C++11 Concurrency and Multithreading
For Hedge Funds & Investment Banks:
Concurrency and Multithreading with C++11
for Thread Management & Data Sharing Between Threads

Yogesh Malhotra, PhD

www.yogeshmalhotra.com

Global Risk Management Network, LLC
178 Columbus Ave, New York, NY 10023, U.S.A.
www.FinRM.org

May 6, 2013
Concurrency and Multithreading with C++11
for Thread Management & Data Sharing Between Threads

Abstract

The current investigation aimed to understand primary factors motivating adoption of concurrency and multithreading capabilities by hedge funds, investment banks and trading desks and future trajectory of such implementations. Primary focus was on examining if the C++11 Standard represents a plausible basis for standardization among above institutions for concurrency and multithreading needs. Technical focus was on understanding how C++11 addresses concurrency and multithreading challenges as known in prior C++ standards and alternatives. Specific investigation, development, and demonstrations focused on two key aspects of concurrency and multithreading: thread management capabilities and protected thread data sharing capabilities. In each of the two categories, more than a dozen key technical issues representing most relevant concurrency and multithreading capabilities were analyzed. Based upon the analysis, it was concluded that the C++11 Standard represents a viable future trajectory of technical standardization and development of concurrency and multithreading for hedge funds, investment banks and trading desks. It was also recognized that greater technical sophistication does not lessen the need for programming discipline: rather it becomes even more critical in ensuring simplicity and transparency of technical design.
Summary

The purpose of the investigation was to examine if the C++11 Standard represents one such plausible basis for standardization among hedge funds, investment banks and trading desks for their concurrency and multithreading (C&M) needs. Review of recent applied case studies about C&M implementations reported by representative firms enhanced prior experiential understanding gained while working with such firms. Given primary focus on C&M related thread management capabilities and thread data sharing capabilities, two levels of technical review were done.

The overall technical focus was on broader capabilities of the C++11 Standard which represents the first C++ standard to truly acknowledge multi-threading earlier implemented by diverse APIs and compiler-specific extensions. Within broader context of the C++11 Standard, primary technical focus was on thread management capabilities and protected thread data sharing capabilities such as those available in Boost libraries. Based upon a review of the Standard as well as multiple presentations, publications, and related research materials by most relevant top technical experts, specific thread management and thread data sharing capabilities in C++11 were identified for analysis.

The above specific implementation focused capabilities were found interesting in two respects: either they solved C&M problems inherent in earlier technical C++ implementations that are platform and implementation dependent, or they provided enhanced capabilities not feasible earlier with prior C++ standards. The result of analysis was a survey of more than 25 selected C&M implementation related technical capabilities that seem most relevant to the future C&M needs of hedge funds, investment banks and trading desks.
Introduction

Background

"Soon, programs that are not multithreaded will be at a disadvantage because they will not be using a large part of the available computing power."

- Vance Morrison, Microsoft .NET Runtime Compiler Architect

Good design fundamentals are similar across sequential and multithreaded programs in how they protect program invariants that are needed elsewhere. Yet protecting such program invariants gets complicated in the multithreaded case. For example, it may not be apparent how other threads may be changing memory while a specific thread is executing its updates on same memory. Consequently, race-free multithreaded programs require programming discipline such as in appropriate use of locks for protecting memory while ensuring simplest design requiring minimal locks. The C++11 standard, and in particular its C&M multi-threading capabilities aim to help programmers and developers achieve such goals. New multicore architectures enable even more sophisticated C&M implementations for which C++11 is particularly suited. Such architectures enable greater efficiencies as they are neither limited by processing speed of a single core nor entail their significant overhead of context switching.

