads can negate the benefits of parallelism offered by the multiple CPU cores. Moreover, threads are expensive memory-wise. By default, a single thread allocates 1 MB of stack space on Windows and 8 MB on Linux. While this number is configurable, reducing the stack size may break programs with deep call chains or multiple stack-allocated objects.

Thread pools were invented to solve this problem. A thread pool maintains the optimal number of threads in the process – no more than necessary, but enough to maximize the CPU utilization and ensure forward progress of the program. Ideally, the number of threads would be equal to the number of processor cores in the system, but when the threads block, a thread pool can inject more threads into the system, allowing more useful work to take place. While no perfect thread pool algorithm exists, modern thread pools do a reasonably good job at maintaining the optimal number of threads for most common applications.

However, thread pools do not solve another fundamental problem of threads – which is the lack of an easy-to-use programming model. We can quote Prof. Edward E. Lee of UC Berkley, who in his seminal paper (1) "The Problem with Threads" declared:

> *[Threads] discard the most essential and appealing properties of sequential computation: understandability, predictability, and determinism.*
>
> *Threads, as a model of computation, are wildly nondeterministic, and the job of the programmer becomes one of pruning that nondeterminism.*

[1] PPL and TBB implementations differ in some details but provide similar semantics overall. Please consult with the documentation for details.

It is our belief that threads are not a good construct to program against. We need higher level abstractions – in the form of either a library or a language-based solution, that allow us to be productive while writing parallel code, and ideally, steering us towards good parallel patterns and practices.

To further illustrate this point, we refer the reader to the work of Victor Duvanenko (2) that demonstrates using TBB and PPL to build various implementations of parallel sort. The author provides an implementation based on the *parallel_invoke* (that we will introduce on page 3) which, expectedly, outperforms the serial version of the sort for sufficiently large arrays. The algorithm does not have constraints on the upper bound of the array – it is limited only by the amount of memory available to the program.

When, for the sake of an experiment, we replaced the invocation of *parallel_invoke* with an explicit thread creation in the *parallel_merge_sort_hybrid_rh*, the implementation crashed consistently due to thread exhaustion (we explain this phenomenon on page 2 below) when run on arrays as small as 50K elements. Note that 50K is below the minimum array size for which a parallel version of sort outperforms the serial version.

Further, we experimentally introduced threads in the *merge_parallel_L5* function, invoked by the *parallel_merge_sort_hybrid_rh*. After this change, the function would not scale beyond 15K integers. This speaks to the importance of *composability* of parallel algorithms, a property lacking in threads. To generalize this observation, we can claim that the scalability of a solution composed of several thread-based components tends to be less than that of each of the components.

Structured Parallelism in PPL and TBB

As the first step towards the goals, we depart from the thread as the basic building block of concurrency in favor of the *task*. A task is a unit of work that runs sequentially and produces a result by either explicitly yielding a value or via a side effect. The job of the programmer is to divide a workload into independent (or interdependent) tasks and allow the thread pool to map these tasks to threads. Furthermore, structuring the relationships between tasks permits more efficient mapping to threads.

In this paper we consider the *fork/join* parallel model in which an algorithm is partitioned into many independently executing tasks, that are spawned, run independently, and then are joined to produce the final result. Much as serial control structures simplify reasoning about sequential control flow, the fork/join control structures simplify reasoning about parallelism – the parallelism created is limited to the dynamic scope delimited by the fork and the join.

task_group

The most basic construct of the fork/join parallelism in PPL is called the *task_group*[2]. A task group is a collection of tasks that can run in parallel and can be collectively waited on.

A typical use of a task group goes like this:

```
task_group tg;
```

[2] The name resides in the *concurrency* namespace. All PPL constructs are defined in this namespace, which for brevity is not explicitly mentioned elsewhere in the document. In TBB, *task_group* and other parallel constructs are also defined in the *tbb* namespace.

```
tg.run([] {
    f();
});
tg.run([] {
    g();
});
tg.wait();
```

Here we created an instance of the *task_group* class and added two work items to it using the *run* method. This method expects a callable object – which typically would be a C++ lambda expression.

