CHAPTER 3
Parallel C++

In this chapter, we introduce the C++11 Standard Threading Library, discuss its main components, and develop a basic thread pool. We reuse that thread pool to perform concurrent calculations in the rest of the book.

In order to understand concurrent programming and make sense of this chapter, we must first appreciate what are the different pieces involved, how they interact, and who is responsible for their management.

- **Cores** are the hardware processing units that execute sequences of instructions. Typical laptops and PCs have two to six physical cores today, typical workstations have eight cores and up.
  Intel's hyperthreading technology runs multiple (two) *hardware threads* per core. Hardware threads, also called *logical cores*, accelerate execution by switching logical cores on physical cores when beneficial, in accordance with a logic hard coded on the chip. From the application's point of view, hardware threads are seen and treated as physical cores.
  Performance benefits are entirely dependent on the number of cores on the executing machine. Multi-threaded applications run no faster (actually, slower) on single-threaded computers.[1]

- **Threads** are the sequences of instructions executed on the machine's available cores. Every application creates at least one thread, called *main thread*, which starts when the application launches. Multi-threaded applications are those who explicitly create additional threads, with a syntax explained in Section 3.2.
  The threads created by an application are distributed, or *scheduled* for execution, by the OS, and outside of the application's control,[2] over the available cores.[3]
  Performance is not the only reason for building multi-threaded applications. Other reasons include multi-tasking (the ability to do several things at the same time is in itself valuable, even when it is the same CPU that performs these tasks: otherwise we couldn't run calculations on Excel and document them on Word simultaneously) or separation of concern (in a video game, the movements of the player and their enemies may be coded on separate threads, which clarifies code, and *may* increase performance if executed on multiple cores).
  When we have more threads than cores, the OS emulates concurrency by sharing CPU time, quickly switching execution threads, evidently without a performance benefit. On the contrary, performance typically suffers from context switching. The application may create a lower or higher number of threads than there are available cores, although performance benefits are limited to the number of cores.
  The creation of threads involves an overhead, and the management of a large number of threads may slow down the OS. For these reasons, we must carefully choose the number of threads and refrain from starting and ending them repeatedly.
- **Tasks** are the logical slices of a parallel algorithm that may be executed simultaneously, like the transformation of a subset of the source collection, or the processing of a batch of scenarios in a Monte-Carlo simulation. The number of tasks may be very large, and it is the application's responsibility to schedule their execution on the application's threads.

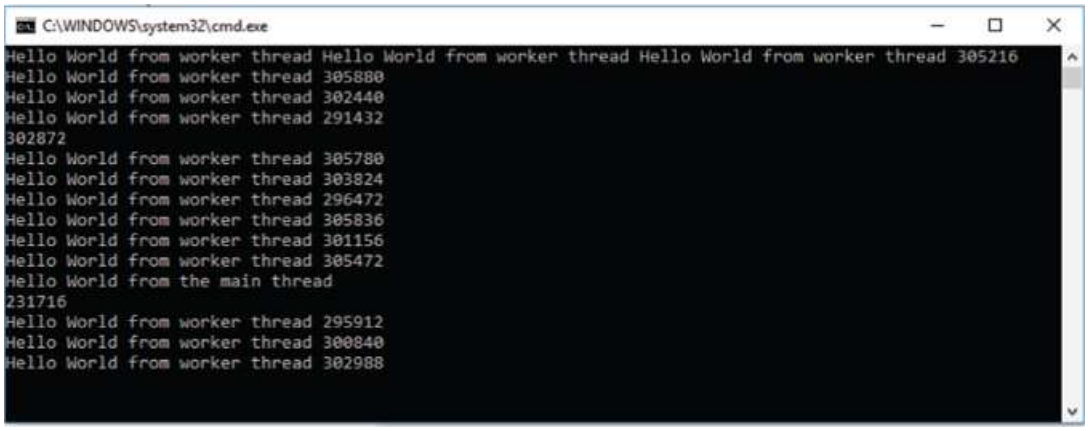
- **Thread pools** are constructs that centralize and encapsulate the scheduling of tasks over threads in a multi-threaded application. Thread pools are a best practice that help produce modular, readable code and improve performance. They are not strictly speaking necessary. The management of threads and tasks in the application could be conducted manually, although this is not recommended. The best design for a thread pool depends on the application's specific needs. Therefore, the C++ standard library does not provide thread pools, although it provides all the building blocks for a flexible, custom implementation. Financial applications typically only require basic scheduling, and the simple design implemented in Section 3.18 is sufficient for our needs. Williams develops in [15] more sophisticated thread pools, and introduces advanced scheduling techniques like *work stealing*. That publication, written by a developer of the Standard Threading Library, provides a more comprehensive discussion. It is a recommended reading for anybody wishing to earn a deep understanding of concurrency and parallelism. Our coverage is more condensed, concurrency in itself not being the primary topic of this publication.

3.1 MULTI-THREADED HELLO WORLD

It is customary to begin programming discussions with something that displays “Hello World.” To do so concurrently, we implement the following code. It compiles and works out of the box in Visual Studio, without the need for linking to any third-party library or special project settings. All we need do is include the header <thread>.

```
1  #include <thread>
2  using namespace std;
3
4  void threadFunc()
5  {
6      cout    << "Hello world from worker thread "
7              << this_thread::get_id()
8              << endl;
9  }
10
11 int main()
12 {
13     const size_t n = thread::hardware_concurrency();
14     vector<thread> vt(n);
15     for( size_t i=0; i<n; ++i)
16     {
17         vt[i] = thread(threadFunc);
18     }
19     cout << "Hello world from main thread " << endl;
20     for(size_t i=0; i<n; ++i)
21     {
22         vt[i].join();
23     }
24     cout << "Completed " << endl;
25 }
```

$thread :: hardware\_concurrency()$ is a static method that provides at run time the number of hardware threads on the machine executing the code. Our code results in interlaced messages on the application window, since multiple threads are writing into the window without any form of synchronization:



3.2 THREAD MANAGEMENT

The instruction $vt[i] = thread(threadFunc)$ on line 17 creates a new object of a class type $thread$ and initializes it with a callable object. This results in the creation of a new thread, which immediately

begins the execution of the callable argument, while the main thread continues the execution of $main()$.

The $thread$ object returned in $vt[i]$ is not an actual thread but a thread *handle*, conceptually similar to a pointer on a thread. Its constructor fires the execution of the callable argument on a newly created parallel thread; the handle provides basic control over that thread. It is this handle and not the thread itself that is represented in the $thread$ class.

This is how we start parallel threads. We create new threads and give them a job to do, but the actual *scheduling* (the execution of the threads on CPU cores) is delegated to the OS.[4] We access the number of available hardware threads with $thread :: hardware\_concurrency()$, which helps write generic code that scales at run time with the numbers of cores. The standard library does not allow to schedule a thread on a specific core.[5] This is called *affinity* and may be implemented with OS specific libraries. We will not use affinity.

The callable argument to the $thread$ constructor may be a function, a member function belonging to an object, a function object, or a lambda. It may have parameters, in which case we pass its arguments to the thread constructor after the name of the function. This pattern for callable objects is consistent throughout C++11 and works as follows. It is important to understand this part well, because this pattern for callables is reused everywhere in C++11. In a vast majority of cases of practical relevance, however, the most convenient way to manipulate callables is with lambda expressions.

- **Free function** We give the name of the function followed by its arguments. For instance:

```
int someFunc(const int a, double& b, const string& c);
//  ...
double x;
auto t = thread(someFunc, 2, ref(x), "I am multithreading!");
```

We note that whatever the function returns gets lost in the process, and, indeed, we cannot return results from threads this way. Passing results from worker threads to caller threads is discussed in Section 3.17.

We can pass arguments by reference, but in this case, there is a subtlety we must point out. We don't send arguments directly to our function; we pass them to the constructor of *thread*, which *forwards* them to our function. Our function may take some arguments by reference, but the thread constructor is implemented with (variadic) templates, where type resolution would result in passing the argument by value, unless we explicitly make it a reference with the keyword *ref* (which is actually a function located in the header <functional>). When in doubt, it is best to use pointers to avoid confusion. Even more convenient is to use a lambda that can capture variables by reference instead of taking them as parameters.

- **Function Object** A function object is an instance of a class that defines the operator () and which instances are therefore callable like functions. For example:

```
//  Some class for the representation of implied vol surfaces
class ImpliedVolSurface
{
    //  ...
public:
    //  Picks an implied vol for a given strike and expiry
    void operator() (const double K, const double T, double* ivol);
};
```

In this case, to pick a volatility *asynchronously* on the surface we proceed as follows:

```
//  An instance of ImpliedVolSurface named iVolSurf
ImpliedVolSurface iVolSurf;
//  ...
double strike, mat;
//  ...
double vol;
//  Pick vol from a different thread
//  We pass the object (INSTANCE) as callable
//      followed by arguments
thread t(iVolSurf, strike, mat, &vol);
//  Do something in main thread in parallel
//  ...
//  Wait for parallel thread to finish
t.join();
//  The required volatility is is now computed in vol
//  Do something with the picked implied vol
//  ...
```

The first argument to the thread constructor is the name of the function object (the instance, not the class), followed by a list of the arguments. Note that the argument for the result is passed by pointer to avoid difficulties with references.

- **Member Function** We may also ask the thread to execute a method on a given object. For instance, the previous example could have been written differently:

```
//      Just an object, not a function object
class ImpliedVolSurface
{
//  ...
public:
    //  Just another method, not the operator ()
        void getVol(const double K, const double T, double* ivol);
};

// The instance
ImpliedVolSurface iVolSurf;
//  ...
double strike, mat;
//  ...
double vol;
//  Pick vol on the surface from a different thread
thread t(&ImpliedVolSurface::getVol, &iVolSurf, strike, mat, &vol);
//  Do something in main thread in parallel
//  ...
//  Wait for parallel thread to finish
t.join();
//  Do something with the picked implied vol
//  ...
```

In this case, the first argument to the thread constructor is the name of the *method* to be called, qualified with the name of the class: "&class::method." The second argument is the instance that executes the method. And the rest is the list of the arguments to the method as before.

Now that may be confusing and hard to remember. It is most convenient to use lambdas in all cases. Of course, we can easily replicate all the examples above with simple lambdas, and there may not be any doubt about references or values in this case, as this is clearly specified in the lambda's capture clause. For example, the last example could be rewritten as follows:

```
thread t( [&] () { iVolSurf.getVol(strike, mat, &vol); } );
```

That syntax "thread t(callable,...)" creates a new thread and fires the execution of the $callable$ on that thread. When the execution of the $callable$ completes, the thread is destroyed. The thread handle t provides the caller with basic means to inspect and control the thread. More specifically, the caller may use the handle to:

- **Identify the thread** with $t.get\_id()$.
- **Wait for the thread to complete** with $t.join()$. If the thread already completed, $join()$ returns immediately. Otherwise, it causes the calling thread to wait *in idle state*, without consuming any CPU resources, for the thread referenced by t to complete. $t.join()$ cannot return before "t" completes, hence it is guaranteed that the job that was given to t on construction completed after $t.join()$ returned. The code that comes after this can rely that the asynchronous job is complete; in particular, it may safely use the results.
- **Break the handle** with $t.detach()$. This breaks the *link* between the handle t and the thread it represents. It does not affect the thread itself in any way. This means that we lose control of the thread; we have no longer any means to know what is going on

there. In particular, we can no longer wait for the thread to complete its job. The thread will finish its work and terminate behind the scenes, unmanaged and invisible.

Thread handles are *moveable* objects. They cannot be copied (an attempt to do so results in a compile error) but they can be *moved*. When something like “thread lhs = move(rhs)” is executed, rhs loses the link to the thread, which ownership is transferred to lhs. As a curiosity, the instruction we programmed in our Hello World, $vt[i] = thread(threadFunc)$, implements an implicit move from a temporary object. $thread$ does not allow copy assignment.

When the thread linked to a handle completes, the handle is still considered *active* until its method $join()$ is invoked. After that call, the handle becomes inactive. When the link between the thread and the handle is broken by a call to the handle's $detach()$ method, the handle also becomes inactive. The active status of a handle may be checked with a call to its $joinable()$ method (returns $true$ if the handle is active). We cannot call $join()$ on an inactive handle. An active handle, on the other hand, cannot exit scope. If that ever happens, the application *terminates*. This means that either $join()$ or $detach()$ *must* be called on the handle before it vanishes. We must either wait for the thread to complete or release it to finish its life outside of our control, and we must do so explicitly.

