Scientific Programming with C++

MULTITHREADING

Learning objectives

- After this lecture and related assignments, you will…
  - Know how to use multithreading with C++
  - Be introduced to different classes for asynchronous operations
  - Know the pitfalls related to multithreading

Multithreading

- Several sub-programs run concurrently
- Utilizes multi-core processors
- High-performance applications
  - Helps with real-time computation
- Lots of pitfalls
  - Data races
    - Multiple threads try to access the same resource at the same time
  - Exception management
    - Has to be implemented inside the thread
- C++ classes in header <thread>

Managing multiple threads

- Launching a thread or threads
  - Using the constructor of std::thread
  - If you need to put the threads in a container, note that threads cannot be copied
    - Use std::move to put already constructed threads into a container
- Waiting for the thread(s) to finish or detaching them
  - Waiting will freeze the calling thread until the waited thread finishes
  - Detaching will relinquish control of the thread, letting it finish independently

std::thread

- Constructor: used to launch functions in a thread
  - First arguments is the name of the function
  - The following arguments are the arguments of the function
- Method join()
  - Waits for the thread to finish
- Method detach()
  - Lets the thread execute independently of the object
  - Like letting go of a balloon: it will float somewhere without our control

Notes about using threads

- When passing by reference, arguments must be wrapped with std::ref
  - Creates a sort of faux reference
- Passing arguments by reference can be unsafe
  - The object used as argument may go out of scope while the thread still runs
    - For example, when the thread is detached
    - "Dangling references"
- For repeated tasks, thread pools are used
  - Instead of creating a new thread for each task, an existing thread is reassigned to a task

Example: launching threads

```cpp
void calculateMean(const std::vector<double> vec, unsigned int startIdx, unsigned int endIdx, int idx) {
    double total = 0;
    for (unsigned int i = startIdx; i < endIdx; ++i) {
        total += vec[i];
    }
    double mean = total / (endIdx - startIdx + 1);
    std::cout << idx << ": " << mean << std::endl;
}

int main() {
    // initialize a vector
    std::vector<double> doubles;
    doubles.reserve(1e6);
    for (unsigned int i = 0; i < 1e6; ++i)
        doubles.push_back(i);
    // launch all threads and store them in a vector
    std::vector<std::thread> threadContainer;
    for (unsigned int i = 0; i < 10; ++i) {
        std::thread thr(calculateMean,doubles, i * 1e5, (i + 1) * 1e5 - 1, i);
        threadContainer.push_back(std::move(thr));
    }
    // make sure all threads finish running
    for (unsigned int i = 0; i < 10; ++i) {
        threadContainer[i].join();
    }
}
```

```
0: 49998.5
1: 149998
2: 249997
3: 349996
4: 449995
5: 549994
6: 649993
7: 749992
8: 849991
9: 949990
```

```
0: 49998.5
1: 149998
2: 249997
4: 3: 349996
449995
5: 549994
6: 649993
7: 749992
8: 849991
9: 949990
```

Managing access to resources

- Mutexes
  - Ensure that a section of code can only be executed by one thread at a time
    - Indicate beginning of the section with lock() or try_lock()
    - Indicate end of the section with unlock()
  - Cannot be owned by the calling thread
    - Undefined behaviour
- Atomic data types
  - Data types that have defined safe behaviour when multiple threads access them
  - Example: std::atomic<int> number = 2;

Example: mutex

```cpp
std::string hello = "Hello";
std::mutex mutex;

void overwriteHello(unsigned int i) {
    std::lock_guard<std::mutex> myLock(mutex); // without this, we risk a data race
    hello = "Hello from string " + std::to_string(i) + "!";
}

int main() {
    unsigned int nThreads = 1000;
    std::vector<std::thread> threadContainer;
    // launch a number of threads
    for (unsigned int i = 0; i < nThreads; ++i) {
        std::thread thr(overwriteHello, i);
        threadContainer.push_back(std::move(thr));
    }
    // make sure all threads finish running
    for (unsigned int i = 0; i < nThreads; ++i) {
        threadContainer[i].join();
    }
    // print the contents written by the thread that last accessed hello
    std::cout << hello;
}
```

Hello from string 999!