The work items added to the task group are not started immediately but are placed on the thread pool[3]. As soon as the thread pool finds a thread available to run it, the work item gets executed by that thread.

An important optimization implemented by the task group is called the *task inlining*. When the *wait* method on a task group is called, some or all of the pending tasks in the task group can execute on the calling thread. The idea is easy to understand – instead of waiting for someone else to become available to pick up and complete the work, it's better to just do it yourself. This technique is essential in preventing the "thread explosion" phenomenon – a situation where a thread spawns and then waits for another thread to perform work, which in turn spawns and waits for another thread, ad infinitum. Thread explosion can occur in parallel divide-and-conquer algorithms, such as a typical parallel implementation of sort (such as the one described in (12)).

Task inlining can be enforced by the *run_and_wait* method that has the same signature as the *run* method. Whereas *run* followed by the *wait* may inline if the task hasn't started running already, the *run_and_wait* is guaranteed to inline the task. Note that enforced task inlining is also preferred to direct execution of a work item followed by the *wait*, as it may help to manage HW cores more efficiently, and may return earlier if the *task_group* was cancelled.

parallel_invoke

The *parallel_invoke* function is one of the simplest constructs in PPL/TBB. The function takes two or more callable objects and executes them in parallel. Here is how our previous example of running *f* and *g* in parallel could be written with the *parallel_invoke*:

```
parallel_invoke(
    []{f();},
    []{g();}
);
```

As it is well known, functional parallelism, which *parallel_invoke* represents, only scales up to the number of specified tasks (which typically is a fairly small number), unless applied recursively.

[3] The Microsoft's PPL uses a specialized implementation of a thread pool called the Concurrency Runtime, or ConcRT for short

The Microsoft version of *parallel_invoke* is implemented based on the *task_group*, but that's not the only possible implementation. Variadic templates would simplify the implementation significantly, though this recent C++ feature was not available during the implementation.

Loop Paralelization with parallel_for

Anecdotally, the *parallel_for* function is by far the most widely used algorithm in the PPL and TBB. For many of PPL users, this is the only algorithm they ever used.

Indeed, loops are often the most computationally intensive part of an application, and parallelizing the "hot" loops is the surest way to improve the program's performance. Unlike the *parallel_invoke*, the *parallel_for* can scale much better, as long as the number of iterations in the loop is sufficient to give enough work to all cores in the system. Ideally, *parallel_for* should produce much more tasks than the number of cores, to provide slack for load balancing.

In the following example, we use the *parallel_for* to parallelize a matrix multiplication algorithm:

```
void MatrixMultiplyPar(int rows1, int cols1, float **mat1, int rows2, int
cols2, float **mat2, float **result)
{
    parallel_for(0, rows1, [=](int i)
    {
        for(int j=0;j<cols2;j++)
        {
            float tempResult = 0;
            for(int k=0;k<rows2;k++)
            {
                tempResult += mat1[i][k]*mat2[k][j];
            }
            result[i][j] = tempResult;
        }
    });
}
```

The *parallel_for* looks sufficiently close to a typical *for* statement in a C++ program to be easy to understand and use. The function takes the upper and the lower bounds of the loop, the optional step value (1 if omitted), and the callable object (typically, a C++ lambda expression) that will be invoked for each iteration of the loop. The input parameter to the lambda is the induction variable of the loop.

PPL and TBB have been on the market for several years now, giving us ample opportunity to observe both the good usage patterns as well as the common pitfalls with the *parallel_for*.

Most novice users quickly realize that not every loop is amenable to parallelization. (A C++ hobbyist once commented on the Microsoft C++ user's group that *parallel_for* breaks his implementation of bubble sort). Other hazards include race conditions and a variety of performance issues (using locks inside loops, false sharing etc.) Both Microsoft and Intel have accumulated vast experience educating users on the best practices of parallel programming in C++. Microsoft has published a guide on best practices (2) as well as an in-depth exploration of PPL patterns and anti-patterns (4). Intel likewise has a guide on TBB patterns (9).