Industry Case Studies

Industry case studies of a representative set of hedge funds, investment banks and trading desks provide a perspective about increasing interest in such C&M and related multi-threading capabilities. In execution of financial market trades, where differential of microseconds can mean significant gains or losses, such technical capabilities are really appreciated. According to one recent industry report (LowLatency 2012) on low latency trading, the ‘tick to trade’ benchmark in 2012 was of 20 µs, was expected to be 10 µs this year and in the range of 1 µs for some trades next year. Low latency enabled
by C&M and related multi-threading capabilities is thus critical for investors and traders in managing risk as another recent report (High Frequency Trading Review 2011) on high-frequency trading underscores. A CTO of concurrent programming framework for a UK trading exchange notes for instance (HFT Review 2011): “In the retail space, latency is very critical to people managing their risk when getting out of positions by getting a closing order onto an order book quickly… We discovered that putting things onto the queue and taking things off the queue was taking the most time, introducing lot of latency into the process.” As evident from such experience reports from the field, text book approaches often may not yield critical execution efficiencies necessary for survival in such dynamic information intensive environments.

**Scientific & Technical Research**

Complementary perspective about the C&M challenges and opportunities was developed from a review of scientific and technical research literatures. A just released technical report on multithreading from Columbia University (2013) underscores the critical importance of multicore architectures in the financial firms discussed above (emphasis added): “two technology trends have made the challenge of reliable parallelism more urgent… the rise of multicore hardware... developers *must* resort to parallel code for best performance on multicore processors...[and] our accelerating computational demand.” Similar sentiment is evident in a University of Cambridge (2013) technical report released last month which observes that (emphasis added): “transition to multi-core processors has yielded a fundamentally new sort of computer... Software can no longer benefit passively from improvements in processor technology, but *must* perform its computations in parallel if it is to take advantage of the continued increase in processing power.” The above reports echo the emphasis of an earlier computer science research article (Sodan et al. 2010) on multi-threading and
parallelism on computationally powerful cores for numeric applications that are characteristic of financial firms we have discussed. Many such reports emphasize the much needed technical advancements given C&M challenges observing for instance that such programs are difficult to write, test, debug and analyze. A computer science journal article (Hummel 2013) of current month reiterates the critical need for embracing multi-core CPUs and GPUs noting that any future viable C&M standard will need to address C++ given its dominance for high-performance computing and C++11 is one such standard.

**Scope**

The focus of the current C&M focused report is on thread management capabilities and thread data sharing capabilities in C++11. These two aspects even though central to the ‘thread-aware memory model’ (Anthony 2012) of the C++11 Standard are not its only C&M capabilities. In addition to C&M capabilities proposed for C++11 but yet to be formally integrated from libraries such as Boost (Anthony 2012), other C&M aspects of C++11 include the new threading memory model, the new multithreading support library, enhanced C++ memory model and operations on atomic types, lock-based and lock-free concurrent data structures, concurrent code, and, advanced thread management. Those aspects of C++11 C&M are expected to be the focus of continuing research and practice development focused on trading and financial risk management for financial firms such as hedge funds, investment banks and trading desks.

**Limitations**

Additional limitations of the current investigation are related to the narrow but critical C&M focus on thread management capabilities and thread data sharing capabilities.
C++11 standard has much broader implications beyond the technical focus of the current report which may not represent even the metaphorical tip of the iceberg. For instance, there are specific C++11 standard issues relevant to Java or Python specialists or to specific platforms such as Microsoft or Intel that are deemed beyond the scope of the current report. In addition, there is already ongoing work in progress on standardization for C++ 2014 and beyond. Within broader frame of evolution of technologies, other alternative paradigms such as inherently concurrent Erlang and emerging ‘big data’ capabilities based on cloud computing may also be relevant. Related reports and presentations reviewed are not included here for apparent reasons.

Discussion and Results

C++11 Standard

As C++11 supports multiple hardware threads, it standardizes C&M in that there is no more need for writing platform-specific APIs or extensions. Programs developed using C++11 will run across diverse platforms without modification, hence it synchronizes software development with the latest multi-core multi-threading technology hardware capabilities. Instead of an illusion of concurrency based on context switching and time slicing on a single CPU, it enables real hardware concurrency wherein multiple single-threaded processes can run simultaneously.