For this reason, it is often advised to manage thread handles with the RAII idiom, similarly to smart pointers. We should wrap a $thread$ within a custom class that calls $join()$ in its destructor. This would guarantee that an active handle exiting scope would cause to wait for the linked thread to complete rather than crash the application. However, all this doesn't matter that much to us because *we never manage threads directly*.

The creation of a thread comes with a significant overhead, and the OS can only handle so many threads. For these reasons, we limit the num-

ber of threads we create (typically to the number of cores *minus one* so that one core remains available for the main thread) and we want to keep the thread logic outside of the algorithmic code, both for performance (no thread creation overhead during the execution of the algorithm) and encapsulation (algorithms shouldn't worry about hardware logic).

For example, in the case of parallel Monte-Carlo simulations, we execute different paths on different threads. If we simulate 65,536 paths, we cannot create 65,536 threads. That would bloat the OS. Instead, we could create $thread::hardware\_concurrency() - 1$ threads, divide the 65,536 paths into $thread::hardware\_concurrency()$ batches, send one batch to each created thread, and keep one batch for execution on the main thread. This whole logic should not be implemented in the Monte-Carlo code. This hardware-dependent division does not belong to simulation logic. Besides, thread creation is expensive so we would rather create the worker threads once and for all (for instance, when the application starts) and have them wait (without consuming any CPU or other resources) to be given jobs to do in parallel, until they are destroyed when the application exits.

When some algorithm such as the Monte-Carlo simulator sends a job for a parallel execution, some kind of scheduler should assign a worker thread to execute it and then send the thread back to sleep, but without destroying it (otherwise we would incur its creation cost again when we need it next). This whole logic is best encapsulated in a dedicated object. An object that works in this manner is called an *active object* because it sleeps in the background and awakens to conduct parallel work when needed. Modern GUI design is based on this active object pattern. The particular active object that dispatches tasks over worker threads is called a *thread pool* and we discuss these in detail in Section 3.18.

In addition, basic thread management does not provide mechanisms to return results to the caller or communicate exceptions in case

something goes wrong. We will provide such facilities in our thread pool.

First, we investigate the *synchronization* of threads.

3.3 DATA SHARING

Each thread owns its private execution stack. Concretely, this means that each thread works with its own copy of all the variables declared within the functions and methods it executes, including nested ones. When the type of the variable is a reference or a pointer, however, the thread still exclusively owns its copy of the pointer/reference, but this is not necessarily the case for the referred object. The object could live on the heap, or on the stack owned by another thread. It could be *shared*, referenced, and accessed *concurrently* (meaning at the same time) from multiple threads. Multiple threads reading a shared object causes no trouble, but concurrent writes produce deep, dangerous, hard-to-debug problems called *race conditions*. The same applies to global and static variables shared by all threads.

The demonstration code below shows a few examples.

```cpp
#include <thread>
using namespace std;

//  Global
vector<int> v1 = { 1, 2, 3, 4, 5 };

void threadFunc(vector<int>* v)
{
    //  Local
    vector<int> v3 = { 5, 4 };

    //  OK: reading shared data, writing into own data
    v3[0] += (*v)[0];

    //  RACE: changing shared data
    v->push_back(1);

    //  RACE: writing into global data
    v1[0] += 1;
}

int main()
{
    //  Belongs to the main thread's stack
    vector<int> v2 = { 3, 2, 1 };

    const size_t n = thread::hardware_concurrency();
    vector<thread> vt(n);
    for (size_t i = 0; i<n; ++i)
    {
        //  v2 passed by reference to other threads
        vt[i] = thread(threadFunc, &v2);
    }
    for (size_t i = 0; i<n; ++i)
    {
        vt[i].join();
    }

    for_each(v1.begin(), v1.end(),
        [](const int& i) {cout << i << " "; }); cout << endl;
    for_each(v2.begin(), v2.end(),
        [](const int& i) {cout << i << " "; }); cout << endl;
}
```

Worst thing is, this code will probably run just fine and display the expected results. We may have to run it many times to see a problem. Races occur when threads read and write into memory *at the same time*. That is a low-probability event. But this does not make it better. It makes it worse. Races cause random, nonreproducible bugs that may crash the application, produce absurd results, or (which is much worse) produce wrong results by small amounts so they may not be noticed.

3.4 THREAD LOCAL STORAGE

The variables that are global (declared outside classes and functions) are shared among threads. That is also true of *static* variables, which, for all intents and purposes, are global variables declared within classes or functions. Concurrent writes into global or static variables cause data races, just as with other shared objects. For this particular case, however, C++11 provides a specific solution and even a special keyword $thread\_local$.

This keyword only applies to global and static variables, and when any such variable is declared with the $thread\_local$ specifier, every thread works with its own copy of it. The code accesses this variable like any other global or static variable, but in doing so, each thread reads and writes to a different memory location.

We will see a simple, yet very useful example in our thread pool implementation of Section 3.18, where a thread local variable represents the index of the threads: 0 for the main thread or 1 to

$$thread::hardware\_concurrency() - 1$$

for the worker threads. Another example will be given in Part III, where the *tape*, the data structure at the core of AAD that records mathematical operations, is declared thread local, allowing AAD to work with parallel algorithms.

3.5 FALSE SHARING

We briefly introduced (true) sharing and the dangers of writing into shared data, and discussed thread local storage as a solution for the (rare but important) case of global/static data. Before we investigate more general mitigation to (true) sharing, we introduce *false sharing*.

As a first example of a real multi-threaded algorithm, we compute the non-parametric distribution of some data in a vector. The single-

threaded code is self-explanatory.

```cpp
#include <vector>
#include <algorithm>
using namespace std;

vector<int> dist(
    const vector<double>& data,
    const vector<double>& knots)
{
    //  Vector of results: count data between knots
        const size_t n = knots.size() + 1;
        vector<int> res(n, 0);

    //  Loop on knots
        for (size_t i = 0; i < n; ++i)
        {
            //  Lower bound
                const double lb = i == 0 ?
                    -numeric_limits<double>::max()
                    : knots[i - 1];

                //  Upper bound
                const double ub = i == n - 1 ?
                    numeric_limits<double>::max()
                    : knots[i];

        //  Count data
        for (size_t j = 0; j < data.size(); ++j)
            res[i] += (lb < data[j] && data[j] <= ub);
        }
        return res;
}
```

We test the algorithm as follows:

```
#include <iostream>
#include <ctime>
int main()
{
    const size_t n = 50000000;
    vector<double> data(n);
    srand(12345);

    //  Populate data with n uniform numbers in (0, 1)
    generate(data.begin(), data.end(),
        [] () {return double(rand()) / RAND_MAX; });

    //  Knots
    vector<double> knots = { 0.1, 0.15, 0.2, 0.21, 0.25, 0.35, 0.4,
        0.5, 0.55, 0.6, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95 };

    //  Measure time
    time_t t1 = clock();
    vector<int> result = dist(data, knots);
    time_t t2 = clock();

    //  Display results
    cout << "result " << endl;
    for_each(result.begin(), result.end(),
        [=](const int& i) { cout << double(i) / n << " "; });
    cout << endl << "time (ms) " << t2 - t1 << endl;
}
```

We generate a large vector of random uniform data between 0 and 1 (using the STL algorithm $generate()$), send it to our algorithm and display results (with the very useful algorithm $for\_each()$) as well as the time spent.

We get the expected results in around 3 seconds on our iMac Pro. To make this algorithm parallel (across knots) is apparently trivial. Note that we create threads inside the algorithm. We are not supposed to do that, but we don't know any other way yet:

```
1  #include <thread>
2  vector<int> parDist(
3      const vector<double>& data,
4      const vector<double>& knots)
5  {
6      const size_t n = knots.size() + 1;
7      vector<int> res(n, 0);
8      vector<thread> threads(n);
9
10     for (size_t i = 0; i < n; ++i)
11     {
12         //  Exactly the same code, except we send each knot
13         //      on a separate thread
14
15         //  i is captured by value
16         threads[i] = thread([&, i]()
17         {
18              //  i read here, main thread would interfere
19              //      if captured by reference
20              const double lb = i == 0 ?
21                 -numeric_limits<double>::max()
22                 : knots[i - 1];
23
24              const double ub = i == n - 1 ?
25                 numeric_limits<double>::max()
26                 : knots[i];
27
28              for (size_t j = 0; j < data.size(); ++j)
29              res[i] += (lb < data[j] && data[j] <= ub);
30         });
31     }
32     for (size_t i = 0; i < n; ++i) threads[i].join();
33     return res;
34 }
```

Note that the lambda captures the counter i by value. The main thread changes i in the dispatch loop, so if we captured it by reference, we could read a changing value of i from the lambda executed on worker threads. The (const) vectors are captured by reference.

We get the exact same results (thankfully) in around a second. Now this is disappointing. Why only 3 times the speed with 8 cores?

Note that this code does not write to any shared data. On the line 28, each thread writes into its dedicated entry i of the result vector, hence, threads don't interfere with one another. Or do they? What if we change this line of code so that within the loop we write into a local variable instead, replacing the code of lines 28–29 by:

```
int count = 0;
for (size_t j = 0; j < data.size(); ++j)
    count += (lb < data[j] && data[j] <= ub);
res[i] = count;
```

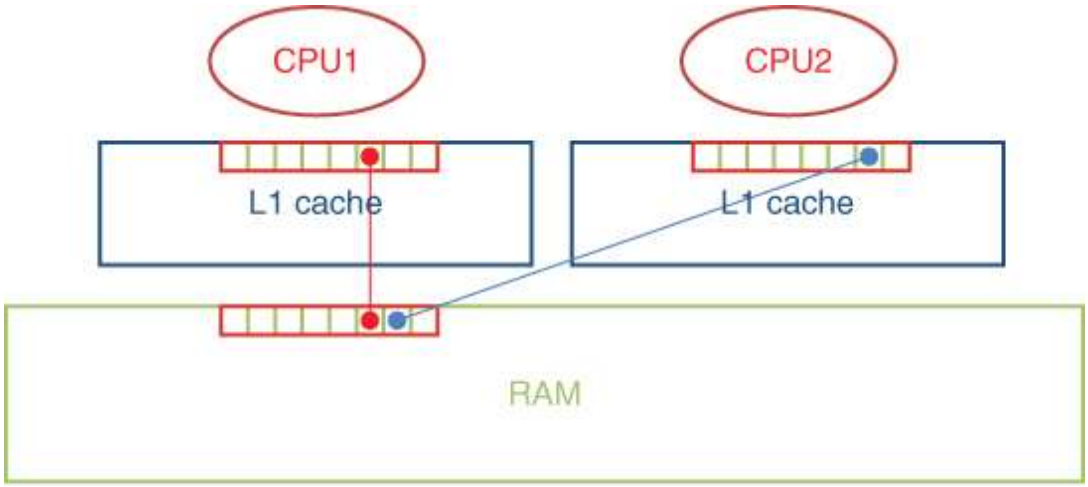
Now we get the same results in 2.5 seconds single-threaded, 300 milliseconds multi-threaded. Not only is the serial code faster; because it is faster to access the stack variable $count$ than access $res[i]$ on the the heap, we also get a parallel acceleration of more than 8 times, as expected from multi-threading over 8 hyper-threaded cores.

Note that if we had used the STL algorithm $count\_if()$ in place of the hand-crafted loop, as is recommended best practice, we would have got this result in the first place:

```
res[i] = count_if(data.begin(), data.end(),
    [=](const double x) { return lb < x && x <= ub; });
```

This illustrates, once again, that we should always prefer STL algorithms to hand-crafted loops. So $count\_if()$ would have shielded us from the problem by avoiding repeated concurrent writes into the vector res. But why was that a problem in the first place? Each thread was writing into its own space within that vector, so there shouldn't have been any interference. And yet they did interfere. While the threads wrote into separate spaces of memory, so there was no “true” sharing; they were writing to locations close enough to one another so they belonged to the same *cache line*. We recall that a cache line is what is transferred back and forth between the RAM and the CPU cache, and its size is generally 64 bytes or 8 integers (we compiled in 64 bits). So the entries $res[0]$, $res[1]$,..., $res[7]$ lived on the same cache line. Threads were writing into separate memory cells but into the same cache line. So in some sense there was sharing involved. This form of sharing through cache lines is called *false sharing* or cache ping-pong for a reason that will be made apparent imminently.