parallel_for Implementation Overview

The Microsoft implementation of the *parallel_for* is based on the *task_group*. Depending on the nature of the workload, different partitioning strategies can be used:

- **Static Partitioner** divides the range into as many chunks as there are cores, and runs these chunks as parallel tasks. This partitioner is used for well-balanced workloads – that is, when all iterations of the loop are of roughly equal computational complexity.
- **Simple Partitioner** divides the range into multiple chunks where the size of the chunks is specified by the user. Having a large number of tasks is useful for uneven workloads – when some tasks finish, others will be picked up by the thread pool ensuring continuous CPU utilization. The drawback of this approach is greater memory consumption and more contention on the thread pool task queue.
- **Auto Partitioner** is the default partitioner in PPL. This partitioner divides the range into a fixed number of chunks and then employs a technique for dynamic redistribution of work known as *range-stealing.* This allows the tasks that complete sooner to “steal” iterations from the other, still running, tasks.

Our measurements show that the auto partitioner performs better in most cases, and is effective for uneven workloads, such as the Mandelbrot algorithm. For the workloads known to be balanced we recommend either the static or the simple partitioner.

The Intel implementation of the *parallel_for* employs task stealing and a heuristic that dynamically estimates how many iterations to put in each task (7).

parallel_for_each

The *parallel_for_each* function applies a specified functor to each element within a range, in parallel. It is semantically equivalent to the *for_each* function in the *std* namespace, except that iteration over the elements is performed in parallel, and the order of iteration is unspecified.

The effectiveness of the *parallel_for_each* depends on the type of the iterator provided. For a random access iterator, the entire range can be determined up front, and the implementation can fall back to the *parallel_for*.

For forward iterators, the *parallel_for_each* iteratively “peels off” portions of the range and executes these portions in parallel. Clearly, the cost of advancing the iterator plays a major role in the effectiveness of the *parallel_for_each*.

In TBB, Intel additionally provides *parallel_do* which is analogous to *parallel_for_each* but also has an option to dynamically add new work items produced during its execution. For applications where it fits it can reduce the iteration cost and improve scaling.

parallel_transform

This functional in PPL is semantically equivalent to the *std::transform*. It applies a specified function object to each element in a source range, or to a pair of elements from two source ranges, and copies the return values of the function object into a destination range, in parallel.

For all intents and purposes, the *parallel_transform* can be thought of as a faster version of the *std::transform.* The only obvious caveat is, the function object must be concurrency safe (normally it is a pure function, so it is not an issue).

parallel_reduce

The *parallel_reduce* function is the parallel counterpart to the *std::accumulate* method with two important caveats.

The last parameter to *std::accumulate* is the initial value, whereas the last parameter to *parallel_reduce* is the identity value. Consider an array of integers:

```
int data[] = {1,2,3};
```

The call to *std::accumulate(begin(data), end(data), 1)* will return 7, whereas the result of *parallel_reduce(begin(data), end(data), 1)* will be undefined, because 1 is not the right identity value for '+' (0 is).

The other difference is that the binary operation in *parallel_reduce* must be associative, which is not required for the *std::accumulate*.

Due to these subtle but important differences, we decided to **not** call the function *parallel_accumulate* and avoid bugs introduced by developers replacing *std::accumulate* to *parallel_accumulate* wholesale.

Parallel Sorting

PPL offers three versions of parallel sorting:

- **parallel_sort**, which is a general-purpose predicate-based in-place sort based on the parallel Quicksort algorithm.
- **parallel_buffered_sort** , which is a faster general-purpose predicate-based sort that requires additional O(N) space.
- **parallel_radixsort**, which is a stable sort function that requires a projection function that can project elements to be sorted into unsigned integer-like keys. The function requires additional O(N) space.

For workloads where allocating additional memory is acceptable, we recommend using *parallel_buffered_sort*, and for types that are amenable to mapping to integer key, *parallel_radixsort* can offer the best performance.

Picking the right version of parallel sort is based on a number of criteria. Without going into too much detail here, we refer the interested reader to our blog posts (5) and (6) that provide comprehensive analysis of the three available choices.