Before C++11, the C++ standard neither acknowledged threads exist nor defined a thread aware memory model: making multithreaded applications impossible without compiler-specific extensions that sacrificed efficiency and increased latency. Nor did prior C++ standard support IPC: so applications requiring multiple processes had to rely perforce on platform-specific APIs. Regardless, C++11 demands the discipline consistent with more complex inter-thread sharing of data and address space while lowering thread management overhead given lack of inter-thread boundaries.
shared memory between threads in C++11, programming discipline is necessary to ensure consistency of data seen by each thread. Next we discuss how thread management overhead is minimized by the new capabilities in C++11 followed by how C++11 enables protected inter-thread data sharing.

**Thread Management in C++11**

The current section provides a summary of key thread management capabilities of interest in C++11 and how they are implemented. Given applied implementation focus, current focus on thread management capabilities and subsequent focus on protected inter-thread data sharing highlight specific capabilities using code illustrations specially created for this presentation and inspired by work on Boost (Anthony 2012) and its adaptations for the C++11 standard (Stroustrup 2013, Myers 2013, Morrison 2005).

1. **Starting a thread as a function and controlling its exit**

As highlighted above, the extra `#include <thread>` declares the standard library containing related classes and functions. Each new thread is expected to have its own *initial function* as shown above for the thread constructor calling hello function. The original thread has main as its initial function. Future of the new thread is controlled by a call such as `join()` specifying original thread to wait for the new thread to return.
2: Starting a thread as class method & as lambda expression

For more fine-tuned control over what the new thread does, the constructor may be called from within a class declaration as shown next. An alternative to class declaration on left is to use what C++11 calls as lambda expression, a notation that allows more concise specification while allowing more fine-grained control over parameters.

3: Allowing launched thread to run on its own

detach is used instead of join to detach the new thread from the original thread so that it runs in background on its own like a Unix daemon process. A call to detach or join is necessary before constructed thread is destroyed. A detached thread cannot be joined.
4: Preventing reference by launched thread to a dangling reference

Following code shows the detached thread trying to reference a variable that has already been destroyed as the original thread with that local variable could have already terminated. Such dangling reference problem can be prevented by copying the local variable into the new thread before it is detached or ensuring to use join instead of detach so that the local variable is available.

```cpp
struct dostuff
{
    int& i;
    dostuff(int& i_);i(i_)
    void operator()()
    {
        for(unsigned l=0;l<1000000;++l)
        {
            incrementvar(l);
        }
    }
};

void trouble()
{
    int some_local_value=0;
    dostuff my_dostuff(some_local_value);
    std::thread t(my_dostuff);
    t.detach();
}
```

Trouble: trouble launches my_thread to run on its own and exits thus destroying defined value of my_dostuff, parameter of my_thread.

5: Effectively managing exceptions thrown by launched thread

Following code demonstrates use of try-catch-throw as in C++ to capture an exception while also taking care of normal exit. When such an exception needs to be addressed, it is important to ensure that all potential exit paths of the thread are accounted for.

```cpp
struct dostuff;
void trouble()
{
    int some_local_value=0;
    dostuff my_dostuff(some_local_value);
    std::thread t(my_dostuff);
    try
    {
        do_stuff_in_current_thread();
    }
    catch(...)
    {
        t.join();
        throw;
    }
}
```
Using enhanced Resource Acquisition is Initialization (RAII)

RAII is an important capability further enhanced in C++11 and helps prevent deadlocks. RAII, often used for controlling mutex locks, ensures that resources acquired during initialization of objects are released with destruction of then same objects even when program terminates because of error conditions. For example, code that locks the mutex includes the logic to release the lock when the specific object goes out of scope.

```
class thread_guard
{
    std::thread & t;
public:
    explicit thread_guard(std::thread & t): t(t) {}
    ~thread_guard()
    {
        if(t.joinable())
        {
            t.join();
        }
    }
    thread_guard(thread_guard const&)=delete;        //copy constructor
    thread_guard& operator=(thread_guard const&)=delete; //copy ass. constructor
};
```

Above code shows local RAII objects destroyed in reverse order of construction when code execution ends or if the code throws an exception. The test of the destructor when calling join ensures that a thread is joinable before doing so. Declaring copy constructor and copy assignment constructor as deleted ensures that they are not generated by the compiler and any attempt to use their specific object will generate compile time error.