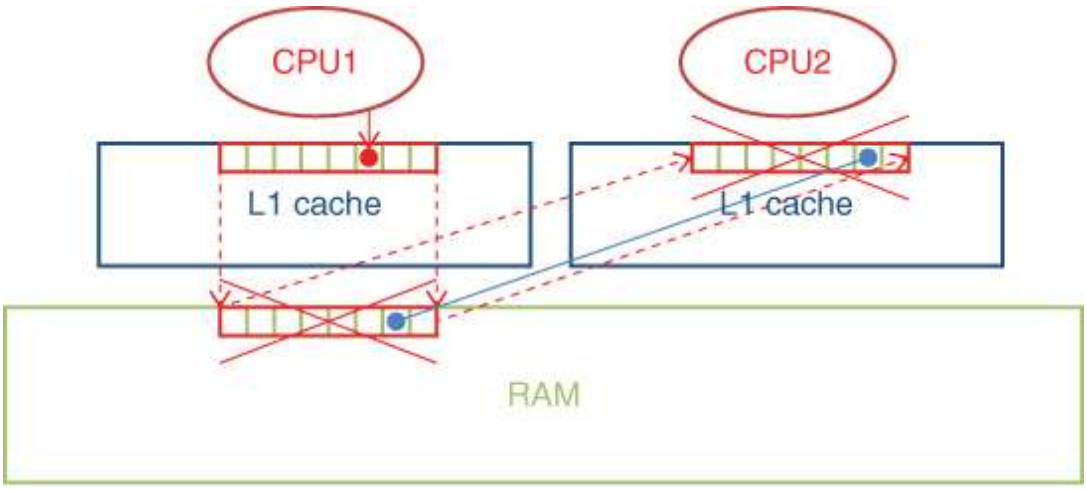
We have seen in Chapter 1 that when a core reads a memory cell, it first looks for that location in the cache. If it is not there, the cache is *missed*, and the cell is fetched from RAM and cached along with its line (the 64 bytes surrounding it). Each core has its own cache. If another core reads another RAM cell on the same line, it also caches the line:



Read

At this point, both cores cached the line. When one core subsequently *writes* into a memory cell on that line, it actually writes into its own cache. This causes an inconsistency between the RAM and the cache. If another core tried to read the memory cell, it would get a wrong read from the RAM because the first core only updated its cache. So the first core must "warn" the system about the discrepancy. It does so by marking the RAM cell invalid, for everyone else to know. But it cannot invalidate a single cell; it invalidates the whole line. When the second core reads another cell on the same line, it cannot read it from its own cache, because it sees the whole line as invalid. It cannot read it from RAM, either, for the same reason. The only place it is guaranteed to find an up-to-date read is in *the first core's cache*. So, the whole line must be transferred from the first core's cache into RAM, and from RAM into the second core's cache, where the second core finally reads it. Then, of course, the second core writes into the same memory line and the whole process is repeated. We know that this is all unnecessary, that each core actually invalidates one cell on the line that is not used by the other cores. But the CPU does not know that. It can only

see that lines are valid or not; it doesn't know the status of individual cells. And, when in doubt, it transfers the whole line.



Write

This is false sharing, or cache ping-pong, and this is why our initial concurrent code did not perform as expected. Now we understand the matter, we may even wonder by what miracle it still achieved a decent speed improvement? It could easily have resulted in a parallel code *slower* than its serial counterpart.

The answer is that we simplified the design of CPU caches to avoid confusion in our initial discussion. In reality, modern CPUs typically incorporate a hierarchy of three levels of cache. L1 cache is the smallest and fastest and sits on CPU cores. L2 is bigger and slower and there is also one for every core. L3 is slower, bigger[6] and more importantly shared among all cores on a CPU. RAM is the biggest and slowest of all. These cache levels form a hierarchy, which means that every memory access is first checked in L1, if missed, checked in L2, if missed in L3 and finally in RAM, caching the line all the way up to the core.

That the L3 cache is shared among cores is what mitigates false sharing, because cache ping-pong occurs between L2 and L3 caches, not between cache and RAM. This is why we still saw a decent speed improvement from our initial multi-threaded code.

We noted in Chapter 1 that code *within a thread* should operate on coalescent memory for cache efficiency. Now we see that in addition, codes that execute on separate threads should work with memory distant from one another. We are now aware of false sharing and we learned a few things about hardware in the process, so we can compile a few guidelines to mitigate false sharing, and for the production of effective code in general:

1. Prefer STL algorithms to hand-crafted loops.
2. Cache repeatedly accessed heap data into local stack variables.
3. When concurrent writing into close memory is unavoidable, *pad* that memory with blocks of dummy data to separate memory written into from different threads. For example, say we use a data structure:

```
struct SomeObject
{
    double x;
    double y;
};
```

so that one thread frequently writes into x and another into y, producing false sharing interference. We can eliminate false sharing (at some memory cost) by padding the structure with data the size of a cache line:

```
struct SomeObject
{
    double x;
    char pad[64];
    double y;
};
```

3.6 RACE CONDITIONS AND DATA RACES

Race conditions and data races are the most common flaws in multithreaded programs, caused by incorrectly synchronized data sharing

among threads.

A *race condition* is a flaw that occurs when the timing or ordering of events affects a program's correctness. A *data race* is one nasty race condition where a thread reads some memory location *while* it is being written by another thread, which may result in reading invalid or corrupted data.

The textbook example of a race condition is the concurrent access to a bank account. Consider the class:

```
1  class SharedBankAccount
2  {
3      double myBalance;
4
5  public:
6
7      double withdraw( const double amount)
8      {
9          if( amount <= myBalance)
10         {
11             myBalance -= amount;
12             return amount;
13         }
14         else return 0;
15     }
16 };
```

When the method $withdraw()$ is called concurrently from two threads, its instructions are executed in parallel from both threads. How fast each thread executes the sequence of instructions depends on scheduling and may be different every time the program runs. For instance, two concurrent withdrawal requests for $100 from an account with a balance of $120 could be scheduled as follows:

thread 1	**thread 2**	**balance**
withdraw(100)	withdraw(100)	120
amount balance : true		120

thread 1	thread 2	balance
	amount balance : true	120
balance -= amount		20
	balance -= amount	−80
get 100		−80
	get 100	−80

This is one *possible* order of execution where thread 2 checks that the balance is sufficient after thread 1 conducted the same check, but *before* thread 1 actually updated the balance. Hence, at the exact moment where thread 2 checks the balance, it is still sufficient and thread 2 is authorized to perform the withdrawal. Both threads perform the withdrawal, resulting in a negative balance, something that the program is supposed to prevent.

The order of execution could have been different. The execution of the actual withdrawal on line 11 could have been scheduled on thread 1 *before* thread 2 checked line 9. In that case, thread 2 would have correctly been denied because the balance would have been insufficient at check time. Another possibility is that thread 2 could have been scheduled to conduct the check on line 9 *while* thread 1 was updating $myBalance$ on line 11. In that case, thread 2 could have read some invalid, corrupted, partly written data, resulting in a random number or a crash.

Therefore, depending on the exact scheduling of the execution on the two threads, the result could be correct, incorrect, or corrupted. The result may be different every time the program runs *with the same parameters*, depending on *who gets there first*, hence the name *race* con-

dition. When the result is a corruption where a thread reads a location in memory at the exact same time another thread is writing into it, we have a *data race*.

Races are particularly hard to identify, reproduce, debug, and correct, precisely because they may happen or not depending on scheduling. The fact that we inspect or debug a program may in itself affect scheduling and prevent races. This is why some developers call them “quantum bugs.” A lot of concurrent programming is about avoidance of races. One means of avoiding races is by locking parts of the code so they don't execute concurrently.

3.7 LOCKS

C++11 provides a *thread synchronization primitive* called *mutex* in the <mutex> header. A mutex is an object that may be locked (by calling its $lock()$ method) by one thread at a time. Once a thread *acquired a lock* on a mutex, the mutex will deny locks to other threads until its owner thread unlocks it (by a call to $unlock()$). In the meantime, other threads will wait for the lock *in idle state*. When the owner thread releases the lock while other threads are waiting on it, one of the waiting threads will awaken and acquire the lock. Therefore, the blocks of code that cannot be safely executed in parallel (we call those *critical sections*) may be surrounded by calls to $lock()$ and $unlock()$ on some mutex, which guarantees that only one thread can execute this code at a time.

The withdrawal code is fixed as follows:

```
#include <mutex>
using namespace std;

class SharedBankAccount
{
        mutex myMutex;
        double myBalance;

public:
        double withdraw( const double amount)
        {
                myMutex.lock();

                if( amount <= myBalance)
                {
                        myBalance -= amount;
                        myMutex.unlock();

                        return amount;
                }
                else
                {
                        myMutex.unlock();

                        return 0;
                }
        }
};
```

A thread locks the mutex before checking the balance and only releases it after it finished updating the balance. This guarantees that another thread cannot read or write the balance in the meantime. In order to do so, it would need to acquire the lock, and for that, it would have to wait until the lock is released. The code surrounded by $myMutex.lock()$ and $myMutex.unlock()$ is *atomic*: it is executed entirely on one thread without exterior interference. Other threads may read or write the balance before or after, but not during, the execution of the critical section. The following possible scheduling illustrates this notion:

thread 1	**thread 2**	**balance**
withdraw(100)	withdraw(100)	120
lock granted		120

thread 1	thread 2	balance
	lock denied	120
amount balance : true	wait	120
balance -= amount	wait	20
unlock	lock granted	20
get 100	amount balance : false	20
	unlock	20
	get 0	20

It is clear that we may no longer see data races or negative balances. However, we have *not* eliminated race conditions. Whether it is thread 1 that is served and thread 2 that is denied, or the contrary, still depends on which locks the mutex first, which depends on scheduling and may be different on every run. However, such remaining race is *benign* in the sense that it cannot defeat the balance check or corrupt data. One of the users always withdraws $100, the other user is always denied, and the balance always ends at $20. No corruption may occur.

In order to effectively prevent races, locks must be acquired before the first manipulation of shared data and released after the last manipulation, so as to make the combined manipulation atomic. On the other hand, locks serialize code and defeat the benefits of parallelism, since only one thread can execute it at a time. For this reason, locks should never be acquired earlier than necessary and never released later than necessary. In our example, we acquire a lock before we

check the balance and we release it as soon as we are done writing its update.

3.8 SPINLOCKS

How difficult is it to code a mutex? The following code apparently works:

```
1  class MyOwnMutex
2  {
3      bool myLocked;
4
5  public:
6      MyOwnMutex() : myLocked( false) {}
7      void lock() { while( myLocked); myLocked = true; }
8      void unlock() { myLocked = false; }
9  };
```

but it is effectively flawed: the implementation of $lock()$ consists of two *distinct* instructions: check ("while(myLocked)") and set ("myLocked = true"). Because these two instructions are distinct (or *non-atomic*), another thread may step in between the check and the set and execute the critical code along with the thread that locked the mutex, defeating protection.

This is fixed by making the check and set *atomic*, meaning that they are executed in such a way that another thread could see the state of $myLocked$ before or after, but not during the whole operation. The class atomic<bool> from header <atomic> defines a method $exchange()$ that sets the state to the argument and returns the former state, and does all of this *atomically*, as a single instruction, so that another thread cannot step in or peek into the data during its execution. Therefore, the following code effectively fixes our custom mutex, which we now call a "spinlock":

```cpp
#include <atomic>
using namespace std;

class SpinLock
{
    atomic<bool> myLocked;

public:
    SpinLock() : locked( false) {}
    void lock() { while( myLocked.exchange( true)); }
    void unlock() { myLocked = false; }
};
```

That is a perfectly valid lock and it effectively protects critical code against concurrent execution.

But it is not a mutex.

In fact, we just developed a well-known synchronization object called a *spinlock*. A spinlock works just like a mutex and it even implements the same interface, but it behaves differently when waiting on a lock. A spinlock waits "actively," constantly checking availability, consuming an entire CPU core while waiting, which makes that CPU unavailable to other threads. A mutex puts a waiting thread in idle state, not consuming any resource while waiting, and awakens the thread to resume execution when the lock becomes available. In the meantime, the core is available to conduct work for other threads.

The truth is *we cannot program a mutex in C++*. A mutex really is a primitive, which implementation is delegated to the OS, and ultimately to hardware.

What we can easily program is a spinlock, and we should use one or the other depending on the context. A thread that waits on a mutex releases its CPU for other threads, but a thread that waits on a spinlock resumes faster when the lock becomes available. When the speed of resuming execution on acquiring the lock is critical, we use a spinlock. When we know that the thread may wait a while and need its CPU for other tasks, we use a mutex. Mutexes are generally preferable and spinlocks must be used with care in particular situations only.