Overview of Concurrent Objects

Concurrent objects enable concurrent sharing and updating of an object without needing a mutex to serialize operations on it. Except for *combinable*, the concurrent objects presented in the document are containers. The goal of the containers is to permit higher concurrency than would be possible by protecting a non-concurrent container with a mutex. Higher concurrency than a mutex-protected container is possible by use of sophisticated implementation techniques such as internal fine-grained locking[4], non-blocking techniques, or aggregation of operations. The container interface hides these implementation details from the user.

Concurrent containers are not a cure-all for lock-free programming. Each concurrent container protect invariants local to the container, not across multiple containers. Nonetheless we believe they have a place in the toolbox of parallel programmers since they can offer significant performance gains for common scenarios.

The TBB/PPL concurrent containers have interfaces similar to standard STL interfaces, but different enough to permit safe concurrent operation. The differences most involve packaging several STL operations into commonly used transactions. For example, for serial operation of a *std::queue*, a common idiom is:

```
if( !q.empty() ) {
    value=q.front();
    q.pop();
}
```

No matter how the queue is implemented, the sequence is unsafe if *q* is concurrently emptied. The container class *concurrent_queue* has a single operation, *try_pop*, for the idiom. A consequence of designing for common transactions is that the concurrent containers are containers in their own right, not adaptors around other containers.

The permitted concurrency for the objects is not tied to any tasking framework, and thus the containers are generally safe to use with any form of thread-based parallelism, such as *std::thread*, OpenMP parallel regions, or PPL/TBB parallel algorithm templates.

We invite readers interested in implementation issues to study some of the papers in the references, and inspect the open-source TBB distribution, which has implementations of all concurrent objects mentioned except for *concurrent_unordered_multimap* and *concurrent_unordered_multiset*. Likewise, all of the PPL sources ship with the Visual Studio and can be examined.

Note on linearizability: The TBB/PPL concurrent objects have semi-*linearizable* concurrent operations. *Linearizability* is well established in the literature as an important property for reasoning about shared objects. It means that the effects of each operation appear to happen atomically at some time (its *linearization point*) during the invocation of the operation. By semi-linearizable, we mean that effects of the operation *not* involving external functionality, appear atomically. For example, two *push_back* operations on a *concurrent_vector* each must allocate space, and then execute copy constructors. At the linearization point for each *push_back*, space is allocated and the size of the vector is updated, but the copy constructor for the item is invoked later. Hence the copy-construction action, which is external functionality provided by the user, is not guaranteed to be part of the linearization point for the container update. In effect, it is up to the user to ensure that their copy constructor is a second linearization point if they want linearizability. Our semi-linearizability convention seems to be a good tradeoff between simple reasoning and enabling concurrency.

The following subsections discuss particulars of the various concurrent objects.

[4] With care, of course, to never hold a lock while calling user-defined code.

concurrent_queue

Queues are a popular way for threads to communicate with each other. Instances of *concurrent_queue* support arbitrary mixes of concurrent pushes and pops. As in N3353 (11), the interface is value-based. The principal methods for pushing and popping are:

```
void push( const T& source );   // Push value of source
bool try_pop( T& destination ); // Try popping head value into destination
```

Method *try_pop* returns true if successful, false if the queue was empty. Note that the interface permits a non-blocking implementation where consumers never wait[5].

Method *empty* is provided. Method *size* is renamed *unsafe_size* because the size might be reported inaccurately if the queue is undergoing modification.

Given multiple producers or multiple consumers, there are several possibilities for the ordering semantics of a concurrent queue. We chose to guarantee "first-in first-out" according to the "happens before" relation: If *push(x)* happens before *push(y)* and *try_pop(a)* happens before *try_pop(b)*, then the outcome *x=b* and *y=a* is not allowed.[6]

Because of an inherent tradeoff between performance and iterator support, the iterator support for *concurrent_queue* is limited, and intended only for debugging. It enables walking a quiescent *concurrent_queue* so that its contents may be inspected. To emphasize the limited nature (and lack of concurrency safety), the methods for returning the endpoint are named *unsafe_begin* and *unsafe_end*.

