High Level Parallel Programming

Multi-Threading in C++

Alessandro Savino
Dipartimento di Automatica e Informatica
Politecnico di Torino

Introduction

- C++ can run multiple threads simultaneously
  - Multiple threads use the same memory
    - They may read from the same memory location
    - All other accesses (i.e. read-write, write-read, write-write) are called conflicts
  - Conflicting operations are only allowed when threads are synchronized
  - This can be done with mutexes or atomic operations
  - Unsynchronized accesses (also called data races), deadlocks, and other potential issues when using threads are undefined behavior

Introduction

- The header <thread> defines the class std::thread that can be used to start new threads
  - Using this class is the best way to use threads platform-independently
  - Using it may require additional compiler flags
    - -pthread for gcc and clang

```
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++14 -pthread")
```

Example

```
void foo(int a, int b);

std::thread t1(foo, 123, 456);

std::thread t2([] { foo(123, 456); });

std::thread t3;
```

Starts a thread that calls foo(123, 456)

Also works with lambdas

Creates an object that does not refer to a thread

Join Threads

- The member function **join** can be used to wait for a thread to finish
  - Function **join** must be called exactly once for each thread
  - When the destructor of a **std::thread** is called, the program is terminated if it has an associated thread that was not joined

Example

```
std::vector<std::thread> threadPool;
for (int i = 1; i <= 9; ++i) {
  threadPool.emplace_back([i] { safe_print(i); });
}

for (auto& t : threadPool) {
  t.join();
}
```

Digits 1 to 9 are printed (unordered)

Moving Threads

- **std::threads** are not copyable, but movable, so they can be used in containers
  - Moving an **std::thread** transfers all resources associated with the running thread
  - Only the moved-to thread can be joined

Example

```
std::thread t1([] { std::cout << "Hi\n"; });
std::thread t2 = std::move(t1);
t2.join();
```

t1 is now empty

The thread originally started in t1 is joined

Other useful functions

- The thread library also contains other useful functions that are closely related to starting and stopping threads
  - std::this_thread::sleep_for
    - Stop the current thread for a given amount of time
  - std::this_thread::sleep_until
    - Stop the current thread until a given point in time
  - std::this_thread::yield
    - Let the operating system schedule another thread
  - std::this_thread::get_id
    - Get the (operating-system-specific) id of the current thread

Exercise

- Write a simple program that evaluates all prime numbers in a given range
  - Range is passed to the program as a parameter
  - Set the number of parallel threads using
    - std::thread::hardware_concurrency
      - It reports the actual max number of threads based on your architecture: get along and find out performances

Mutual Exclusion

- When working with threads, mutual exclusion is a central concept to synchronize threads
- The standard library defines several useful classes for this in <mutex> and <shared_mutex>
  - std::mutex
    - Mutual exclusion
  - std::recursive_mutex
    - Recursive mutual exclusion
  - std::shared_mutex
    - Mutual exclusion with shared locks

Mutual Exclusion

- std::unique_lock
  - RAII wrapper for std::mutex
- std::shared_lock
  - RAII wrapper for std::shared_mutex

- Note
  - Mutexes are usually inefficient as they are used very coarse-grained and sometimes require communication with the operating system

Mutexes

- A mutex is the most basic synchronization primitive which can be locked and unlocked by exactly one thread at a time
  - **std::mutex** has the member functions
    - **lock** and **unlock** that lock and unlock the mutex
    - **try_lock** that tries to lock the mutex and returns true if it was successful
  - All three functions may be called simultaneously by different threads
  - For each call to **lock** the same thread must call **unlock** exactly once

Example

```
std::mutex printMutex;

void safe_print(int i) {
  printMutex.lock();
  std::cout << i;
  printMutex.unlock();
}

int main() {
  thread t1(safe_print), t2(safe_print);

  t1.join();
  t2.join();
}
```

Recursive Mutexes

- Recursive mutexes are regular mutexes that additionally allow a thread that currently holds the mutex to lock it again
  - Implemented in the class **std::recursive_mutex**
  - Has the same member functions as **std::mutex**
  - **unlock** must still be called once for each **lock**
  - Useful for functions that call each other and use the same mutex

Example

```
std::recursive_mutex m;

void foo() {
  m.lock();
  std::cout << "foo\n";
  m.unlock();
}

void bar() {
  m.lock();
  std::cout << "bar\n";
  foo(); // This will not deadlock
  m.unlock();
}
```

Shared Mutexes

Since C++17 !