Multiple spinlocks in waiting state cause a heavy CPU load that may result in a severe performance drag. This is particularly the case with hyper-threading, where threads share CPUs.

We call mutexes and spinlocks and anything that exposes the methods $lock()$ and $unlock()$ for the purpose of protection of critical code *lockable* objects.

3.9 DEADLOCKS

What happens when a thread acquires a lock and never releases it? Another thread attempting to acquire the lock will wait *forever*. Such situation is known as a *deadlock*. This is obviously something we must avoid.

How could that happen? One textbook situation (not really applicable in contexts of practical relevance in finance) is when thread 1 acquired lock A and needs to acquire lock B to complete a critical operation before it may release lock A. Thread 2 holds lock B and must acquire lock A to complete the execution of its critical code and release lock B. Both threads are waiting for the other to release its lock so they may release their own. Obviously, this never happens and both threads are effectively deadlocked.

This is why we must avoid nested locks. When nested locks cannot be avoided[7] C++11 provides in <mutex> a free function $lock()$ that acquires multiple locks atomically with a deadlock avoidance algorithm. It is used as follows:

```
mutex m1, m2, m3;
//  ...
lock( m1, m2, m3);
//  ...
m1.unlock(); m2.unlock(); m3.unlock();
```

The free function $lock()$ works not only with mutexes, but all lockable objects, including spinlocks.

3.10 RAII LOCKS

The other cause of deadlocks is much more common. It happens when a thread acquires a lock and "forgets" to release it. Then any other thread that attempts to acquire the lock is deadlocked. This may happen in functions with multiple branches. Our withdrawal code has two branches, depending on whether the balance is sufficient or not. The lock must be released on all branches. This includes "implicit" branches, when the function exits on an *exception*. An exception may be thrown in the function, or in a nested call to another function. Exceptions cause functions to exit immediately, and in that case an acquired lock may not be released.

This situation is similar to memory leaks that occur when a function allocates memory, then exits on an exception without freeing memory. Memory leaks are undesirable, and resolved with smart pointers. Smart pointers release memory in their destructor, so memory is released whenever they exit scope for whatever reason. Forgetting to free memory is a serious nuisance, but forgetting to release a lock is much worse.

For this reason, we *never* manipulate mutexes[8] directly but always through RAII objects that release the locks in their destructors. This ensures that acquired locks are always released when execution exits the block where the RAII object was declared, whatever the reason. This is so important that C++11 provides two such RAII objects out of the box, in the header <mutex>: the $lock\_guard$ and the $unique\_lock$.

A $lock\_guard$ acquires a lock on a mutex on construction, and releases it on destruction. It is a templated type that works with mu-

texes, spinlocks, and anything that implements the *lockable* interface. It is used as follows:

```
#include <mutex>
using namespace std;

class SharedBankAccount
{
    mutex myMutex;
    double myBalance;

public:
    double withdraw( const double amount)
    {
        lock_guard<mutex> lk( myMutex);
        //  Acquire lock

        if( amount <= myBalance)
        {
            myBalance -= amount;
            return amount;
        }
        else
        {
            return 0;
        }
    } // lk goes out of scope, the lock is released
};
```

We acquire the lock through the *lock_guard* and we never release it explicitly. When execution exits the block that declares the *lock_guard*, for whatever reason, its destructor is invoked and the lock is released automatically.

A *lock_guard* acquires a lock on construction, releases it on destruction, and doesn't do anything else. It is typically declared on the first line of a block of code (code surrounded by brackets), which makes this block atomic.

C++11 also provides a more flexible construct *unique_lock*, also in the header <mutex>. That is another RAII lock that offers the functionality of a *lock_guard*, plus:

- A *unique_lock* can *defer* the acquisition of a lock until after construction, when constructed with the *defer_lock* flag, whereas a *lock_guard* always acquires the lock on construction.

- A *unique_lock* can explicitly release the lock *before* destruction with a call to its *unlock()* method. A *lock_guard* only releases the lock on destruction.
- *unique_lock* provides a non-blocking *try_lock()* method. If the mutex is already locked by another thread, *try_lock()* returns false but does not put the caller thread to sleep or stop execution.
- A *unique_lock* can *adopt* a mutex previously locked on the caller thread when constructed with the *adopt_lock* flag.
- And more, like a timer base *try_lock()* that attempts to acquire a lock for a given time.

A *unique_lock* is used as follows:

```
mutex m;
//  ...
unique_lock<mutex> lk( m, defer_lock);   //  not locked
//  ...
lk.lock();          //  now locked
//  ...
lk.unlock();        //  unlocked here, otherwise on destruction


mutex m2; m2.lock();
//  ...
//  adopted: released on destruction
unique_lock<mutex> lk( m2, adopt_lock);
```

The flexibility of a *unique_lock* comes with an overhead, whereas a *lock_guard* provides RAII management for free.

RAII locks can be instantiated with all types of lockable objects. A lockable class is any class that defines the methods *lock()* and *unlock()*. Hence, a *unique_lock* is itself lockable (but a *lock_guard* is not). In particular, *lock()* can acquire multiple *unique_lock*s at the same time.

None of the RAII locks is copyable. A *unique_lock* is moveable. A *lock_guard* is not.

3.11 LOCK-FREE CONCURRENT DESIGN

It is clear that locks effectively *serialize* critical code, preventing its parallel execution in all circumstances, hence preventing races but also defeating the benefits of concurrency. In the Introduction, we explained how partial parallelism leads to disappointing results in accordance with Amdahl's law. Locks always result in partial parallelism and cause poor parallel efficiency.

In addition, locks are expensive in themselves. Standard mutexes may put a thread in idle state if the mutex is locked by another thread, and awaken it when the mutex is unlocked. This may result in a context switch on the executing core. All of this causes an overhead. Spinlocks, on the other hand, occupy an entire core while waiting on a lock, making the core unavailable for other threads to perform useful operations. Finally, locks may result in deadlocks, and for this reason they must be used with care and parsimony and always with RAII constructs.

Locks are necessary in certain circumstances, but to use them systematically may lead to parallel code both slower and more dangerous than its serial counterpart. The art of concurrent programming is very much about making code thread safe *by design*, without locks.

How do we concretely achieve lock-free thread safe design? First, reading from shared objects is always safe; it is only concurrent *writes* that cause races. For this reason, *const correctness* goes a long way toward thread safety. The compiler guarantees that methods marked `const` cannot modify objects. We may therefore assume that they are safe to execute concurrently. The same applies to `const` arguments. For this reason, we must clearly mark `const` all the methods that don't modify the state of the object, and the arguments that are not modified by functions or methods. This is called *const correctness* and is routinely ignored by many developers. It doesn't matter that much in a single-threaded context, but it is critical with concurrent programs.

In addition, we must absolutely refrain from *false constness*, whereby methods marked $const$ somehow modify state nonetheless. Such undesirable behavior is achieved, for example, with the $mutable$ keyword. Data members that are *conceptually* not part of the state (such as a cache or some preallocated working memory) may be marked $mutable$ so the compiler does not complain when $const$ methods modify them. But, of course, that would cause race conditions all the same when such methods are executed concurrently. The $mutable$ keyword must be banned from concurrent code. Similar behavior is achieved by casting away constness, something also best avoided for the same reasons.

Another example of false constness is when $const$ methods on an given object O effectively don't modify that object, but change the state of another object, that may be referred to by pointer or reference in O. In this case, just like with mutables, $const$ methods may no longer be assumed safe, making concurrent programming more difficult and prone to error. Thread safe design starts with const correctness and the avoidance of all types of false constness.

Following this logic to its extreme consequences, we may be tempted to program software where all methods are (truly) $const$ and where free functions never modify their arguments. Such a program would ban all kinds of mutable state, forcing functions and methods to compute results out of arguments without side effects. The absence of a mutable state and side effects is a major principle of *functional programming* and is enforced in pure functional languages like Haskell. One reason why functional programming earned so much popularity recently is that stateless programs are thread safe by construction. When the language guarantees that functions and methods cannot modify state or parameters, they are all (truly) $const$ and therefore safe for concurrent execution. Races cannot happen and locks are unnecessary.

So functional programming is particularly well suited to concurrency. C++, on the other hand, is a multi-paradigm language, and state is extremely useful in C++ programs. We believe it would be a mistake to forbid state in C++ programs for the benefit of concurrency. Many of the concurrent programs in this book use state. Our methods are not all *const* and they are not all safe for parallel execution. Our free functions occasionally modify their (non *const*) parameters. Yet, all our calculation code is absolutely lock-free. The only place we have locks is in the thread pool of Section 3.18, and even there they are all encapsulated in concurrent data structures.

When our concurrent code calls non-*const* methods or modifies non-*const* arguments, we have *mutable objects*. How we deal with mutable objects is this way: *we make copies of all the mutable objects for every thread before we execute concurrent code*. Our thread pool of Section 3.18 facilitates the management of thread-wise copies. The simple instructions on page 109 allow to determine the number of threads involved so we can make copies, and identify the thread executing concurrent code so it can work with its own copy.

Finally, we reiterate that memory allocation involves hidden locks, unless we use a purposely designed concurrent allocator from a third-party library like Intel's TBB. Therefore, we must also ban memory allocation from concurrent code, preallocating necessary memory before multi-threading, which may be difficult. First, we don't always know how much memory we need ex-ante. Second, whenever we create, copy, or pass to a function or method by value, objects that manage resources, like vectors or other containers, all these operations involve hidden allocations. It takes creativity and careful inspection of the code to move both explicit and implicit allocations out of concurrent code. Our simulation and differentiation code in Parts II and III show some examples. It is not always possible to remove all allocations from concurrent code, but at the very least we must strive to minimize them.

3.12 INTRODUCTION TO CONCURRENT DATA STRUCTURES

We know enough at this stage to start implementing concurrent data structures, classes that store data in a way that is safe to manipulate concurrently. We illustrate our purpose with the design of a basic concurrent queue. A queue is a container that implements a FIFO (first in, first out) logic whereby elements are pulled from the queue in the same order they have been pushed into the queue. The standard STL queue[9] exposes the methods $push()$ (push an element into the queue), $front()$ (access the element in front of the queue), and $pop()$ (removes the front element in the queue), as well as the accessors $empty()$ and $size()$ and it *cannot* be safely manipulated from different threads at once. None of the STL containers can. The following code, therefore, wraps a STL queue with locks to produce a *concurrent queue*, one that is safe to manipulate concurrently:

```cpp
#include <queue>
#include <mutex>
using namespace std;

template <class T>
class ConcurrentQueue
{
    queue<T> myQueue;
    mutable mutex myMutex;

public:

    bool empty() const
    {
        // Lock
        lock_guard<mutex> lk( myMutex);
        // Access underlying queue
        return myQueue.empty();
    }   // Unlock

    // Pass t byVal or move with push( move( t))
        void push( T t)
    {
        // Lock
        lock_guard<mutex> lk( myMutex);
        // Move into queue
        myQueue.push( move( t));
    }   // Unlock

    // Pop into argument
    bool tryPop( T& t)
    {
        // Lock
        lock_guard<mutex> lk( myMutex);
        if( myQueue.empty()) return false;
        // Move from queue
        t = move( myQueue.front());
        // Combine front/pop
        myQueue.pop();

        return true;
    }   // Unlock

    void clear()
    {
        queue<T> empty;
        swap(myQueue, empty);
    }
};
```

This implementation is rather basic. All it does is serialize accesses to the underlying queue with locks. It exposes three methods, *empty()*, *push()*, and *tryPop()*, and all three are integrally protected with a *lock_guard*. This is not a particularly efficient concurrent queue. A high-performance implementation would require a re-implementation of the internal mechanics of the queue with minimum granularity locks; [15] provides such an implementation in its Section 6.2.3. Perhaps surprisingly, it is even possible, although very difficult, to im-

plement a concurrent queue, safe for concurrent manipulation, without locks at all! Such high-performance implementations make a significant difference when concurrent data structures are heavily manipulated from within concurrent threads. For the purpose of parallel Monte Carlo simulations, our basic implementation is sufficient, and a more advanced implementation makes no difference in performance.

