Concurrent Programming in C++11

Mark Slater (based on slides by Ben Morgan)

THE UNIVERSITY OF WARWICK

UNIVERSITY OF BIRMINGHAM

Concurrency

- Can no longer rely on processor clock speed for increasing computational throughput - instead, try to split tasks across N>1 parallel “things”
- There are several levels of parallelism
  - SIMD or “vectorization” (on chip)
  - **Multithread/Multicore (single machine)**
  - Multiprocessor (multiple machine)

Concurrency in Action (1)

- Concurrency is a fundamental part of modern computing
- Modern OSs use it extensively to allow users (and itself) to perform multiple tasks at the same time
- Having several windows open on a desktop is a very obvious form of this concurrency

Concurrency in Action (2)

- Individual programs can also take advantage of the concurrency offered by the underlying OS, e.g. Web browsers:
    - You download a file - this happens in a separate thread.
    - Means you can continue browsing while the file downloads in the background.
    - The browser may download updates for itself in the background (Chrome for example)
    - Multiple tabs can have web scripts/services running and updating at the same time

Modern PCs and Threading

- You aren’t restricted to having the number of threads = number of cores
- The operating system will take care of scheduling the waiting tasks across the CPU cores available
- The majority of threads don’t use CPU most of the time – they are waiting for input, disk access, network, etc.
- During these ‘sleep’ times, the OS can give CPU time to other threads to continue their tasks

Concurrency in C++11

- Prior to C++11, concurrent programming relied on the underlying OS implementation (pthreads on UNIX, CreateThread on Windows)

- C++11 introduced the thread support library which provides a cross-platform API hiding the underlying implementation.

- Provides all of the main abstractions of multithreading in a series of headers:

    - http://en.cppreference.com/w/cpp/thread

`std::async` and `std::future`

- C++11 provides both high and low level thread creation/management interfaces (cf new/delete vs make_shared/make_unique for memory)

- We'll only look at the high level interface:

  - `std::async` : Takes a function that will be run asynchronously, returns a `std::future` instance that will hold the result of the function call.

  - `std::future` : Wraps result of an asynchronous operation. Provides interface to query, wait for or get result of the operation.

Basic Threading Example

```
int main(int, char **) {

  auto fn = [] () {std::cout << ”[thread] Wait for it…\n”;
    std::this_thread::sleep_for(std::chrono::seconds(10));
    std::cout << “[thread] Done!\n”;
    return 8;
  };

  // Start up a thread
  auto future1 = std::async(std::launch::async, fn);

  // wait a bit
  std::this_thread::sleep_for(std::chrono::seconds(2));

  // start another one
  auto future2 = std::async(fn);

  // wait for the second to finish
  std::future_status status{std::future_status::ready};
  do {
    status = future2.wait_for(std::chrono::seconds(1));
    if (status == std::future_status::timeout) {
      std::cout << "[main] waiting...\n";
    } else if (status == std::future_status::ready) {
      std::cout << "[main] finally, an answer!\n";
    }
  } while (status != std::future_status::ready);

  std::cout << ”Answer is: " << future2.get() <<"\n";
}
```

It is useful to use the ‘chrono’ header and functions for specifying timeouts

The sleep_for function is very useful if you know a thread won’t need to do anything for a while.

Create and start a new thread that will run the given function to completion. This returns a ‘future’ that can be queried for the status and result

Use ‘wait_for’ on a future to wait for it to complete or the timeout occurs

‘get’ will return the result of the function run in the thread when available

Making MPAGS Cipher Multithreaded

- A possible use of multithreading is when processing a very large file
- The input text could be split up into ‘chunks’ which could all be processed independently using threads
- There are a few things to consider:

1. You will need to construct strings for each thread to run on
2. You only need one cipher object as the `applyCipher` function is `const` and already thread-safe
3. You will need to keep track of several threads and so will need to store the futures in a vector
4. Your main code will need to wait until all threads have returned a result and then concatenate them together

Exercise: adding threading to MPAGSCipher

- To implement the multithreading in MPAGSCipher, work through the following:

    1. You'll need to include the '`future`' header as well as the '`thread`' header

    2. You'll need to link against the threading library. This can either be done by adding '-lpthread' explicitly to the CMAKE_CXX_FLAGS or doing the following:

```
find_package( Threads )
add_executable(mpags-cipher mpags-cipher.cpp)
target_link_libraries(mpags-cipher PRIVATE MPAGSCipher ${CMAKE_THREAD_LIBS_INIT})
```

    3. Loop over the number of threads you want to use (should be configurable but don't worry about that now!)

    4. For each iteration, take the next chunk from the input string

    5. Start a new thread to run a lambda function that calls the '`applyCipher`' function on the constructed Cipher object

    6. Loop over the `futures` and wait until they are all completed

    7. Get the results from them and assemble the final string

Exercise: adding threading to MPAGSCipher – Some Notes!