Opening new document using detached thread

Following code shows how another instance of a current process such as editing of a document can be started. The code also shows how besides passing the name of the thread constructor function, the name of the new file to be opened can also be passed as
a parameter. Same function in use for editing current document can be used for the new thread and after the new thread to open a new document is launched it is detached.

```cpp
void edit_doc(std::string const& doc_name)
{
    open_doc_and_display_gui(doc_name);
    while(!done_edit_doc())
    {
        user_command cmd=get_user_input();
        if(cmd.type==open_new_doc)
        {
            std::string const new_name=get_doc_from_user();
            std::thread t(edit_doc,new_name);
            t.detach();
        }
        else
        {
            process_user_input(cmd);
        }
    }
}
```

8: Handling parameter passing in threads

```cpp
void func(int i,std::string const& s);
void trouble(int a_param)
{
    char buffer[1024];
    sprintf(buffer,"%i",a_param);
    std::thread t(func,3,buffe); 1
    t.detach();
}

void no_trouble(int a_param)
{
    char buffer[1024]; 1
    sprintf(buffer,"%i",a_param);
    std::thread t(func,3, std::string(buffer)); 2
    t.detach();
}
```

Above code in the left panel shows new thread of execution calling a function and a pointer to a local variable passed to the constructed thread. However, the calling function trouble may exit before buffer is converted to string thus resulting in undefined behavior. Such potential problem can be eliminated by passing specific parameters to handle explicit prior conversion of buffer to string thus preempting and preventing the missing reference problem as shown in the code in the right panel.
9: Ensuring object reference is copied and not the copy of object

Following code shows the use of std::ref to ensure that reference to data is passed rather than the copy of the object. std::thread, being oblivious of function arguments, may otherwise simply copy given values rather than their updated version as shown on left.

```
void update_data_for_object (object_id w, object_data& data);
void trouble_again(object_id w) {
    object_data data;
    std::thread t
        (update_data_for_object,w,data);
    display_status();
    t.join();
    process_object_data(data);
}
```

‘Wrapping’ the arguments to be referenced in std::ref ensures that the object reference is correctly passed and not the copy of the object.

10: Comparing std::thread with C++ smart pointers

```
void process_dynamic_object(std::unique_ptr<dynamic_object>& p);

void process_dynamic_object(std::unique_ptr<dynamic_object>& p)
    (new dynamic_object);
```

std::unique_ptr is a smart pointer for automatic memory management used for objects for which memory is dynamically allocated. Following the RAII principle, it has only one instance pointing to a specific object which is deleted when that instance is destroyed. As shown above, resource ownership of threads can be transferred between such instances for thread execution by using std::move. Several classes of C++11 Standard Thread Library allowing similarly to programmatically transfer ownership between objects, std::thread being one of those classes.
11: Transferring ownership among threads

```cpp
void a_func();
void another_func();
std::thread t1(a_func);     //new thread started, associated with t1
std::thread t2=std::move(t1); //thread ownership passed from t1 to t2
// new thread started & auto-transfer to t1
// t3 constructed with no associated thread, default
// ownership passed from t2 to t3
// calls std::terminate() as t1 already has other thread.
```

Above code shows how ownership of thread execution can be transferred across different std::thread instances. New thread associated with t1 transfers its execution ownership to t2 when t2 is constructed by using std::move(). t1 is then reinitialized as a new thread. When the last line of code attempting to transfer execution ownership from t3 to t1 executes, t1 is found to have a prior associated thread, therefore program terminates. A thread must be either detached or joined for consistency; similarly execution ownership can’t be replaced by assigning new value to its std::thread object.