3.13 CONDITION VARIABLES

Condition variables (CVs), declared in the header <condition_variable>, provide a mechanism for different threads to communicate and synchronize. When a thread calls $cv.wait()$ on an object cv of type $condition\_variable$, the thread is put to sleep (passively, in idle state, without consuming any resources), until *another* thread calls either $cv.notify\_one()$ or $cv.notify\_all()$ on the same object. $cv.notify\_one()$ awakens one thread waiting on cv (if any). $cv.notify\_all()$ awakens all the waiting threads. CVs allow threads to control one another and to react on events occurring on different threads.

CVs and mutexes are the *only* thread synchronization primitives in the Standard Threading Library, in the sense that:

1. CVs and mutexes cannot be developed in standard C++; their implementation is delegated to the OS, and ultimately to hardware. We already discussed this point with mutexes; the same applies to CVs for the same reasons.
2. CVs cannot implemented with mutexes (and mutexes cannot be implemented with CVs).[10]
3. Other classical synchronization constructs, briefly discussed later, can all be implemented with a combination of mutexes and CVs. Hence, mutexes and CVs are the two building blocks of thread synchronization. The Standard Threading Library, following the C/C++ tradition of flexible, close-to-the-metal programming, only provides these two building blocks, and lets developers build their own,

higher level synchronization constructs in accordance with the specific needs and design of their application.

Condition variables are extremely useful in concurrent programming. The reactive, event-driven behavior of many popular applications use CVs behind the scenes. As a useful example, we make our concurrent queue *reactive*. Pushing an element onto the queue *automatically* causes the element to be processed on a different thread. The threads that push elements are called *producers*; those that process the elements are called *consumers*. Consumers hitting an empty queue are put to sleep. Pushing an element onto the queue causes one of them to awaken and process the element. This design pattern is called the *active object pattern*, because it makes threads react to events occurring on these objects. An active concurrent queue is called a *producer-consumer queue*. Producer-consumer queues are implemented with mutexes and condition variables:

```cpp
#include <queue>
#include <mutex>
#include <condition_variable>
using namespace std;

template <class T>
class ConcurrentQueue
{
    queue<T> myQueue;
    mutable mutex myMutex;
    condition_variable myCV;

public:

    bool empty() const
    {
        //  Lock
        lock_guard<mutex> lk( myMutex);
        //  Access underlying queue
        return myQueue.empty();
    }   //  Unlock

    //  Pop into argument
    bool tryPop( T& t)
    {
        //  Lock
        lock_guard<mutex> lk( myMutex);
        if( myQueue.empty()) return false;
        //  Move from queue
        t = move( myQueue.front());
        //  Combine front/pop
        myQueue.pop();

        return true;
    }   //  Unlock

    //  Pass t byVal or move with push( move( t))
    void push( T t)
    {
        {
            //  Lock
            lock_guard<mutex> lk( myMutex);
            //  Move into queue
            myQueue.push( move( t));
        }   //  Unlock before notification

        myCV.notify_one();
    }

    //  Wait if empty
    bool pop( T& t)
    {
        //  (Unique) lock
        unique_lock<mutex> lk( myMutex);

        //  Wait if empty, release lock until notified
        while( myQueue.empty()) myCV.wait( lk);

        //  Re-acquire lock, resume, combine front/pop
        t = move( myQueue.front());
        myQueue.pop();

        return true;

    }   //  Unlock

    void clear()
    {
        queue<T> empty;
```

```
70          swap(myQueue, empty);
71      }
72  };
```

The accessor *empty()* and the non-blocking *tryPop()* are unchanged. The method *push()* gets one additional line of code "myCV.notify_one()" that awakens one thread waiting on the condition variable so it consumes the new element. Note that we unlocked the mutex *before* the notification. When a CV awakens a thread, the thread must acquire the lock before it can do anything else. If the lock is unavailable, the thread goes back to sleep until the lock is released. This would cause an unnecessary overhead unless the lock is released prior to notification.

We have a new method *pop()* that is called by consumer threads to dequeue the front element. The consumer thread starts by acquiring a lock on the concurrent queue's mutex. Multiple threads cannot manipulate the queue at the same time. It locks the mutex with a *unique_lock* rather than a *lock_guard* for a reason that will be explained shortly. If the queue is empty, the consumer thread calls *wait()* on the CV and goes to sleep until notified.

Note that we wrote:

and not

This is the correct syntax due to *spurious wakes*. Spurious wakes occur when a thread awakens without being notified, something that may happen due to low-level details in the OS and hardware-specific implementation of CVs. Because of spurious wakes, when a thread awakes on a CV, it should always start with a check that the condition for the

notification (the queue is no longer empty) is satisfied, and go back to sleep otherwise. In practice, this is achieved by expressing the condition for waiting on the CV with a rather than an .

Calling on the CV puts its caller to sleep and *releases its lock*. Evidently, a thread should never hold onto its locks when it sleeps. Nobody could push an element and awaken the waiting thread if it had kept its lock. Therefore, the CV's method takes a lockable object as an argument and releases the lock when it puts the calling thread to sleep. A is not a lockable object. It does not define a method . This is why a cannot be used with a CV and this is why we used a instead. The implementation of the CV also ensures that the thread reacquires the lock when it awakens on a notification. This is why we released the lock in prior to notification, so that the consumer may reacquire the lock with minimum overhead. Once it reacquires the lock, the consumer thread *moves* the front element (we know there is one at this point) into the argument to and removes it from the queue, all under the protection of the lock it reacquired on awakening. When exits, the destructor of the releases the mutex.

A consumer-producer queue is the centerpiece of a thread pool. In this case, the elements in the queue are the *tasks* to be executed in parallel. Worker threads are the consumers and wait for tasks to appear in the queue. Every task pushed into the queue awakens one worker thread to execute the task. When multiple tasks are pushed into the queue, worker threads dequeue and execute the tasks in order and go back to sleep when there are no more tasks to execute. A client pushing a task into the queue is guaranteed that a worker thread will awaken to execute the task, unless all workers are busy executing prior tasks, in which case the tasks are executed by worker threads, in parallel, in the same order they were pushed into the queue.

Our producer-consumer queue is *almost* complete. We need to notify all waiting threads in its destructor, so they may awaken before the CV

is destroyed. A CV, like a mutex, should *never* be allowed to die while threads are waiting on it. We define an method that pulls all consumer threads out of waiting and invokes it on destruction:

We also modify so the threads awakened by a call to don't go back to sleep immediately and exit instead (hence also releasing the mutex):

Below is the final listing for the concurrent producer-consumer queue in ConcurrentQueue.h:

3.14 ADVANCED SYNCHRONIZATION

C++11 provides only two synchronization primitives: the mutex and the condition variable. Parallel programming classically also involves more advanced synchronization constructs such as semaphores, barriers, and shared mutexes. The standard threading library does not provide these constructs.[11] They can all be developed by combining mutexes and condition variables.

Although we don't need these constructs in the remainder of the publication, or in typical financial applications, readers are encouraged to attempt implementing them on their own, as this is an excellent opportunity to manipulate mutexes and condition variables and earn a deeper understanding of concurrent programming.

Semaphores

A semaphore is essentially a mutex with added flexibility.

First, a semaphore may be constructed locked. A mutex is always constructed unlocked. Hence, a semaphore may forbid access to a critical section until the gate is explicitly open, something a mutex cannot do.

Second, a semaphore can grant access to critical code to more than one thread at a time. It has a parameter so that no more than threads may acquire a lock and execute the critical section concurrently. When threads are executing the critical section, another th thread requesting the lock waits until one of the threads leaves the critical section and unlocks the semaphore. For example, the management of multiple downloads typically uses semaphores to limit the number of simultaneous downloads and avoid network congestion.

The Semaphore class signature is as follows:

For historical reasons, is called on a semaphore and is called . We define and anyway so the semaphore implements the lockable interface and works with RAII locks. We also template it in the mutex type so it may be instantiated with a mutex or a spinlock.

Barriers

Barriers are another classical synchronization construct. Threads that call on a barrier wait there (actively or not depending on whether the barrier is implemented with a mutex or spinlock) until threads checked out, at which point the threads resume execution.

The signature is as follows:

Calling puts the thread to sleep or active wait until threads made the call; then all the threads resume. The reduction algorithm described in the Introduction typically applies barriers to synchronize threads so that all threads complete a step in the algorithm before any thread starts working on the next step.

Shared locks

Much more difficult is the correct implementation of a shared mutex, a construct particularly useful in the context of concurrent data structures. Imagine we implement a concurrent container. A simplistic implementation would protect all kinds of data access with locks. Even read access must be protected because a thread cannot be allowed to read data while another thread is writing into the container. So every type of access must be protected, as we did with our concurrent queue, which results in a completely serialized data structure that only one thread may manipulate at a time.

This is obviously inefficient and may be improved with the realization that multiple *reader* threads can access data concurrently, as long as no *writer* thread is manipulating data at the same time. Therefore, reader threads could acquire a *shared* lock, whereas writer threads would need an *exclusive* lock. A *shared mutex* exposes a method for the acquisition of a shared lock, in addition to the usual for the obtention of an exclusive lock. We would also need a class that provides RAII management for shared mutexes in the same way a manages a standard mutex. That part is easy, but the correct development of a shared mutex is more involved.

Shared and exclusive locks are granted according to some policy. One possibility is to prioritize reader threads, since multiple ones may access the data simultaneously. The problem with that is, as long as at least one reader is accessing data, writers are denied access. In a situation where readers and writers come in continuously, writers will *never* be granted access and will wait forever. This is sometimes called *thread starvation*. Similarly, priority might be given to writer threads; for instance when a writer requests access, no further readers are allowed and the writer may come in as soon as the present readers cleared the section. This policy would starve reader threads. Writers need access exclusive from readers and other writers, so they access the structure one at a time. As long as there are writers in the queue, no readers will be allowed.

Hence, some accountancy is necessary here. Priority must depend on the number of readers and writers in the queue, and perhaps how long the threads have been waiting. Like a security employee at the gate of a Disneyland attraction, the shared mutex must balance the shared and exclusive passes as best it can.

3.15 LAZY INITIALIZATION

The lazy initialization idiom is a frequently used programming pattern, including in finance. When some object, like a database connection, or an interpolated curve, is expensive to initialize, and may or may not be used during the execution of the program, it is common sense to postpone the costly initialization to the first use.

Here is an illustration of lazy initialization in the case of a curve. Lazily initialized curves are frequent in finance in the context of splines. The initialization of splines is costly and a natural candidate to laziness.

The code should be self-explanatory, especially with the comments.

This is a perfectly valid curve object in a serial context. But it cannot be used concurrently. Picking a value on the curve is thread safe, of course, and we note that the related method is . But initialization is not, and we note that the public , where lazy initialization is implemented, is *not* .

How can we make that curve safe for concurrent use without sacrificing this very useful lazy initialization feature? A natural solution would be to protect initialization with locks:

(note that we used a so we can unlock the mutex *before* the call to the thread safe . Locks should always be released ASAP.)

There is, however, a problem with this solution: it is *extremely* inefficient. A lock must be acquired *every time* is called, although it is necessary only once, for initialization. To acquire a lock is expensive, typically more expensive than reading a curve; hence that solution is not satisfactory at all. In fact, it is worse than the sacrifice of lazy initialization.

Another solution called *double checked locking* has been proposed, and although that solution is *broken*, it may still be found in literature:

This code does not work: a thread may be checking under the first, unlocked check while another thread is writing into , executing “myInit=true” under the protection of a lock. This of course would result in a data race. It could be fixed by making an atomic boolean type, and call an atomic to conduct checks,

like we did for spinlocks, but this is not necessary. Lazy initialization is so common that the standard threading library provides a dedicated solution with the type and the function , both defined in the header <mutex>. This construct resolves the problem and guarantees that the *callable* argument to is called once and once only. One possible implementation is with atomic double checked locking, as discussed above. This is how it is applied to fix our curve class:

The arguments to following the follow the callable pattern described in Section 3.2.

3.16 ATOMIC TYPES

We have briefly introduced atomic types when we coded the spinlock mutex in Section 3.8. More specifically, we introduced the type that behaves like a for all intents and purposes, but provides a method that sets the state of the atomic bool and returns its former state, *atomically*, that is in such a way that another thread cannot read or write the atomic object during the whole operation, providing a guarantee against data races.