- A shared mutex is a mutex that differentiates between shared and exclusive locks
  - Implemented in the class **std::shared_mutex**
  - A shared mutex can either be locked exclusively by one thread or have multiple shared locks
  - The member functions **lock** and **unlock** are exclusive
  - The member functions **lock_shared** and **unlock_shared** are shared

Shared Mutexes

- The member functions **try_lock** and **try_lock_shared** try to get an exclusive or shared lock and return true on success
- Shared mutexes are mostly used to implement read/write-locks
  - Readers use shared locks, writers use exclusive locks

Example

```
int value = 0;
std::shared_mutex m;

std::vector<std::thread> threadPool;

// Add readers
for (int i = 0; i < 5; ++i)
  threadPool.emplace_back([&] {
    m.lock_shared();
    safe_print(value);
    m.unlock_shared();
  })

// Add writers
for (int i = 0; i < 5; ++i)
  threadPool.emplace_back([&] {
    m.lock(); ++value; m.unlock();
  })
```

Join and other coordination mechanisms still needed

Try this with POSIX mutex !

Usage Rules

- Mutexes have several requirements on how they must be used
  - For each call to **lock**, **unlock** must be called exactly once
  - **unlock** must only be called by the thread that called **lock**
  - The above also holds for **unlock_shared** and **lock_shared**
  - A thread A should not wait for a mutex from thread B to be unlocked if B needs to lock a mutex that A is currently holding (i.e. avoid deadlocks)

Usage Rules

- Note the following when using mutexes to make data structures thread-safe
  - The member functions **lock** and **unlock** are non-const
  - If const member functions of the data structure should also use the mutex, it should be mutable
  - If a member function that locks the mutex calls other member functions, this can lead to deadlocks
  - **recursive_mutex** can be used to avoid this

Mutex RAII Wrappers

- Mutexes can be thought of resources that must be acquired and freed with lock and unlock
  - The RAII pattern should be used
  - std::unique_lock is an RAII wrapper for Mutexes that calls **lock** in its constructor and **unlock** in its destructor
  - **std::unique_lock** is movable to "transfer ownership" of the locked mutex
  - It also has the member functions **lock** and **unlock** to manually control the mutex

Example

```
std::mutex m;

int i = 0;

std::thread t([&] {
      std::unique_lock l{m};
   ++i;
   ...
});
```

m.lock() is called

m.unlock() is called

Mutex RAII Wrappers

- Shared mutexes additionally need an RAII wrapper that calls lock_shared and unlock_shared
  - For this **std::shared_lock** can be used
  - Note
    - **std::shared_lock** is only movable and not copyable (unlike std::shared_ptr)

Example

```
std::shared_mutex m;

int i = 0;

std::thread t([&] {
    std::shared_lock l{m};
    std::cout << i;
    ...
});
```

m.lock_shared() is called

m.unlock_shared() is called

Avoiding Deadlocks using C++

- Deadlocks can occur when using multiple mutexes
  - When two different threads each succeed to lock a subset of the mutexes and then try to lock the rest
  - Can be avoided by always locking mutexes in a consistent order

Example

Concurrent calls to threadA() and threadB() can lead to deadlocks. E.g., A could get the locks for m1 and m2 while B gets a lock for m3.

```
std::mutex m1, m2, m3;

void threadA() {
   std::unique_lock l1{m1}, l2{m2}, l3{m3};
}

void threadB() {
   std::unique_lock l3{m3}, l2{m2}, l1{m1};
   // DANGER: order not consistent with threadA()
}
```

Avoiding Deadlocks using C++

- Sometimes, it is not possible to always guarantee a consistent order
  - The function **std::lock** takes any number of mutexes and locks them all by using a deadlock-avoiding algorithm
  - **std::scoped_lock** is an RAII wrapper for **std::lock**

