УДК 004.4’6:004.43

Krivuliak V. V., instructor

C++11/14 FACILITIES FOR HIGH PERFORMANCE COMPUTING

Zaporizhzhya State Engineering Academy, department SoAS

High performance (“parallel”) computing is a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently ("in parallel"). Parallel computing has become the dominant paradigm in computer architecture, mainly in the form of multi-core processors[1]. One direction is focused on utilizing multi-core CPU, the other – on utilizing GPU.

New standard C++11/14 introduced new features that allow programmers utilize multi-core hardware efficiently with a new thread library. This library includes utilities for starting and managing threads. It also contains utilities for synchronization like mutexes and other locks, atomic variables and other utilities.

When you create an instance of a `std::thread`, it will automatically be started. When you create a thread, you have to give it the code it will execute. The first choice for that is to pass it a function pointer.

```
void func(int& i, double d, const std::string& s)
{
   i++;
   std::cout << i << ", " << d << ", " << s << std::endl;
}
int main()
{
   int count = 0;
   std::thread t(func, std::ref(count), 12.50, "sample");
   t.join();
   std::cout << count << std::endl; //count is 1
   return 0;
}
```

The call to `join()` blocks the calling thread (in this case the main thread) until the joined thread finishes execution. Apart from the join method, the thread class provides a couple more operations: `swap()` exchanges the underlying handles of two thread objects, `detach()` allows a thread of execution to continue independently of the thread object. Detached threads are no longer joinable (you cannot wait for them). If the thread function returns a value, it is ignored. However, the function can take any number of parameters. Even though it's possible to pass any number of parameters to the thread function, all parameters are passed by value. If the function needs to take parameters by reference, the passed arguments must be wrapped in a `std::ref` or `std::cref`. Each thread has a single id allowing us to distinguish each of them. The std::thread class has a `get_id()` function returning an unique id for this thread. You can get a reference to the current thread with the `std::this_thread` variable.[3]

When the code that has to be executed by each thread is very small, you do not necessary want to create a function for that. In that case, you can use a lambda to define the executed by a thread. You can write the code using lambda easily [2]:

```
threads.push_back( std::thread( [](){ std::cout << "Hello from thread " <<
std::this_thread::get_id() << std::endl; } ) );
```

An important thing to note is that if a thread function throws an exception it will not be caught with a regular try-catch block. To propagate exceptions between threads you could catch them in the thread function and store them in a place where it can be accessed later. [3]

The other problem is synchronization of access to shared data. Interleaving describe the possible situations of several threads executing some statements. Even for three operations and two threads, there is many possible interleavings. The problem can also occurs when a thread gets preempted between instructions of the operation. There are several solutions to fix this problem: semaphores, atomic references, monitors, condition codes, compare and swap, etc. [2]

A `mutex` is a core synchronization primitive and it C++11 it comes in four flavors in the `<mutex>` header. It provides the core functions `lock()` and `unlock()` and the non-blocking `try_lock()` method that returns if the mutex is not available.

- `recursive_mutex`: allows multiple acquisitions of the mutex from the same thread;
- `timed_mutex`: similar to mutex, but it comes with two more methods `try_lock_for()` and `try_lock_until()` that try to acquire the mutex for a period of time or until a moment in time is reached;
- `recursive_timed_mutex`: is a combination of `timed_mutex` and `recusive_mutex`.

The `lock()` and `unlock()` methods should be straight forward. The first locks the mutex, blocking if the mutex is not available, and the later unlocks the mutex.

Explicit locking and unlocking can lead to problems, such as forgetting to unlock or incorrect order of locks acquiring that can generate deadlocks. The standard provides several classes and functions to help with these problems. The wrapper classes allow consistent use of the mutexes in a RAII-style with auto locking and unlocking within the scope of a block. These wrappers are:

- `lock_guard`: when the object is constructed it attempts to acquire ownership of the mutex (by calling `lock()`) and when the object is destructed it automatically releases the mutex (by calling `unlock()`). This is a non-copyable class;
- `unique_lock`: is a general-purpose mutex wrapper that unlike `lock_quard` also provides support for deferred locking, time locking, recursive locking, transfer of lock ownership and use of condition variables. This is also a non-copyable class, but it is moveable.

The constructors of these wrapper guards have overloads that take an argument indicating the locking strategy. The available strategies are `defer_lock` of type `defer_lock_t`: do not acquire ownership of the mutex, `try_to_lock` of type `try_to_lock_t`: try to acquire ownership of the mutex without blocking, `adopt_lock` of type `adopt_lock_t`: assume the calling thread already has ownership of the mutex. [3] Developer can use any strategy to achieve the desired behaviour.

Conclusions:

The C++11 standard enables C++ developers to write multi-threading code in a standard, platform independent way, it doesn't clear the problems concerning parallel programming. It will not help in breaking the limit of theoretical speed up described by Amdahl's law. The speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program. Also, many technologies still support only C++98. The standard now includes things like portable thread affinity - this definitely does not play well with other threading paradigms, which might provide their own binding mechanisms that clash with those of OpenMP. Once again, the OpenMP language is targeted at HPC (data and task parallel scientific and engineering applications).

Sources:

1. Parallel computing [Web – resource], Parallel computing, free encyclopedia – Electronic and graphic data – access http://en.wikipedia.org/wiki/Parallel_computing/ open access
2. Baptiste Wicht [Web – resource], @Blog("Baptiste Wicht") Website about technologies Java, Spring, OSGi, Hardware – access http://www.baptiste-wicht.com/2012/03/cpp11-concurrency-part1-start-threads/ open access
3. C++11 threads, locks and condition variables [Web – resource], by Marius Bancila, Code Project, 27 May 2013 http://www.codeproject.com/Articles/598695/Cplusplus-threads-locks-and-condition-variables/ open access