Another typical example is counting the number of times an object is accessed. For instance, a counter could be put on a curve and increase every time the curve is read. We define a basic counter class below:

It should be clear that this counter is not thread safe and cannot be manipulated concurrently. One problem is that operations *a la* ,

, , , etc. are not atomic. For instance, a statement like “x++” really translates into multiple instructions:

1. Read the state of .
2. Add 1.
3. Set the state of to the result.
4. Return the *former* state of .

The problem, of course, is that another thread may step in during that sequence of operations and cause a race condition. Our counter may be made thread safe, at the cost of performance, with locks:

Atomics provide an alternative to locks that may be lighter, faster, and closer to the metal. We can implement our thread safe counter with atomics in place of locks, with a considerable performance gain. In fact, the atomics-based counter is almost identical in performance to its unsafe counterpart:

The atomic template can be instantiated with any type : bool, int, double, any struct or class type, or, importantly, a pointer or a reference to any type: is an atomic *pointer* to an object of class type , contrary to , which is the atomic equivalent of an object of type MyClass. Atomic types provide a number of methods that are guaranteed to execute atomically. The most common methods are:

- **store()** sets the state of the object to the value of the argument.
- **load()** returns the state of the object.
- **operators *a la* , , , , etc.** are guaranteed to execute atomically.

- **exchange()** sets the state of the object and returns its former state, atomically.

What is an atomic object? The generic template is a *wrapper* to any type that uses locks to guarantee the atomicity of certain operations. In this respect, our counter code with atomics should be identical to the version with locks. However, and this is the whole point with atomics:

The template is *specialized* for types that support atomic operations on OS or hardware level, including:

- **All integral types** like bool, char, int short, long, unsigned and friends.
- **Pointers and references to any type** pointers are *always* integral types.
- **Other POD types** like float, double, etc. depending on implementation, OS, and hardware.

For those supported types, atomic operations are delegated to the OS/hardware and implemented without locks, at a fraction of the cost of locks. Whether a type supports lock-free atomic operations may be checked with a call to on an abject of type . Again, all pointer and referenced types are lock-free.

Applications include spinlocks, counters, and reference counts for smart pointers. More generally, atomics provide a faster alternative to locks in many situations. It is even possible to program a concurrent queue without locks (!) with atomic pointers, and a detailed example is given in [15]. Such developments, called *lock-free algorithms and data structures*, are *very* advanced and make a difference in specific cases, to our knowledge, not relevant to finance. We only use atomics in simple cases like reference counters or spinlocks.

3.17 TASK MANAGEMENT

The standard threading library also provides, under the header <future>, a number of convenient constructs for sending *tasks* for execution on another thread, and monitor the execution from the dispatcher thread: the function, the and classes, and the construct. The free function is a one-stop solution that sends a callable to execute on another thread. The other, lower level constructs, provide facilities for inter-thread communication but not the management of the threads themselves. Combined with thread management explained in Section 3.2, these constructs provide a convenient and practical way to manage the execution of tasks on a set of worker threads in a thread pool, of which they constitute an important building block.

Async

The first and simplest construct C++11 provides for asynchronous task execution is the free function , defined in the header <future>. This function takes a callable argument, sends it to be executed asynchronously, and returns a so the caller thread may monitor its execution.

The free function offers the simplest form of parallelism in C++11. To some extent, it is comparable to OpenMP's loop parallelism in terms of simplicity: instead of calling a function, wrap the call within and it executes on a parallel thread. The asynchronous execution is controlled by the implementation, delegated to the OS, and does not offer much control or flexibility over the process. It is perfectly suited in simple cases where we execute a simple function on a parallel thread while performing other tasks on the main thread. A typical application is perform an expensive task in the background without blocking the application. For example, a database query could be executed asynchronously with without freezing the GUI. An example is given below:

The arguments to follow the callable pattern described in Section 3.2. We may pass a function, function object, member function, or more conveniently a lambda. The function also accepts an optional parameter *before* the callable to specify the *launch policy*:

- **async(launch::async, callable,...)** forces execution on a parallel thread.
- **async(launch::deferred, callable,...)** defers execution until or is called on the returned .
- **async(callable,...)** when no launch policy is specified, the implementation "decides" on the best asynchronous execution based on hardware concurrency, how busy the CPUs are, and so forth.

In all cases, the call to returns immediately and provides the caller with a templated on the type of the result of the task: in our example, the task returns a double; hence the call to returns a .

Futures and promises

The free function encapsulates thread creation and management; hence its simplicity. But it does not offer sufficient flexibility to achieve maximum performance or multi-thread sophisticated algorithms. In these cases, we must manage the process ourselves, but the standard library does provide the necessary building blocks for task management. Futures and promises, also declared in the header <future>, provide an inter-thread communication mechanism, whereby threads communicate to one another the result of their work. These constructs are not *synchronization primitives*. They allow threads to communicate, but not *control* one another in any way, and their implementation is in OS-independent, standard C++. Developers can implement their own futures and promises with mutexes and con-

dition variables and this is indeed a recommended exercise. With futures and promises, threads can communicate:

1. Whether a computation is complete.
2. Whether the computation completed normally or threw an exception.
3. The result of the computation or the thrown exception.

Futures Futures are on the *receiving* end of the communication and live on the dispatcher thread. We just met them with . The caller to (dispatcher thread) starts a computation, possibly on a different thread (computer thread), and receives a future for monitoring this computation. The dispatcher thread can call the method on this future and block until the computation completes (or return immediately if the computation already completed). This guarantees that the instructions following a call to will not execute before the asynchronous task completed.

The same result is obtained by a call to the future's method , which, in addition to blocking the caller until the asynchronous computation completes, returns the result of the computation[12] if it completed normally, or *rethrows* any exception thrown during the computation.

Futures may be moved but not copied.

Promises The function communicates its status and the results of the asynchronous computation (including exceptions) to its associated future. The promise, also declared in <future>, is a lower level mechanism to communicate this information to a future. The promise is on the *broadcasting* end of the communication and lives on the computer thread.

A promise object has an associated future, accessed by a call to the promise's method, and that subsequently receives com-

munications from the promise. The promise has two methods, and , to communicate completion, along with either a result or an exception. Prior to a call to one of these methods, the status of the task is incomplete, and a call to or on the associated future from a different thread blocks that thread. A call to either or on the promise wakes the thread (if any) waiting on the associated future.[13] After such call on the promise, a thread calling on the associated future will not block, and a call to returns the result or rethrows the exception immediately.

The implementation of generally uses promises. Promises are flexible, low-level constructs that may accommodate many kinds of inter-thread communication architectures. As an example, we develop our own version of with promises. Our code is simplified in the sense that launch::async is compulsory (it is by far the most frequent use) and the callable takes no arguments (which is not a problem if we use lambdas: lambdas capture their environment):

The code illustrates how to use promises. This implementation takes a callable argument and wraps it into a ; see Section 2.2. The first thing it does is construct a , templated on the return type of the task, and held by shared pointer; see Section 2.4. Shared pointers automatically destroy objects and release memory when the *last* pointer referencing the object exits scope. Shared pointers is what guarantees that the promise remains alive during the asynchronous execution of the task, even though returns immediately.

The code then invokes the method on the promise. As the name indicates, this method returns the associated to the , the object that may read communications from the promise from a different thread.

Next, we create a new thread to execute in parallel, with the callable and the *shared pointer* on the promise as arguments, and return the to the caller of . Note that we detach the thread immediately, so the execution of continues in the background when returns. This breaks the thread handle, so we lose the connection to the computer thread (see Section 3.2), which is fine; we have a future to directly monitor the asynchronous task.

The function completed and returned the future to the dispatcher thread. The execution of on the computer thread started and may still be running. The promise is used from within so it must remain alive throughout the computation. It was passed as an argument to the thread constructor by shared pointer, and the thread constructor forwarded the shared pointer to . Therefore, as long as is running, there is one shared pointer that references the promise and the promise remains alive. When returns, the last shared pointer on the promise exits scope, the promise is destroyed, and its memory on the heap is released.

The function executes the callable in a try-catch block in case the callable throws, and invokes the method on the promise. In case the callable throws, we catch the exception and set it with the promise's method . The call to either or on the promise causes *a prior or future call* to either or on the associated future (the one we got earlier with a call to on the promise), maybe from a different thread, to unblock, and in the case of , return the result of or rethrow the exception of .

Promises provide flexible means to control the state accessed by their associated futures from different threads, and may be applied to communicate between threads in many kinds of parallel designs. We could

develop a thread pool with promises, but the perhaps higher level *packaged task* is better suited for this purpose.

Packaged tasks

A packaged task, also defined in <future>, is a convenient construct that encapsulates the promise – future mechanics for us, like , but, contrary to , it does not manage threads for us. The type is a *callable* type. It wraps a callable and defines an operator to invoke it. It is templated by its return and parameter types, like .

A packaged task is constructed with a callable of any type. It provides an associated future (with its method), and when the packaged task is invoked on a thread, its wrapped callable is executed on that thread. When execution completes, however, the packaged task does *not* return a result or throw an exception; instead, it sets the result or the exception on a shared state, like a promise. Packaged tasks are typically implemented with promises. The shared state is accessible through the associated future, possibly from a different thread.

A packaged task, like , implements type erasure so it may wrap all types of callables. Its creation involves an allocation. The implications for us are that:

1. We should not use this type at low level, and especially not within repeated code.
2. We should avoid the creation of a large number of small tasks, and instead attempt to combine small tasks into larger ones and send those for parallel execution. We will discuss this point again in Section 3.19.

Packaged tasks are non-copyable, moveable types. We illustrate how they work with an example. Say we have a function that reduces a

vector into a number; for instance it could compute its average or standard deviation and throw an exception if something goes wrong:

Say that we have a number of (large) vectors to process:

We want to process the vectors on parallel threads. This is exactly where we would use a thread pool, although we don't know how to do that yet, so, instead, we apply basic thread management for now with the convenience of packaged tasks:

The processing of each vector is a separate parallel task. For every vector, we create the packaged task from the callable , which, in this case, is a free function taking a vector as a parameter by const reference, and returning a double as the result of its processing. Hence, the packaged task template is instantiated with .

Once the packaged task is created, we keep track of its associated future. Like promises, packaged tasks expose their future with . Finally, we create a parallel thread for the execution of the task. We pass the task itself to the thread as the thread constructor's callable argument, along with the argument of the task, the th vector to process. We note that the task must be explicitly moved, since it cannot be copied, and that the argument must be explicitly passed as a reference, otherwise it would be passed by copy, as explained in Section 3.2.

We detach the new thread straightaway. We don't need a handle on the thread, since we have a future on the task. We don't store the thread handle into a variable; hence, the temporary handle returned by the thread constructor is immediately destroyed, which would terminate the application if we had not detached the thread.

The task is the callable that is passed to the thread constructor for execution, so the task's operator will be invoked on the parallel thread with its (by const reference) argument, the th vector to be processed. This will execute the wrapped callable on the parallel thread and set its result or exception on a shared state, accessible with the future from the main thread:

Calling on the future blocks the caller until the execution of the corresponding task completes on the parallel thread: more precisely, it blocks until the packaged task sets the result or the exception on the shared state. If this is already done, returns the result immediately. Otherwise, the caller waits in idle state. Any exception thrown during the execution of the task is rethrown on the thread calling . Our code would catch and handle this exception with the hypothetical function .

What makes packaged tasks particularly well suited to thread pools is that, as illustrated in the previous examples, packaged tasks provide futures so callers may monitor the asynchronous tasks.

3.18 THREAD POOLS

The singleton pattern

We are finally ready to build the thread pool we will use in the remainder of this publication. A thread pool is designed to encapsulate thread logic in an application. It controls a collection of worker

threads. It is convenient to implement the well-known *singleton* pattern (see [25]) so that the application holds one and only one instance of the thread pool. The singleton instance is created when the application starts, destroyed when the application exits, and remains globally accessible in the meantime. One benefit is that the thread pool is globally accessible so functions don't have to pass it to one another as an argument.

This pattern guarantees that there may be only one instance of a thread pool in the application. It is constructed when the application starts, and destroyed when it exits. This unique instance is accessed with the static method:

which provides a pointer for the manipulation of the thread pool.

Task spawning

The purpose of the thread pool is to encapsulate thread logic, so that algorithms divide into logical slices, or *tasks*, *spawned* for parallel execution into the thread pool, without concern of how the tasks are scheduled over threads. Algorithms have no knowledge or control of what thread executes what task: this is the private responsibility of the thread pool. Algorithms just know that the pool executes the spawned tasks in parallel and in the same order they are spawned into the pool.