There are numerous papers in the literature on implementing concurrent queues (for instance, see (12) and its references). The current TBB/PPL implementation is based on atomic counting and fine-grained locking.

Possible Future Directions for concurrent_queue

Some improvements in efficiency are possible if the number of producers is known to be one or the number of consumers is known to be one. For example, a single-producer single-consumer queue can be implemented on some architectures with extremely low overhead (13). Hence it might be desirable to have a template parameter indicating whether the intended usage limits the number of producers and/or number of consumers to one.

concurrent_priority_queue

Class *concurrent_priority_queue* is to *concurrent_queue* what *std::priority_queue* is to *std::queue*. Items are delivered in priority order instead of first-in-first out order. The priority is specified as a strict weak order, as in *std::priority_queue*.

[5] The current implementation is technically "blocking". Specifically, the consumer of a value can block between the time that the producer of a value starts pushing and the copy of the value is actually constructed in the queue. A fully non-blocking alternative was possible, but the performance cost did not seem worth the benefit.
[6] This is the intended semantics, albeit the TBB documentation says something else, even though the shipping implementations make the guarantee stated in this paper.

There are numerous papers in the literature on implementing concurrent priority queues (14). The current TBB/PPL implementation is based on a traditional heap in conjunction with a flat-combining mechanism (15) that dynamically aggregates requests.

Possible Future Directions for concurrent_priority_queue

Class *current_priority_queue* does not currently support a debugging iterator interface like class *concurrent_queue* does. Ideally such an interface would provide a means to traverse the elements in priority order, though we do not know of a way to implement such efficiently in space and time.

Unordered Associative Containers

The TBB/PPL concurrent containers include concurrent variants of the unordered associative containers in the C++ standard library. They are named *concurrent_unordered_map*, *concurrent_unordered_set*. PPL also has the *multiset* and *multimap* analogs.

The concurrent forms support concurrent *insert*, *find*, and iteration. An example use case for concurrent insert and find is memoization, such as in parallel implementations of the "Hash Life" algorithm (16). Experience with TBB's concurrent_hash_map was that many users requested the ability to walk the table while concurrently inserting new items.

Concurrent erasure is not supported in the current shipping implementations, but could be, though the lack of garbage collection in C++ makes such a feature tricky to use. The problem is that if there is a race between erasing an item and accessing via one of the other operations, the access might end up accessing an item that is being (or is) destroyed. We have investigated protocols for solving the concurrent erasure problem, in which a value is notified when destruction or access is requested, and can resolve the conflict, but so far these protocols have added considerable complexity to the interface and added as much hanging rope as they removed.

The recommended implementation technique is a split-ordered list (17), which enables the concurrent operations to be non-blocking.

concurrent_vector

concurrent_vector enables concurrent *random* access and growth of a densely indexed table. A *concurrent_vector* does not guarantee contiguous storage, that is it does not promise that *&v[i]-&v[j]==i-j*. However, like *std::vector*, it does promise constant-time indexing and furthermore, unlike *std::vector* and like *std::deque*, it does *not* move existing items when more items are appended.

A *concurrent_vector* supports two growth idioms:

- appending a contiguous sequence of *n* elements
- ensuring there are at least *n* elements

Examples for a concurrent_vector v:

- *v.push_back(x)* appends x to the sequence.
- *v.grow_by(n)* appends a contiguous sequence of n default-constructed elements, and returns an iterator pointing to the beginning of the sequence.
- *v.grow_to_at_least(n)* appends enough elements to *v* so that *v.size()>=n*.

For example, the following code appends the contents of a *std::vector u* to a *concurrent_vector v*:

```
copy( u.begin(), u.end(), v.grow_by(u.size()), s );
```

The current implementation uses a two-level indexing scheme that enables constant-time indexing, yet bounds asymptotic memory consumption to no more than twice what the equivalent *std::vector* would use. The first level is a small table of pointers. The second level consists of segments, where each segment is twice the size of the preceding segment.

Dmitry Vyukov has shown that for x86 processors the instruction count for *operator[]* can be as few as 4 instructions (18). With indexing cost that low, an iterator to *concurrent_vector* can be implemented as a mere index; otherwise, it can cache and increment a pointer to its current element and keep the amortized cost of iteration below that of indexing.