Hello from string 998!

Example: atomic

```
std::atomic<unsigned int> lastAccessor = 0;

void overwriteInt(unsigned int i) {
    lastAccessor = i; // if lastAccessor is a normal int, we risk a data race
}

int main() {
    unsigned int nThreads = 1000;
    std::vector<std::thread> threadContainer;
    // launch a number of threads
    for (unsigned int i = 0; i < nThreads; ++i) {
        std::thread thr(overwriteInt, i);
        threadContainer.push_back(std::move(thr));
    }
    // make sure all threads finish running
    for (unsigned int i = 0; i < nThreads; ++i) {
        threadContainer[i].join();
    }
    // print the index of the thread that last accessed lastAccessor
    std::cout << lastAccessor;
}
```

```
999
```

Example: using mutex (1/2)

```cpp
// mutex must not be owned by the thread that uses it, otherwise we have undefined behaviour; therefore it shall be a global variable
std::mutex mutex;

void findOptimum(unsigned int maxIterations, std::vector<double> independentVariable, std::vector<double> y, std::array<double, 2>& params) {
    // initialize random number generator seed
    std::random_device rd;
    // initialize mersenne twister pseudorandom generation algorithm
    std::mt19937 gen(rd());
    // construct the distribution we use to generate random numbers
    std::uniform_real_distribution<> dis(-10.0, 10.0);
    // initialize local (this function only) parameter guesses a and b
    double aLocal = 0;
    double bLocal = 0;
    // initialize lowestError to a high value
    double lowestError = 1e9;
    // initialize best parameter guesses to initial values for now
    double aBest = aLocal;
    double bBest = bLocal;

    // loop through all desired iterations
    for (unsigned int iter = 0; iter < maxIterations; ++iter) {
        // randomly generate new parameters
        aLocal = dis(gen);
        bLocal = dis(gen);
        // calculate error with these new parameters
        double error = calculateError(aLocal, bLocal, independentVariable, y);
        // if error in this iteration is lower than the lowest error so far, update the lowest error and the best parameters
        if (error < lowestError) {
            lowestError = error;
            aBest = aLocal;
            bBest = bLocal;
        }
    }

    // wrap a mutex with lock_guard that will keep the block locked until the lock_guard goes out of scope
    std::lock_guard<std::mutex> lock(mutex); // without this we risk having a data race as several threads could write to params simultaneously
    std::cout << "a=" << aBest << ", b=" << bBest << ", error=" << lowestError << std::endl;
    // check against the previous lowest error from parameters possibly calculated by other threads so far, and update parameters and error if error is lower
    double previousLowestError = calculateError(params[0], params[1], independentVariable, y);
    if (lowestError < previousLowestError) {
        params[0] = aBest;
        params[1] = bBest;
    }
}
```

Example: using mutex (2/2)

```cpp
double calculateError(double a, double b, const std::vector<double>& x, const std::vector<double>& y) {
    std::vector<double> results;
    for (unsigned int i = 0; i < y.size(); ++i)
        results.push_back(a * x[i] + b * pow(x[i], 2));

    double error = 0;
    for (unsigned int i = 0; i < y.size(); ++i)
        error += pow(results[i] - y[i], 2);
    return error;
}
```

```
a=-5.0303, b=0.212523, error=0.318527
a=-5.04782, b=0.218923, error=0.330053
a=-5.02069, b=0.213488, error=0.331503
a=-5.05341, b=0.216722, error=0.329
Final a=-5.0303, b=0.212523
```

```cpp
// Problem: we have a bunch of data (x and measurements) and we want to fit the function y = ax + bx^2 to that data
// We do not know a and b, so we must find some a and b that minimize the error (a*x + b*x^2 - measurements)^2
// our measurements are y + a normal error with a standard deviation of 0.1
// real a is -5 and real b is 0.2, which we used to generate the values for this example; our final estimated a and b should be close to them
std::vector<double> measurements = { 0.0538, -2.2666, -5.0259, -6.9638, -9.1681, -11.3808, -13.2434, -15.0157, -16.4422, -18.1731, -20.1350, -21.1465 };
std::vector<double> x = {0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5 };

// start with initial parameter guesses a=0 and b=0
std::array<double, 2> params = {0, 0};

// to speed things up, we use 4 threads that concurrently try to find best parameters a and b
std::thread thr1(findOptimum, 1e6, x, measurements, std::ref(params));
std::thread thr2(findOptimum, 1e6, x, measurements, std::ref(params));
std::thread thr3(findOptimum, 1e6, x, measurements, std::ref(params));
std::thread thr4(findOptimum, 1e6, x, measurements, std::ref(params));

// make sure all threads finish
thr1.join(); thr2.join(); thr3.join(); thr4.join();

// print the best estimated a and b
std::cout << "Final a=" << params[0] << ", b=" << params[1];
```

Moving data between threads

- Buffers
  - Data that is accessed by two or more concurrent threads
    - For example: containers, single variables
  - Must be written so that several threads cannot access the same resource at the same time
    - Data race
    - Useful tools: mutexes, atomic data types, condition variables (next slide)

std::condition_variable

- Used to block a thread until another thread sends a signal to continue
  - The notify function signals one or more other threads that are waiting
  - The wait function blocks the thread until the thread is notified

Example: condition_variable

```cpp
std::mutex mutex;
std::condition_variable cv;
int dataValue = 0;

void measurementDevice() {
    // wait until the main function calls cv.notify_one()
    std::unique_lock<std::mutex> lock(mutex);
    cv.wait(lock);
    // write 1 to dataValue
    dataValue = 1;
    lock.unlock();
    // inform the main function that it can print the value
    cv.notify_one();
}

int main() {
    // launch a thread and detach it to operate independently
    std::thread thr(measurementDevice);
    thr.detach();
    // query user for input
    std::string a;
    std::cout << "Write something and press enter" << std::endl;
    std::cin >> a;
    // inform thr that it can stop waiting
    cv.notify_one();
    // wait until thr calls cv.notify_one()
    std::unique_lock<std::mutex> lock(mutex);
    cv.wait(lock);
    // print the dataValue that thr wrote
    std::cout << dataValue;
}
```

```
Write something and press enter
something
1
```

Beyond std::thread

- Tasks can be run independently of one another without using std::thread directly
- std::async
- std::future
- std::promise

std::async

- Runs a task asynchronously
- Launched similarly as std::thread
- Two launch policies: async and deferred
  - Async: task is executed on a separate thread now
  - Deferred: task is executed on the calling thread when the results are asked by the program
    - This option can be viewed as calling a function later than where the code line to call it appears
- Can return values (sort of)
  - Data type std::future

std::future

- A handle to a return type that may yet not be available
- Represents a value that will become available in the future
  - After a thread is finished
  - After some event occurs, without having anything to do with threads
  - Etc
- Fetched from an asynchronous operation using std::promise

std::promise

- Communicates the future to the calling thread from the separately launched thread
- The separately launched thread makes a promise to the calling thread
    - The promise is fulfilled as the future on the calling thread

Asynchronous programming vs multithreading

- Asynchronous programming uses tasks that don't block the program while they run
  - Normally the program waits for the current task to finish before starting another
  - With std::async, you can start another task before waiting for the previous task to finish
- Multithreading can be seen as a part of asynchronous programming
  - Often multithreading tasks have more interaction between them
    - Shared data (buffers)
    - Triggers to let the other threads continue (condition variables)
- std::async can be a convenient alternative to std::thread
  - Less hassle, but also less customization

Summary

- Several classes exist for asynchronous operations
- Threads can increase the performance of your program significantly
  - If used incorrectly, will increase errors as well
- Make sure that if several concurrent threads access the same resource, that behaviour is defined and won’t cause exceptions