Hence, the thread pool exposes a method for spawning tasks. What kind of tasks will clients be spawning? We could design it in a very general manner so that clients may spawn all kinds of callables, but this is unnecessary. It is more convenient to always spawn lambdas, which capture the information they need, hence, don't need arguments, and set results directly in memory, hence, don't need to return anything. However, is not a valid return type for C++11's task

management constructs. Therefore, all our tasks return a , by convention if they execute correctly, throwing an exception otherwise:

The thread pool internally manages a (concurrent) queue of tasks, so to spawn a task really means push it into the queue. The code for the concurrent queue designed in Section 3.13 is in the file ConcurrentQueue.h:

Client code calls to send a task to the thread pool for a parallel execution, which pushes the task in the pool's queue, and returns a future to monitor progress. Note we tasks around to avoid the potential overhead of copying tasks that capture large amounts of data.

Worker threads

The worker threads pop tasks from the queue and execute them in parallel. We recall that the method on the concurrent queue causes the caller to wait *in idle state* until a task is pushed into the queue, which automatically awakens one of the waiting threads to consume the task. This is the keystone of the thread pool's design: its worker threads wait in idle state until client code spawns tasks. The worker threads are created when the application starts, and wait in the background for clients to spawn tasks. Spawning a task does not involve any thread creation overhead, because the threads are already there, waiting for jobs to execute. Further, they don't consume any resources waiting so their presence in the background and readiness to execute the spawned tasks is free.

We can now write the function which each one of the worker threads executes. All they do is repeatedly pull tasks from the queue and execute them. When the queue is empty, causes them to sleep until a task is pushed in the queue, causing one waiting thread to awaken, as seen in Section 3.13, and consume the task. It follows that never exits so the worker threads remain alive until the application exits.

The starter of the thread pool (called once when the application starts) creates the worker threads and sets them to execute .
How many worker threads shall we create? We find it best to create, by default, as many threads as we have hardware threads on the machine running the application, *minus one* so one hardware thread always remains available for the main thread. The number of threads can be changed by the client code, for example, for debugging and profiling, as explained in Section 3.20.

This is all we need. The mechanisms in the concurrent queue ensure that the machinery works as planned. We just need to add a small number of convenient features.

Interruption

First of all, we cleanly dispose of the threads when the pool is destroyed, otherwise the destruction of the pool would cause the application to crash. That would not be a major problem, since the singleton pattern guarantees that the pool is not destroyed before application exit. But to let the application crash on exit is poor development standard, so we program a clean disposal instead. This means that we must interrupt all waiting threads, which somewhat changes our code:

We added a bool that breaks the infinite loop in when true so threads may exit. The interrupter is false on construction, and only a call to makes it true, so the worker threads remain trapped in until the thread pool is stopped, either explicitly, or in its destructor, activated when the application exits. The method also interrupts the queue, awakening the waiting threads so they may all exit, and waits for them to join.

Thread local indices

It is also useful that each thread knows its own index among the worker threads. We achieve this with thread local storage, as explained in Section 3.4. We number worker threads 1 to , leaving number 0 for the main thread. We also expose the number of threads in the pool:

We added a data member as a static integer marked as thread local, which means that every thread holds a different copy of it. The function sets it to the index (passed from the starter) of the executing worker thread. Since runs on the worker threads, each worker thread sets *its own copy* of to its own index. As a result, when the static function:

is called on a worker thread, it returns the index of the caller. We also want the main thread to return 0, so we set the initial value (on the main thread) to 0 in ThreadPoool.cpp:

Active wait

Finally, we add a very useful feature that allows the main thread to help executing tasks while waiting for others. The typical utilization of a thread pool is one where the main thread spawns a large number of tasks and waits on the corresponding futures for the worker threads to complete the tasks. The main thread occupies an entire hardware thread, and its hardware thread will not be doing anything useful while waiting. That is a waste of CPU. Instead, we implement a method causing the caller thread to help executing tasks in the queue while waiting on futures:

This concludes the development of our thread pool. This is a very basic thread pool. It could be improved in a lot of ways, including a high-performance concurrent queue, and *work stealing*, a design whereby each thread works with its own queue and helps other threads with their jobs when their own queue is empty; [15] develops these themes and more and discusses the production of high-performance thread pools, capable of dealing with a large number of tasks, including nested spawning of tasks from within the jobs executed on the worker threads.

Financial algorithms, however, don't typically require such advanced functionality. Our basic thread pool, in particular, provides everything we need to run parallel simulations. We tested more sophisticated, high-performance designs, and they did not make a difference in situations of practical relevance.

The complete code is reproduced below from our repository files ThreadPool.h and ThreadPool.cpp.

3.19 USING THE THREAD POOL

We will use the thread pool in rest of the publication. In fact, the thread pool is the only parallel construct we are going to use in our multi-threaded programs. For future reference, we are providing here a summary of its functionality, and a user's guide for the client code. We also highlight some important properties of the thread pool.

Instructions

The thread pool is designed for the convenience of client code. Client code starts the thread pool when the application opens. The pool sits in the background for the lifetime of the application. It is accessible from any function or method by static pointer, without the need to pass it around as an argument. It is destroyed when the application closes. It doesn't consume any resources unless used.

The instructions for using the thread pool are articulated below:

1. Include “ThreadPool.h” on the files that use it.

2. Access the thread pool by static pointer in any function or method.

   Start it as follows (when the application opens):[14]

3. Give the pool jobs to conduct in parallel, and remember that tasks must return a bool: The pool guarantees that task execution *starts* in the spawning order.

4. Wait for asynchronous tasks to complete, generally helping with the tasks while waiting:

5. After that loop, all the tasks spawned to the pool completed and the worker threads are back to sleep. The main thread can safely use the results.

Mutable objects

The thread pool facilitates the creation and the management of copies, one per thread, of the *mutable objects*, the objects that are not safe for concurrent access. Mutable objects were introduced in Section 3.11, and their management is illustrated in the parallel code of Chapter 7.

The number of worker threads can be accessed with a call to , so we know how many copies we need. Every thread can identify its own index in the pool by calling , which returns an index that depends on the thread making the call: 0 for the main thread, 1 to for the worker threads. This allows threads to work with their own copies of the mutable objects.

When concurrent code modifies objects with calls to non- meth-ods, we can avoid expensive and efficiency damaging locks by making copies of these mutable objects before multi-threading:

In the concurrent lambda, we work with the copy belonging to the executing thread, which results in a thread safe code without locks:

We apply this pattern systematically in concurrent code in the rest of the publication. In addition to mutable objects, the exact same pattern allows to preallocate working memory for different threads. For instance, say the concurrent code needs working memory of size in a vector :

We have an allocation on line 10 in the concurrent code. As explained in Section 3.11, allocations generally involve hidden locks and damage parallel performance. The solution is to preallocate working memory in the initialization phase, before multi-threading. But the different threads cannot write into the same vector ; that would cause race conditions. On the other hand, to lock the piece of code that consumes would damage performance:

Instead, we preallocate workspace *for every thread*, like we made copies of mutable objects:

This is how we concretely achieve lock-free concurrent code, with a thread safe design that does not damage parallel efficiency. The thread pool's dedicated facilities makes the management of thread-wise copies mostly seamless.

Load balancing

Another important benefit of our thread pool is that it implements automatic load balancing. Client codes sends tasks in a queue, and the worker threads pop the tasks and execute then in the same sequence. When a worker thread takes a long time to complete a task for whatever reason[15] the other threads pop and execute tasks in the meantime. As long as there are tasks in the queue, all the threads work at 100% capacity and never wait for one another. In parallel computing, this is called load balancing, and our thread pool implements it by design.

We generally have as many worker threads as we have (logical) cores, minus one for the main thread. Load balancing only works when we have many tasks. Evidently, if we send 16 tasks into a thread pool of 15 worker threads (plus the main thread, who helps), everybody waits for the slowest to complete. If we send a large number of tasks, load balancing comes into play and a linear improvement may be achieved.

To enable load balancing, the number of tasks must significantly exceed the number of threads. So, we should split our parallel algorithms into many tasks. How far should we split? For instance, in a transformation that applies a math function to many numbers, should we spawn each single number as a separate task? The answer is no. There is an overhead (of the order of a few tens of microseconds) to spawn a task. The overhead remains negligible as long as the duration of the task is orders of magnitude longer. This is why, as a rule of thumb, we spawn tasks of around a millisecond. This is not a firm rule. It could be 0.1 ms or 10 ms. But we must refrain from spawning gazillions of microscopic tasks, or our cores spend more time spawning than computing.

Therefore, the rule (of thumb) is to split parallel algorithms into as many tasks as possible, keeping the duration of each task around 1 ms. The exact rule depends on the context. In our simulation code of Part II, we spawn Monte-Carlo paths in batches of 64.

3.20 DEBUGGING AND OPTIMIZING PARALLEL PROGRAMS

Parallel programs are notoriously hard to debug and optimize. In this final section, we briefly introduce some techniques to help identify and eliminate race conditions, and improve parallel efficiency.

Debugging and profiling serial code

Before we discuss the debugging and optimization of parallel code, we briefly introduce the debugger and profiler bundled with Visual Studio in the context of serial code. We expect readers to have some experience with these from lectures and programming. When this is not the case, what follows should help get started.

The Visual Studio Debugger When the program crashes or produces incorrect results, we have a "bug." Bugs can be found by inspection of the code, but we can identify them faster, and more conveniently, with a debugger.

To start Visual Studio's debugger, compile the program in debug mode.[16] Set a breakpoint on the first line of the suspected block of code with the "Toggle Breakpoint" item of the Debug menu, or press F9. Start the program through the Debug menu, Start Debugging or press F5. The program executes and pauses on the breakpoint. Visual Studio displays its state. The code window shows a yellow arrow on the left of the current line of code. The call stack window displays the current sequence of function calls. The watch window shows the current values of variables and expressions that users type in the name column, including classes and data structures.

Navigate the program with the Debug menu: continue to the end or the next breakpoint (F5), advance to the cursor (ctrl + F10), to the next line (F10), step into the function called on this line (F11), execute and exit the current function (shift + F11), and so on. Double click on a

function on the call stack to see the current state in that function's environment, with execution still paused on the same line of code. Toggle breakpoints on and off and add new breakpoints while debugging (F9) or create conditional breakpoints by right clicking on the breakpoint, and select “condition...”.

The debugger provides a vast, and sometimes overwhelming, amount of information. We briefly described the most common ones. With these basic tools, we can execute the program step by step and examine its state as the execution progresses. The environment is simple, clear, and user friendly, and generally helps find bugs very quickly.

The Visual Studio Profiler After the program produces correct results, we may attempt to increase its execution speed. The first thing to do is compile in release mode with all optimizations turned on. Visual Studio switches most optimizations on by default, but the following may be set manually on the project's properties page,[17] on the release configuration for the 32 bit or 64 bit platform, whichever is relevant:[18]

- **Optimization** in C/C++
  Optimization = Maximum Optimization (Favor Speed)
  Inline Function Expansion = Any Suitable
  Favor Size of Speed = Favor fast code
- **Code Generation** in C/C++ Enable Enhanced Instruction Set = Advanced Vector Extension 2[19]
  Floating Point Model = Fast
- **Language** in C/C++ Open MP Support = Yes
  C++ Language Standard = ISO C++ 17 Standard

To accelerate the program, we must first identify *where* most execution time is spent. This is the purpose of a profiler, to measure CPU time spent in various parts of the code during an execution.

To start profiling, compile the program in *release* mode with debug information switched on, both on the compiler and the linker. It should

be the case by default. For avoidance of doubt, check the project properties page for the release configuration. “C/C++ / General / Debug Information Format” should be set to “Program Database,” and “Linker / Debugging / Generate Debug Info” should be set to “Generate Debug Information.”

Start the program from the Debug menu, profiler, performance profiler. Select performance wizard, press start. Skip through the four pages of the wizard window, leaving all settings to defaults, press finish. The program starts. When it completes and exits, Visual Studio displays various statistics. Press “Show Just My Code” in the upper-right corner, and start navigating the functions that consume most CPU time. The profiler shows CPU time spent on every line with a color code from white (negligible CPU time) to red (major CPU time).

The profiler holds much more information than what we introduced. Knowing exactly where the program spends time is enough to get started, and it is priceless information. Armed with these statistics, we can investigate and attempt to accelerate those functions and blocks of code that consume most CPU time, ignoring others:

- Is the algorithm implemented in this function optimal?
- Do we make unnecessary copies? Are we passing arguments by value? Did we forget to implement move semantics?
- Do we allocate memory at low level, explicitly or by construction or assignment of containers and data structures?
- Do we correctly call STL algorithms in place of hand-crafted loops?
- Do innermost loops work on coalescent memory? Are they vectorized?[20]

And of course: is this time-critical code a candidate for multithreading?