- To get a substring from a string, use the `string.substr` function – just make sure you're covering the whole string!

- When creating the threads and getting the futures, you'll need to push them directly onto a vector:

```
std::vector< std::future< std::string > > futures;
futures.push_back( std::async(std::launch::async, <FN>, arg1, arg2, … ));
```

- Though you could create a separate function to apply the cipher, a lambda (using variable capture) is a lot easier. Be careful about passing any local/changing variables by reference!

- Use a range based for loop to go over the futures vector and the `wait_for` function to check the status

- After all have finished, use the `get()` method to put all the output strings together

Traps and Pitfalls

- Concurrent programming requires more thought because data (Objects) can be shared between threads
- For example, what happens if two threads try to add data into the same `std::vector` instance at the same time? Locking and Mutex’s are useful here.
- Since computations may be performed out of sequence, synchronisation may be needed.
- The good news is that designing code for concurrency generally results in cleaner and more coherent code!

Example

https://github.com/mpags-cpp/mpags-cpp-extra

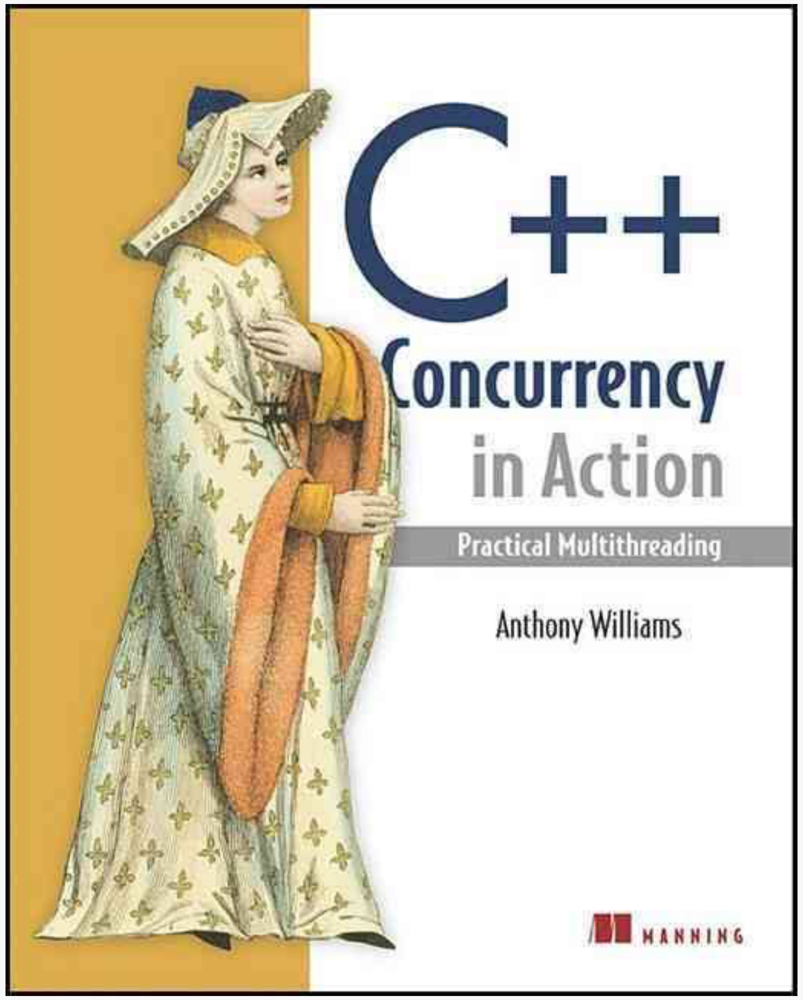
thread.cpp (~/tmp/day6) - VIM

```cpp
#include <chrono>
#include <future>
#include <iostream>


int getUltimateAnswer(const std::chrono::microseconds& waitFor) {
  std::cout << "[getUltimateAnswer] starting to think...\n";
  std::this_thread::sleep_for(waitFor/2);
  std::cout << "[getUltimateAnswer] still thinking...\n";
  std::this_thread::sleep_for(waitFor/2);
  std::cout << "[getUltimateAnswer] got the answer...\n";
  return 42;
}


int main(int, char**) {
  std::chrono::microseconds thinkFor {7500000};
  auto ua = std::async(getUltimateAnswer, thinkFor);

  std::cout << "[main] Now we wait...\n";
  std::future_status status;
  do {
    // wait 1ms, get status of future
    status = ua.wait_for(std::chrono::seconds(1));

    if (status == std::future_status::timeout) {
```

thread.cpp[cpp] [1/35][1]

Further Reading

- For C++, a good reference is Anthony Wiliams' book:



- For more general guides to structuring concurrent algorithms: