Doc number: **N4195**
Revises: n/a
Date: 2014-10-07
Project: Programming Language C++, Concurrency Working Group
Reply-to: Olivier Giroux <ogiroux@nvidia.com>,
H. Carter Edwards <hcedwar@sandia.gov>

`std::synchronic<T>`

Atomic objects make it easy to write custom synchronization primitives with efficiency problems. One particularly thorny issue is the poor performance of the overall system when it is oversubscribed and/or contention is high. Another is the high power consumed under contention, even when there is no oversubscription of the system.

The following example outlines a very common implementation of the mutex concept, dubbed TTAS, taught in colleges and often encountered online (e.g. on Stack Overflow):

```
struct ttas_mutex {
    ttas_mutex() : locked(false) { }
    void lock() {
        while(1) {
            bool state = false;
            if(locked.compare_exchange_weak(state, true, memory_order_acquire))
                break;
            while(locked.load(memory_order_relaxed)==state)
                ; //see below why this is emphasized
        }
    }
    void unlock() {
        locked.store(false, memory_order_release);
    }
    atomic<bool> locked;
};
```

Unfortunately, this example code is terribly flawed unless the failing path is modified to combine all of: some unmitigated spinning (as shown), randomized exponential back-off (either timed, or simply yielded) and a system call such as `SYS_futex`. Implementations of `std::mutex` get most/all of this right, but the technique is beyond virtually all users and the tuning choices are almost all platform-specific (e.g. HPC versus mobile).

This `ttas_mutex` example is only intended to be illustrative of the problem, not to suggest that users should write `std::mutex` themselves. Rather, users need to be able to implement their own primitives with efficiency comparable with what `std::mutex` achieves.

We propose that waiting operations should be provided by way of synchronic objects that implement the atomic concept and are extended with synchronic (*adj*. "that occur in time") operations on the underlying type:

```
enum notify_hint { notify_all, notify_one, notify_none };
enum expect_hint { expect_urgent, expect_delay };

template <class T>
struct synchronic { //this type conforms to the atomic-type concept

    void store(T,
               memory_order = memory_order_seq_cst,
               notify_hint = notify_all) noexcept;

    T load_when_not_equal(T,
                          memory_order = memory_order_seq_cst,
                          expect_hint = expect_urgent) const noexcept;

    T load_when_equal(T,
                      memory_order = memory_order_seq_cst,
                      expect_hint = expect_urgent) const noexcept;

    void expect_update(T,
                       expect_hint = expect_urgent) const noexcept;

    //todo: add/delete special member functions according to atomic-type
    //todo: add plain load, and read-modify-write function variants
    //todo: add timed 'expect' and volatile function variants
};
```

Note – a complete sample implementation of `synchronic<T>` *for Standard C++11 and some specific versions of Linux and Windows platforms can be found here:* *https://code.google.com/p/synchronic/*. *– End Note.*

Invocations of the synchronic wait operations may block indefinitely unless a notifying operation[1] is performed by another thread. Conversely, synchronic wait operations[2] that return do not guarantee that a notifying operation has occurred because implementations may return whenever the condition is true, irrespective of notifying operations.

Each invocation of `store` *synchronizes-with* invocations of `load_when_`*<cond>* that unblock as a result. Invocations of `store` do not *synchronize-with* invocations of `expect_update`, instead the user must perform an additional operation (typically, an invocation of `load`) on the synchronic object to establish the order of operations on the synchronic object.

Note – implementations of synchronic wait functions may be susceptible to issues with transient values, also known as "A-B-A", resulting in continued blocking of threads that could otherwise be unblocked. Users of synchronic objects should ensure that either transient values do not occur or that the program does not depend on threads unblocking when transient values occur. – End Note.

[1] Variants of atomic operations with memory_order_release, acq_rel or seq_cst and a notify hint other than notify_none.
[2] Variants of atomic read operations that specify a condition, and the expect_update function and its timed variants.

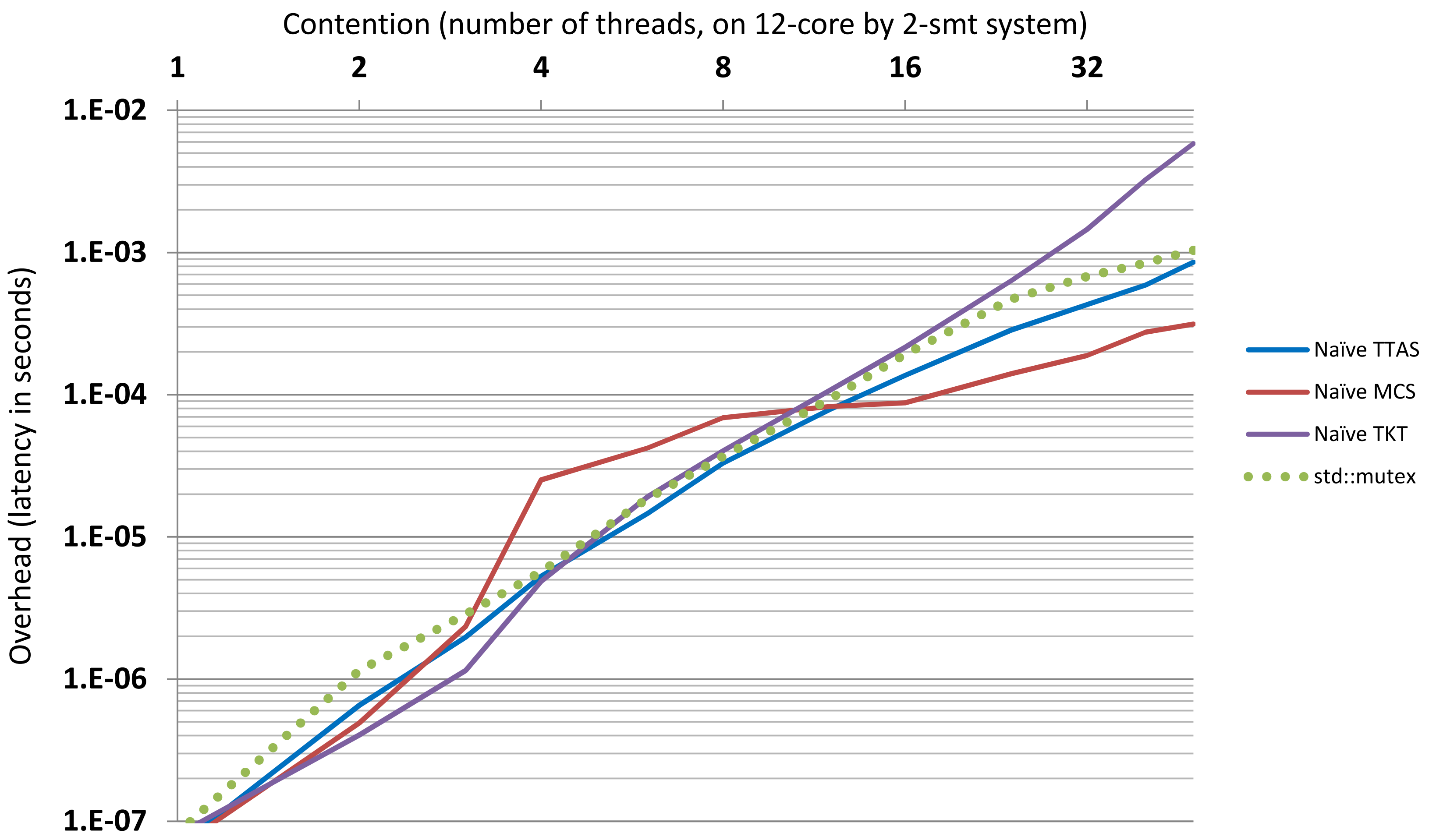
Using a Standard C++ extended with synchronic objects, our earlier example looks mostly the same but now can achieve the same efficiency as `std::mutex`. This is because our toy implementation is now essentially the same as that of the `std::mutex` on many platforms:

```
struct ttas_mutex {
    ttas_mutex() : locked(false) { }
    void lock() {
        while(1) {
            bool state = false;
            if(locked.compare_exchange_weak(state, true, memory_order_acquire))
                break;
            locked.expect_update(state);
        }
    }
    void unlock() {
        locked.store(false, memory_order_release);
    }
    synchronic<bool> locked;
};
```

By changing one atomic object to a synchronic object, the naïve algorithm now matches the performance of the expert algorithm. This result has been demonstrated on several different algorithms, ranging from mutexes to barriers (e.g. those of N3998) and message-passing queues.

See *https://code.google.com/p/synchronic/* for more details.

Latency overhead per critical section, for naïve algorithms using synchronic types, in relation with contention


Contention (number of threads, on 12-core by 2-smt system)
1.E-02
1.E-03
1.E-04
1.E-05
1.E-06
1.E-07
Overhead (latency in seconds)
Naïve TTAS
Naïve MCS
Naïve TKT
std::mutex