Example

Here, calling threadA() and threadB() concurrently will not lead to deadlocks.
Note: This should only be used if there is no other way as it is generally very inefficient!

```
std::mutex m1, m2, m3;

void threadA() {
   std::scoped_lock l{m1, m2, m3};
}

void threadB() {
   std::scoped_lock l{m3, m2, m1};
}
```

Condition Variables

- A condition variable is a synchronization primitive that allows multiple threads to wait until an (arbitrary) condition becomes true
  - A condition variable uses a mutex to synchronize threads
  - Threads can wait on or notify the condition variable
  - When a thread waits on the condition variable, it blocks until another thread notifies it
  - If a thread waited on the condition variable and is notified, it holds the mutex
  - A notified thread must check the condition explicitly because spurious wake-ups can occur

Condition Variables

- The standard library defines the class std::condition_variable in the header <condition_variable> which has the following member functions
  - wait()
    - Takes a reference to a std::unique_lock that must be locked by the caller as an argument, unlocks the mutex and waits for the condition variable
  - notify_one()
    - Notify a single waiting thread, mutex does not need to be held by the caller
  - notify_all()
    - Notify all waiting threads, mutex does not need to be held by the caller

Example

```
using namespace std;

mutex m;
condition_variable cv;
queue<int> taskQueue;

void pushWork(int task) {
   unique_lock l{m};
   taskQueue.push(task);

   cv.notify_one();
}

void workerThread() {
   unique_lock l{m};

   while(true) {
        if (!taskQueue.empty())
        {
            int task =  taskQueue.front();
            taskQueue.pop();
            l.unlock();
            // do something
            l.lock();
        }
        cv.wait(l);
   }
}
```

Tasks are inserted into a queue and then worker threads are notified to do the task

Atomic Operations

- Threads can also be synchronized with atomic operations that are usually much more efficient than mutexes
  - Atomicity means that an operation is executed as one unit, no other operation on the same object can be executed at the same time
  - This is usually implemented by using special CPU instructions, no operating system is needed
    - Note
      - Only the atomicity of single operations are guaranteed to be atomic

Example

If we could guarantee that atomic_add is performed in an atomic way, then a = 13 will be the result nevertheless the scheduling of threadA and threadB

```
int a = 10;

void threadA() { atomic_add(a, 1); }
void threadB() { atomic_add(a, 2); }
```

Atomic Operations Library

- All classes and functions related to atomic operations can be found in the <atomic> header
- std::atomic<T> is a class that represents an atomic version of the type T
- This class has several member functions that implement atomic operations
  - T load()
    - Loads the value (allows concurrent writes)
  - void store(T desired)
    - Stores desired in the object

Atomic Operations Library

- T exchange(T desired)
  - Stores desired in the object and returns the old value
- bool compare_exchange_weak(...) and bool compare_exchange_strong(...)
  - Perform a compare-and-swap (CAS) operation

Atomic Operations Library

- If T is an integral type, the following operations also exist
  - T fetch_add(T arg)
    - Adds arg to the value and returns the old value
  - T fetch_sub(T arg)
    - Same for subtraction
  - T fetch_and(T arg)
    - Same for bitwise and
  - T fetch_or(T arg)
    - Same for bitwise or
  - T fetch_xor(T arg)
    - Same for bitwise xor

Atomic Operations: Modification Order

- All modifications of a single atomic object are totally ordered in the so-called modification order
  - The modification order is consistent between threads (i.e. all threads see the same order)
  - The modification order is only total for individual atomic objects

Example

```
std::atomic<int> i = 0;

void workerThread() {
  i.fetch_add(1); // (A)
  i.fetch_sub(1); // (B)
}

void readerThread() {
  int iLocal = i.load();
  if(iLocal == 0 || iLocal == 1)
    std::cout << "Memory Consistent!";
}
```

Because the memory order is consistent between threads, the reader thread will never see an execution order of (A), (B), (B), (A), for example.

This is always true