12: Using multiple threads with hardware concurrency

```cpp
template<typename iterator, typename T>
struct accumulate_block
{
  void operator[](iterator first, iterator last,T &result)
  {
    result=std::accumulate(first,last,result);
  }
};
template<typename iterator, typename T>
T parallel_accumulate(iterator first, iterator last, T init)
{
  unsigned long const length=std::distance(first,last);
  if(!length)
    return init; // if input range empty, return initial value
  unsigned long const min_per_thread=12; // Min block size
  unsigned long const max_threads=
    (length-min_per_thread)/min_per_thread;
  // a) Max No. of Threads = No. of elements / Min Block Size
  // b) No. of threads that can run truly concurrently: cores
  std::thread::hardware_concurrency();
  unsigned long const num_threads=
    std::min(hardware_threads,5); // Minimum of calculated a) and b) Threads.
  unsigned long const block_size=length/num_threads;
  // Number of entries for each thread to process
  std::vector<T> threads(num_threads); // Intermediate data
  // Subtract one pre-existing thread, find threads to launch.
  iterator block_start=first;
  // Following loop launches threads based on above count
  for(unsigned long i=0;i<num_threads;++i)
  {
    iterator block_end=block_start;
    std::advance(block_end,block_size);
    threads[i]=std::thread(accumulate_block(block_end,block_start,T()));
    block_start=block_end;
  }
  // std:ref passes reference if the object cannot be copied.
```

As C++11 allows real hardware concurrency, a task can be split for execution among multiple threads while using the specification std::thread::hardware_concurrency() to
determine most optimal hardware thread count that can be supported by the system. Related classes shown above are used for thread count computation using iterators.

**13: Using thread IDs to monitor & control division & execution of tasks**

As shown below `std::thread::id` is the unique identification of the thread that can be used for monitoring and controlling execution and can be copied or compared. `std::this_thread::get_id` can be used by a thread to store its own identification before launching other threads for any subsequent comparisons with original identification.

```cpp
std::thread::id main_thread;
void main_program()
{
    if (std::this_thread::get_id()==main_thread)
    {
        do_main_thread_work();
    }
    do_spawn_thread_work();
}
```

**Data Sharing Between Threads in C++11**

“All code in the program must protect any invariants that other parts of the program need… If two threads process code or data simultaneously, bugs which occur because of bad timing between threads are called races.” - Vance Morrison, Microsoft .NET Runtime Compiler Architect

The current section provides a summary of key capabilities related to protected data sharing between threads in C++11 and how they are implemented. Preventing broken invariants is perhaps one of the most important concerns in multi-threading.

**1: Preventing broken invariants in data sharing between threads**

Multi-thread concurrency allows for easily and directly sharing data between threads thus minimizing the overheads typically associated with multi-process concurrency. The related challenge of multi-thread concurrency is also that of shared modifiable data: invariants broken during modifications can create major headaches. Invariants denote statements that are always true about a particular data structure but during
modification or updates don’t hold temporarily. As many of the protected inter-thread data-sharing capabilities in C++11 relate to these specific aspects, it is discussed here separately before specific data sharing issues that are relevant in specific circumstances. Race conditions in C++11 can be prevented by using wrapper for data structures, using lock-free programming, and using the C++11 Standard Thread Library std::mutex which is the most common mechanism for protecting shared data.

2: Using mutexes to prevent race conditions in C++11

```cpp
#include <list>
#include <mutex>
#include <algorithm>
std::list<int> my_list; // global variable
std::mutex my_mutex; // global mutex
void add_to_my_list(int new_value)
{
    std::lock_guard<std::mutex> guard(my_mutex); // mutually exclusive
    my_list.push_back(new_value);
}

bool my_list_contains(int value_to_lookup)
{
    std::lock_guard<std::mutex> guard(my_mutex); // mutually exclusive
    return std::find(my_list.begin(), my_list.end(), value_to_lookup)
        != my_list.end();
}
```

The major headache of broken invariants can be controlled with mutex, abbreviation for ‘mutually exclusive’, as in mutually exclusive access of thread to data when it is being modified. Mutexes help prevent races by using lock and unlock to restrict access to other threads while a thread is active updating specific memory or while an invariant may be broken. Essentially, the thread locks the mutex before accessing to update a data structure and when it is done updating, it unlocks the mutex. However, in C++11 std::lock_guard provides a preferred alternative to mutex lock and unlock. std::lock_guard takes care of locking in the constructor and unlocking in the matching destructor thus avoiding the need for unlocking explicitly which may pose problem if a thread doesn’t execute unlock in case something goes wrong after turning on the lock.
3: Preventing runtime functions from passing arguments to protected data