Combinable

Class *combinable* is the concurrent object that is not quite like an STL container. It addresses a common pattern in parallel programming: using a set of threads to compute a ``reduction'' value from a dataset, such as a sum, by letting each thread do a subcomputation on a portion of the dataset and then merging subresults. Doing so obviates the need for locking or other forms of inter-thread communication during the local subcomputations. Class *combinable* directly support this common pattern. It is a collection of thread-local values, for which each thread can access its own value, and the values can be easily merged after subcomputations finish.

Here is an example that performs a parallel sum reduction of *f(i)*:

```
combinable<float> sum;
parallel_for( 0, n, [&](int i){
    sum.local() += f(i);
}
sum.combine( []( int a, int b ) {return a+b;} );
```

Method *local* returns a reference to the thread's local value. Method *combine* combines the views using the supplied binary functor. For deterministic results, the functor must be both *associative* and *commutative*.

By default, a new thread-local value is default-constructed; for example, in the above example the values are zero-initialized. There are constructors for *combinable* that permit specifying a default value for a new thread-local value, or specifying a functor for constructing a new thread-local value.

Possible Future Directions for combinable

TBB has an extension, *enumerable_thread_specific*, that exposes a "container of views" interface, thus permitting direct iteration over the set of thread-local values using STL iterator conventions.

Concurrent Containers and STL Compatibility

The design of a concurrent container must balance keeping the API as close to STL as possible against making the container both safe and efficient to use in a concurrent context. This section explores ways that might avoid having to tradeoff one of these goals for the other.

As noted before, the standard STL interfaces are often unsafe to use in concurrent contexts. Furthermore, STL compatibility sometimes precludes more efficient concurrent solutions. For example, value-based designs of concurrent hash tables can be significantly faster than the current split-ordered list implementation in TBB/PPL, in exchange for not having concurrent iterators and references to the stored data.

In PPL and TBB, unless impractical, we chose STL similarity at the expense of performance, to ease learning of the API and to facilitate adding parallelism to existing C++ programs. For the latter, possibility to use containers in serial context in a familiar way is important. For example, iterator support in our containers lets them be used with serial STL algorithms.

Sometimes, however, we deliberately chose to omit or rename methods (e.g. the *unsafe_** methods mentioned earlier) to prevent potential bugs when similarity might imply stronger guarantees than we could provide. Unfortunately, this choice disallows perfectly safe use of the expected STL methods in serial contexts.

Moreover, use of concurrent containers in serial contexts is also likely to be suboptimal performance-wise, and in some cases very inefficient, because STL-compliant concurrent methods must have synchronization that is unnecessary in serial contexts, and thus penalizes adoption of parallelism. Worse serial performance can impact parallel performance as well. Consider the following pattern:

1. In the first phase of an algorithm, multiple threads fill a container in parallel.
2. In the second phase of the algorithm, multiple threads read the container but do not modify it.

Having only a concurrency-safe interface forces phase 2 to pay unnecessary synchronization costs. It is not "pay as you go". For instance, we observed this pattern in OpenCV's use of TBB, where it works around the problem by copying the *concurrent_vector* from phase 1 into a std::vector before doing phase 2.

Ideally, we would like it to be possible to have different implementations of certain container methods with the same name and common semantics, with the right one automatically selected depending on whether the context requires concurrency safety. These methods would operate on the same data structure and access the same data, while ensuring best performance and proper correctness depending on the usage context.