It is important to make these investigations with the measures provided by the profiler and not developer guesses. Even the most experi-

enced programmers are often unable to correctly predict the bottlenecks in a C++ program. With modern hardware and compiler optimization, bottlenecks often land in unexpected places, due to cache, memory, and so forth while the compiler optimizes the trivial bottlenecks away on its own.

Debugging and optimizing parallel code

To debug and optimize parallel code is *orders of magnitude* harder. We don't cover this vast subject in detail, instead offering general advice based on our experience, and tips that helped us develop stable, efficient parallel programs, including those in Parts II and III.

Debugging parallel code If the serial code returns the correct results and the parallel code doesn't, the cause has to be one of the following two: either the parallel logic is flawed, or we have race conditions. To find out, *run the parallel code serially*. Assuming the program uses the thread pool, start the pool with zero worker threads. All the tasks will be processed sequentially on the main thread, resulting in a serial execution of the parallel logic. With concurrency and associated problems out of the way, we can debug the parallel logic on its own.

If the results are still incorrect, the parallel code is bugged. Debug a serial run of the parallel code, same as serial code, following the steps of the previous paragraph.

If the results are correct on a serial run, but incorrect with multiple threads, we have race conditions. Race conditions can also be diagnosed directly in what they produce different results on successive executions. This is, however, not entirely reliable, because other flaws, like uninitialized memory, cause similar symptoms.

Debugging race conditions When tests confirm race conditions, we must locate the responsible pieces of code. To automatically detect race conditions is almost impossible. Visual Studio has no such tool.

The few free and commercial tools available are not reliable in our experience. The only way we found to debug races is with *manual code bisection*.

Serialize your entire concurrent code[21] by locking a static mutex on the first line and unlocking it on the last line. The code is now executed on one thread at a time, even in a multiple thread environment, so races cannot occur, and the results must be correct. Now, move the lock somewhere in the middle of the code. Are the results still correct? If so, we have no race in the first half of the code. Otherwise, we do have races there, and perhaps also in the second half.

Implement a classic bisection, narrowing down the protected part of the code, possibly with multiple mutexes protecting multiple regions, until all the races are identified one by one and corrected. This is a slow and frustrating process. Code must be recompiled between steps. It is well worth striving to write const-correct, thread safe code in the first place, as explained in Section 3.11, rather than spending hours and days hunting race conditions after the fact.

Profiling parallel code Once the parallel program returns the correct results, we can measure its parallel efficiency, which we recall is the ratio of the parallel acceleration to the number of physical cores, and attempt to improve it if sub-par, that is, materially under 100%. For instance, our initial parallel histogram code from Section 3.5 completed in over physical cores, compared to serial, a parallel efficiency of only 37.5%.

First, separate the parallel logic overhead from the concurrent inefficiency. Run the parallel program in serial mode, as explained earlier, and measure its execution time . How does it compare to ? The *logic efficiency* measures (the inverse of) the overhead caused by the additional logic necessary to implement the parallel code, irrespective of the number of threads or cores executing it. The

concurrent efficiency measures the acceleration per core of a parallel run *over a serial run of the same parallel code*.

Evidently, , but we now have two measures of parallel efficiency so we know where to focus our efforts: logic, concurrency, or both. Our parallel histogram code (upgraded with the thread pool) runs in over a single thread, hence and .

The parallel histogram is an embarrassingly parallel algorithm, multi-threaded at a high level. Its only logic overhead comes from task spawning and the general administration of the thread pool, the concurrent queue, the tasks, and the futures. In such cases, the expected overhead is around 10%. The expected logic efficiency is around 90%. A materially lower efficiency would indicate an incorrect application of the thread pool, probably too many small tasks with duration below 1 ms so the administration overhead makes a measurable difference.

Other parallel algorithms may require further additional logic. For instance, the parallel reduction algorithm, discussed in the introduction, implements recursion logic in addition to the reduction logic of the serial version. That may cause substantial overhead, and result in a lower logic efficiency.

Parallel logic may be profiled, and perhaps optimized, in the exact same way as serial code, profiling a serial run of the parallel code and repeating the steps of page 114. To what extent parallel logic may be optimized depends on the algorithm. In many instances, logic overhead is part of the algorithm's nature and cannot be improved.

The expected concurrent efficiency also depends on the parallel algorithm. For instance, the synchronization logic necessary for the parallel reduction, where threads must wait on one another to complete a step before the next step begins, is bound to reduce efficiency. For an embarrassingly parallel algorithm multi-threaded at a high level, we

expect efficiency around 100%. For the parallel histogram, 44% clearly indicates a flaw in the parallel code. We now know the problem of that code is false sharing, of course, but in general, the presence of a problem is easily diagnosed with simple efficiency measures.

It is worth considering trivial causes first, and inspect CPU utilization. On Windows, CPU utilization is displayed on the Task Manager. *Before* the parallel program is executed, CPU utilization must be close to 0. Otherwise, other processes are running and the parallel algorithm will share cores with them, obviously reducing efficiency. All the running processes must be stopped in order to measure efficiency accurately.

When the parallel part of the algorithm starts, all CPUs should hit 100% utilization and stay there continuously until it completes. At that point, the utilization of all CPUs should simultaneously drop to 0, safe for one, where the main thread aggregates and displays the results. It is worth running the algorithm for a long time (large data set, many Monte-Carlo paths,...), to obtain a clear CPU utilization chart. Those charts must *always* reproduce the pattern below:

When the chart looks any different, we have a problem so severe that it should be easy to find. If only one CPU is utilized, the program does not run in parallel. The thread pool did not start correctly or the tasks were not properly spawned. When multiple, but not all, CPUs are working, (assuming the thread pool was set to fully utilize hardware concurrency), we probably didn't fire enough parallel tasks. As explained on page 112, an insufficient number of parallel task, also damages load balancing, causing utilization to drop *one CPU at a time*, and not simultaneously, when the parallel part completes.

Try increasing the number of parallel tasks, splitting the algorithm into a larger number of smaller tasks.

In rare occasions, we have seen all CPUs at work but capped around 50–80%. Every time, this curious phenomenon was caused by severe lock contention, or repeated allocations, usually in the innermost loops. On the other hand, false sharing or a more limited contention leave CPUs working at 100% capacity, but damage efficiency all the same.

More generally, an incorrect CPU utilization always reveals a problem, but a correct utilization pattern absolutely doesn't indicate the absence thereof. If concurrent efficiency remains sub-par after the initial checks, we may have a deeper concurrent flaw. It helps to make more precise measures: run the parallel code on 1, 2, 3,… threads up to the number of logical cores. Measure execution times with threads and draw as a function of . The chart below displays this information for the parallel histogram, before and after fixing the false sharing problem.

With the correct code, we get a straight 45deg line up to the number 8 of physical cores, followed by a flattening when we hit the hyper-threaded cores. With false sharing, we obtain the same pattern, but with a more than twice lower slope around 20deg. With locks or allocations, we could obtain a *decreasing* curve.

When tests confirm concurrency flaws, we must identify and locate them. The most common concurrent inefficiencies include:

- **Locks** as explained in Section 3.11, including hidden ones, like memory allocation. The remedy is to remove all locks from the concurrent code, making copies of mutable objects instead, as explained on page 109, and move allocations out of the concurrent code, preallocating all necessary memory in the initialization phase before multi-threading.
- **False Sharing** as explained in Section 3.5, along with remedies.

- **L3 cache contention** for memory intensive parallel algorithms, like AAD, see Part III. Multi-threaded code processing vast amounts of memory may cause multiple threads to compete for the shared L3 cache. L3 cache contention is remedied by splitting processed data over a larger number of smaller tasks, so that each task works in parallel with a smaller amount of data. This may be hard to implement in practice, and may even require a redesign of the parallel algorithm. The parallel histogram would have to be completely rewritten in a way more complicated manner for the data to be processed over each interval by multiple tasks. In the context of AAD, this is achieved with check-pointing; see Chapter 13.

In principle, these problems can be profiled. False sharing causes L2 cache misses, cache contention causes L3 cache misses, and locks cause concurrency contention. All of these can be measured by Visual Studio's profiler. Unfortunately, besides for basic CPU sampling, the profiler is confusing and poorly documented. By contrast, Intel's well-polished and documented vTune Amplifier offers a more satisfactory experience for 800 USD. It is the defacto standard for the optimization of professional parallel software. This being said, results for cache and contention profiling remain hard to interpret. We generally find it easier to resolve those problems manually.

Code bisection, introduced for the debugging of race conditions, may help locate concurrency bottlenecks, too. Disable the first half of the concurrent code and see if efficiency improves. If so, the problem is located in the first half. Split it in two and disable half of it, and keep disabling narrower fractions of the code until the problems are identified one by one, and fixed.

Look inside the innermost loops. Hunt code that may result in locks. Find allocations, including creation and copies of data structures, and move them outside of the concurrent code. Look for code that writes into the heap and may cause false sharing.

To rule out allocations, install a concurrent lock-free allocator, such as the one bundled with Intel's free TBB library, and set it as the application's global allocator. See Intel's documentation on how to do that exactly.

To confirm L3 cache contention, try running the same algorithm on a smaller data set. If the problem persists, cache contention is not the problem. Cache contention problems grow with the size of the working memory per task. As mentioned above, it is resolved by splitting the algorithm into a larger number of smaller tasks, each manipulating smaller memory. This is generally hard to implement, and may take patience and creativity. Parallel AAD, in particular, manipulates vast amounts of memory and is vulnerable to cache contention. We apply a number of techniques in Part III to reduce the size of the tapes (working memory for AAD) and mitigate cache contention.

This completes our general introduction to parallel programming in C++, and the general part about programming. The next part applies these concepts and techniques, and our thread pool, in the context of Monte Carlo simulations.

NOTES

1 Those are not being built anymore. Even phones are multi-core and have been for years.

2 Unless the application uses a specific API to directly program into the OS, something outside of our scope.

3 The OS schedules threads “at best,” and there is no guarantee that threads will execute in parallel. In practice, we could verify that Windows scheduling is very efficient and uses hardware to its full potential. We have no doubt that this also applies to other major OSs.

4 With hyperthreading, the OS schedules the execution of threads on hardware threads and the CPU schedules hardware threads on cores.

5 Or to interfere with scheduling in any way.

6 L3 is what manufacturers advertise. When they claim that some high-end CPU has 25 MB cache, for instance, what this really means is 25 MB L3 cache. The sizes of the (much smaller) core specific L2 and L1 caches are generally much harder to find.

7 We never faced that situation in our years of developing multi-threaded financial code.

8 Or spinlocks or any type of lockable objects.

9 Which is not really a data structure, but an adapter over another data structure, typically a .

10 What prevents CVs to be implemented with mutexes is that only the thread that acquired a mutex can release it. Hence, mutexes, contrary to CVs, cannot be used for inter-thread communication.

11 Although shared mutexes, and their associated RAII shared locks, are planned for future C++ updates.

12 Contrary to , which return type is , the return type of is the future's template parameter.

13 If the thread was waiting on , the result is returned or the exception rethrown.

14 The pool's method takes the number of worker threads as argument. 0 is legal and indeed helpful; see our comments on debugging and profiling, Section 3.20. Default is the number of logical cores minus one for the main thread. The number of worker threads

can be changed at run time by calling to close the pool; then call with the desired number of threads.

15 This thread may land on a slower hyper-threaded core, or it may share its core with a background task on the OS, or some other program the user runs while waiting for the results, or this particular task may be longer than others.

16 Visual Studio's toolbar has a scroll-down menu to switch configuration between release (compile code for practical use, with all optimizations switched on) and debug (compile code for debugging, without the optimizations).

17 Right-click the project in the solution explorer (the project, not the solution), and select “properties.”

18 We generally compile programs in 64 bit, except when we export code to 32-bit Excel. The project in our repository builds a 32-bit xll and therefore compiles on the 32-bit platform. The file xlCpp in our repository includes a tutorial ExportingCpp2xl.pdf that explains how to export C++ code to Excel, and the project in the repository is built this way.

19 Provided the program runs on a recent chip, otherwise it crashes when started.

20 Write “/Qvec-report:2” in the “Configuration Properties/ C/C++ / Command Line/ Additional Options” box of the project's properties to find out.

21 The part of the code that is executed concurrently, written in the lambda spawned into the thread pool.