Following code shows an instance where a runtime function passes arguments to protected data and illustrates the need for discipline in programming design.

```
class my_data
{
    int a;
    std::string b;
public:
    void do_stuff()
    {}
};
class data_wrapper
{
    private:
    my_data data;
    std::mutex m;
public:
    template<typename Function>
    void process_data(Function func)
    {
    std::lock_guard< std::mutex > l(m);
    func(data);
    } // lock guard secured process_data
}; // calls user provided func.

my_data* unprotected;

void harmful_function(my_data& unprotected)
{
    unprotected=&protected_data;
}
data_wrapper z;

void trouble() // trouble can pass harmful function
{
    // to bypass protection...
    z.process_data(harmful_function);
    unprotected->do_stuff(); // ...and call do_stuff...
} // while mutex unlocked.

int main()
{
    trouble();
}
```

Mutex can’t be of help if a using member function returns reference to protected data. The key caution for the designer is to not pass any pointer or reference to protected data outside the scope of lock as it is beyond control of the associated mutex. Hence, the capabilities for multi-threaded protected data sharing in C++11 are critically dependent upon the designer’s discipline as well as sophistication in using them carefully.

4: Preventing stack-associated interface issues from causing race conditions

The following code shows an example to illustrate the subtle issues that distinguish multi-threading environment from single-threaded environment thus requiring greater discipline on part of the designer. When a program thread checks a stack condition, for instance empty() or size() which may trigger associated pop() or push() action, single-threaded environment is not concerned about other threads. In contrast, in multi-threaded environment, after one thread may have checked the shared stack using empty() or size(), another thread may execute a pop() or push(), so that prior
information is invalid for the first thread to act upon. For instance, the pop() action of second thread may cause the shared stack to become empty and if the first thread calls top(), it may result in undefined behavior [as it would in single-threaded mode]. One possible solution to the above situation shown in the following code is to pass a reference and return a pointer to the popped element.

```cpp
template<typename X, typename Container=std::deque<X>>
class stack
{
    public:
        explicit stack(const Container&);
        explicit stack(Container& = Container());
        template <class Alloc> explicit stack(const Alloc&);
        template <class Alloc> stack(const Container& const Alloc&);
        template <class Alloc> stack(Container&&, const Alloc&);
        template <class Alloc> stack(Container&&, const Alloc&&);
        bool empty() const;
        size_x size() const;
        X& top();
        X const& top() const;
        void push(X const&);
        void push(X&);
        void pop();
        void swap(stack&&);
};
```

Can’t rely upon empty() and size()…
... if other interim threads did push() or pop().

Not at all reliable for shared stack…
... Another thread may pop() between current thread’s check for empty() before call to top().

Solution: Pass in a reference & return pointer to popped element.

5: Using std::lock to prevent deadlock conditions in C++11

When two or more mutexes are locked for some operation, each thread may be waiting for the other thread to release the mutex which may cause deadlock, the biggest problem in using mutexes. The problem may be resolved in some cases by locking the mutexes in the same order however this will not work if for example mutexes are protecting separate instances of the same class. For preventing deadlock in such situations, C++11 Standard Thread Library provides the std::lock library. In addition, other solutions for preventing deadlocks include avoiding nested locks, using a lock
hierarchy, and avoiding user supplied codes in lock. The specific example of using std::lock with std::lock_guard is discussed next.