We see at least two possible ways toward this:

1. Exploit a language extension similar to a cv-qualifier but for concurrency, e.g. *concurrent* qualifier (or, maybe, the opposite, *serial* qualifier). Andrei Alexandrescu in (19) attempted to use, or rather abuse, *volatile* for the same purpose. Like the *const* qualifier separates read and write methods and accesses, selecting appropriate overloads in corresponding contexts, the new concurrency qualifier would clearly separate serial methods from concurrent ones for all the purposes: implementation, usage, and documentation. Appropriate rules would be established to disallow calls to serial methods for an instance referred as concurrent, as well as treating all the members of a class as concurrent in context of a concurrent function. There are many issues with this approach that must be resolved, but we believe it is worth considering.
2. Concurrent containers could provide a type-cast interface for switching views (defined as separate classes) for the same physical instance / data layout. For example, the usage pattern described above could be implemented as follows:

```
concurrent_vector<my_type> my_vector;
parallel_for(0, n, fill_in(my_vector));
const auto &my_readonly_vector = serial_cast(my_vector);
parallel_for(0, my_readonly_vector.size(), process(my_readonly_vector) );
```

   It would be programmers' responsibility to ensure that the serial view of a container is not used in a context that allows concurrent modifications. In scenarios when the programmer does not have full control over the context (often the case for library developers), such a cast could be impossible or impractical, causing the programmer to conservatively use the concurrency-safe API.

We realize that even the compiler-based solution cannot resolve all the issues raised here. For example, the tradeoff between best performance and STL compatibility will still remain for concurrent context; and the internal data organization of concurrent containers might still penalize the access in serial context (for example, operator[] of concurrent_vector will still need extra calculations due to non-contiguous storage). But at least in some cases designers and users of concurrent objects would not need to choose between safety, performance, and usability.

Acknowledgment

Anton Malakhov contributed the notions and examples discussed in the section about "Concurrent Containers and STL Compatibility."

References

(1) Lee, E. A. "The Problem with Threads". EECS Department, University of California, Berkeley 2006.
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf
(2) Duvanenko, V. "Parallel Merge Sort". Dr. Dobb's Journal March 24, 2011.
http://www.drdobbs.com/go-parallel/article/229400239
(3) Best Practices in the Parallel Patterns Library
http://msdn.microsoft.com/en-us/library/ff601930.aspx
(4) Patterns and Practices for parallel programming in C++
http://www.microsoft.com/download/en/details.aspx?id=22499
(5) Sorting in PPL
http://blogs.msdn.com/b/nativeconcurrency/archive/2011/01/15/sorting-in-ppl.aspx
(6) How to pick your parallel sort?
http://blogs.msdn.com/b/nativeconcurrency/archive/2011/01/26/how-to-pick-your-parallel-sort.aspx
(7) Intel Threading Building Blocks Tutorial
http://threadingbuildingblocks.org/uploads/81/91/Latest%20Open%20Source%20Documentation/Tutorial.pdf
(8) Intel Threading Building Blocks Reference Manual
http://threadingbuildingblocks.org/uploads/81/91/Latest%20Open%20Source%20Documentation/Reference.pdf
(9) Intel Threading Building Blocks Design Patterns
http://threadingbuildingblocks.org/uploads/81/91/Latest%20Open%20Source%20Documentation/Design_Patterns.pdf
(10) Robison, A. and Voss, M. and Kukanov, K. "Optimization via Reflection on Work Stealing in TBB", HIPS-POHLL 2008.
(11) Crowl, L. *C++ Concurrent Queues*.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3353.html
(12) Moir, M. et al. "Using elimination to implement scalable and lock-free FIFO queues", SPAA '05.
(13) Giacomoni, J. and Moseley, T., and Vachharajani, M. "FastForward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue", PPoPP '08.
(14) Dragicevic, K. and Bauer, D. "A Survey of Concurrent Priority Queue Algorithms", IPDPS 2008.
(15) Hendler, D. et al. "Flat combining and the synchronization-parallelism tradeoff", SPAA '10.

(16) “Hashlife.” http://en.wikipedia.org/wiki/Hashlife

(17) Shalev, O. and Shavit, N. “Split-ordered lists: Lock-free extensible hash tables”, JACM 53(3), May 2006.

(18) Vyukov, D. “Possible optimization of concurrent_vector::operator[]()”, Intel Threading Building Blocks forum, Jan. 18, 2010. http://software.intel.com/en-us/forums/showthread.php?t=71300

(19) Alexandrescu, A. “volatile - Multithreaded Programmer's Best Friend”, C/C++ Users Journal, February 2001. http://www.drdobbs.com/article/print?articleId=184403766