**6: Using std::lock and std::lock_guard to prevent deadlock**

```cpp
// Example of using std::lock and std::lock_guard

class my_stuff
{
    void swap (my_stuff& lhs, my_stuff& rhs)
    {
    }
}

// Example of using std::unique_lock and std::defer

class Z
{
    private:
        my_stuff my_thing;
        mutable std::mutex m; // thread-safe
    public:
        Z (my_stuff const & sd): my_thing (sd) {}

        friend void swap (Z & lhs, Z & rhs)
        {
            if (&lhs == &rhs)
                return;
            std::lock (lhs.m, rhs.m);
            std::lock_guard < std::mutex > lock_a (lhs.m, std::adopt_lock);
            std::lock_guard < std::mutex > lock_b (rhs.m, std::adopt_lock);
            swap (lhs.my_thing, rhs.my_thing);
        }
    }
}
```

Above code shows std::lock() locking the two mutexes and one std::lock_guard instance being constructed for each mutex. As noted, the std::adopt_lock parameter indicates to std::lock_guard that mutexes are already locked and they should adopt ownership of existing lock on mutex and not attempt to lock mutex in the constructor. The above arrangement allows mutexes to correctly unlock in case of both a normal exist or where an exception condition occurs.

**7: Using std::unique and std::defer to prevent deadlock**

std::unique_lock allows instances to relinquish locks anytime with unlock(). This may be required in cases such as in case of deferred locking or transfer of lock ownership from one scope to another as shown on the next page. Its flexibility in relaxing the
invariants has to bear the cost of storing and updating related information. Because of such cost of flexibility, it should be used wisely over the std::lock_guard alternative.

```cpp
class my_stuff
{
    void swap(my_stuff & lhs, my_stuff & rhs)
}()

std::unique_lock instance
doesn't always own its associated mutex
- Flexibility: relaxing the invariants
- Price : Store & Update flags information.

std::defer_lock so that mutex remains unlocked on construction
std::defer_lock leaves mutexes unlocked

std::lock mutexes locked here
```

8: Transferring ownership of mutexes between std::unique instances

```cpp
std::unique_lock<std::mutex> get_lock()
{
    extern std::mutex my_mutex;
    std::unique_lock<std::mutex> lk(my_mutex);
    prepare_data(); // since lk automatic variable,
    return lk;      // compiler calls move constructor
}
void process_data() // process_data transfers ownership
{
    // into its std::unique_lock
    std::unique_lock<std::mutex> lk(get_lock());
    do_stuff(); // call to do_stuff() uses data
    // without alteration by other thread
```

Ownership transfer of mutexes between std::unique_lock instances can be done by moving around those instances automatically or explicitly. Such transfer is automatic if source is lvalue, a real variable or reference thereof persisting beyond single expression. It is explicit when using std::move() if source is rvalue, a temporary variable. A possible use is to allow a function to lock a mutex and transfer ownership of that lock to the caller which can then perform additional actions protected by the same lock.
9: Locking mutexes only for minimal time to access shared data

Unless a lock is used to protect access to a file, it should be held for minimum possible time to perform the required operations as shown in the illustrative code below.

```cpp
void getdata_processdata()
{
    std::unique_lock<std::mutex> some_lock(the_mutex);
    my_class data_to_process=get_next_data();
    some_lock.unlock(); // unlock the mutex
    result_type result=process(data_to_process); // I/O while open lock
    some_lock.lock(); // lock the mutex
    write_result(data_to_process,result);
}
```

Hence, locking of a mutex should be for accessing the shared data and the lock should be released when doing processing of data such as I/O. Similarly, other time-intensive activities such as waiting for I/O or acquisition of another lock should be minimized while holding a lock.

10: Using compare for holding one lock at a time in contrast to using swap

Swap operation shown in item 7 above requires concurrent access to both objects and hence required locking the two mutexes together. Instead, as shown in code below if comparison of two objects is needed and not a swap, compare operator can be used that holds only lock for the object for which the data is being copied for comparison. That being said, it must be noted that if the lock is not being held for the entire duration of the operation, other threads may have changed the data for objects being compared. Hence, one downside of not holding the lock for the entire operation is that doing so may make them more susceptible to race conditions.
Protecting shared data in initialization while preventing race condition

C++ Standard Library provides std::once_flag and std::call_once flags to be used instead of locking a mutex and explicitly checking a pointer anticipating that it would have been initialized by some thread by the time std::call_once returns. std::call_once has
lower overhead than explicit use of mutex, particularly when initialization is already
done. std::call_once thus helps prevent race conditions at the time of initialization.

**12: Doing initialization exactly on one thread for static variable**

When a static variable is initialized for the first time, control passes through its
declaration as shown in the code below. If multiple threads call the initialization
function, race condition can occur. C++11 solves this problem as such initialization can
happen on exactly one thread while other threads are locked out.

```cpp
class some_class;
some_class&
get_some_class_instance()
{
    static some_class instance;
    return instance;
}
```

**13: Single writer & multiple readers for rarely updated data**

```cpp
#include <map>
#include <string>
#include <mutex>
#include <boost/thread/shared_mutex.hpp>
class table_entry;
class table_cache
{
    std::map<std::string, table_entry> entries;
    mutable boost::shared_mutex entry_mutex;
    public:
        table_entry find_entry(std::string const& domain) const
        {
            boost::shared_lock<boost::shared_mutex> lk(entry_mutex);
            // protection for shared read-only access
            std::map<std::string, table_entry>::const_iterator
            it = entries.find(domain);
            return (it==entries.end())? table_entry():it->second;
        }
void update_or_add_entry(std::string const& domain,
                         table_entry const& table_details)
{
    std::lock_guard<boost::shared_mutex> lk(entry_mutex);
    // provide exclusive access during updates
    entries[domain]=table_details;
}
```

"Where threads read shared structures frequently, but write to them infrequently,
reader-writer locks can be used to keep the number of locks in the system low. This
type of lock has one method for entering for reading and one for entering for
writing. The lock will allow multiple readers to enter concurrently, but writers get
exclusive access. Since readers now don’t block one another, the system can be
made simpler (containing fewer locks) and still achieve the necessary
concurrency.” Vance Morrison, Microsoft
For rarely updated data, cache accessible by multiple threads needs protection during update by a writer thread so that threads don’t see a broken invariant data structure. By using Boost, a new type of reader-writer mutex is feasible as shown above to allow exclusive access by a writer thread or concurrent access by multiple reader threads. For the single writer’s exclusive update, std::lock_guard <boost::shared_mutex> and std::unique_lock<boost::shared_mutex> are used for locking just like std::mutex. For the remaining multiple reader threads, shared access is enabled by use of boost::shared_lock<boost::shared_mutex> which allows shared lock at same time by multiple threads just like std::unique_lock allows for a single thread.

**Conclusion & Recommendations**

Industry cases and scientific research reviewed for the current investigation establish the need for high speed analysis and trading execution with microsecond precision as motivations for adoption of C&M capabilities by financial firms. Based upon research supporting the need for analyzing C++11 Standard as a plausible basis for standardization among such firms, current investigation focused on its C++11 related C&M capabilities. Specific technical focus of the analysis was on thread management capabilities and thread data sharing capabilities that are central to such C&M implementations. In each of the two categories, more than a dozen key technical issues representing most relevant C&M capabilities were analyzed. Based upon the analysis, it was concluded that the C++11 Standard represents a viable future trajectory of technical standardization and development of concurrency and multithreading for hedge funds, investment banks and trading desks. It was also recognized that greater technical sophistication does not lessen the need for programming discipline: rather it becomes even more critical in ensuring simplicity and transparency of technical design.
The focus of the concluded investigation was on thread management capabilities and thread data sharing capabilities in C++11. Beyond these two central aspects of the C++11 Standard’s ‘thread-aware memory model’, additional relevant issues of future analysis include the new threading memory model, the new multithreading support library, enhanced C++ memory model and operations on atomic types, lock-based and lock-free concurrent data structures, concurrent code, and, advanced thread management. Those C++11 C&M issues are expected to be the focus of continuing research and practice development focused on financial firms. In the broader frame of alternative technologies that can enable C&M capabilities, alternative paradigms such as Erlang which is used for high-frequency trading by Goldman Sachs and emerging ‘big data’ capabilities based on cloud computing are also of additional interest.
References