CS 4414: Recitation 10

Sagar Jha
Today: Multithreading and Functional Programming in C++

Multithreading Part III (BS Chapter 15)
• std::condition_variable
• Asynchronous computation in C++: std::future<>, std::promise<>, std::packaged_task, std::async
• Code walkthrough of third-party thread pooling libraries

Lazy Evaluation in C++ (From the book Functional Programming in C++)
• Implementation of lazy_val
• Lazy evaluation as an optimization technique
• Generalized memoization
• Expression templates
Synchronization in C++: std::condition_variable

- Pattern: A thread waits for a condition to be true. Another thread updates the condition and notifies the first thread.

- Updating thread
  - acquire a std::mutex (using a std::scoped_lock)
  - perform the update
  - call notify_one or notify_all ()

- Waiting thread
  - acquire the same std::mutex (using a std::unique_lock)
  - call wait, wait_for or wait_until supplying the condition as a predicate
std::condition_variable: wait and notify

```cpp
template< class Predicate >
void wait( std::unique_lock< std::mutex >& lock, Predicate pred );
```

- atomically unlocks lock
- reacquires it after waking up
- continues if the condition is true
- goes back to wait (unlocking again) if the condition is false

```cpp
void notify_one() noexcept;
```

- unblocks one of the waiting threads
- notify_all unblocks all of the waiting threads
Notes about std::condition_variable

• Beware of spurious wake-ups! Always check the condition in wait.
• Why use a std::unique_lock<std::mutex> in wait? Because std::scoped_lock<std::mutex> does not offer the lock and unlock operations required in wait.
• std::condition_variable only works with std::mutex. Use std::condition_variable_any to work with other mutex types, for e.g. std::shared_mutex
• A writer can notify all readers waiting using std::condition_variable_any::notify_all. All readers can now work simultaneously holding the std::shared_mutex.
Thinking in terms of tasks

• In many cases, we don’t need to think at the lower level of threads and locks
• Task – work that needs to be done, potentially concurrently
• C++ provides std::future and std::promise, std::packaged_task and std::async
• Defined in #include<future>
Communicating tasks: std::future<> and std::promise<>

• Enable transfer of value between threads (or tasks) without explicit use of a lock
• std::promise<> is used by the producer task to supply the value
• std::future<> is used by the thread that needs the value
Using `std::future<T>`

```cpp
T get();
• Blocks the thread until the result is available
• If the producer thread sets an exception, throws that same exception
```

template< class Rep, class Period >
std::future_status wait_for( const std::chrono::duration<Rep,Period>& timeout_duration ) const;

• Waits for a value until the provided timeout
• If you only want to check if the result is available without waiting, pass a duration of 0
• returns one of `future_status::deferred`, `future_status::ready` or `future_status::timeout`. If ready, call `get` to obtain the value
Using `std::promise<R>`

```cpp
std::future<R> get_future();
• Returns the associated future object
• Can only call this once

void set_value( const R& value );
• Atomically stores the value into the shared state
• Now get on the associated future will unblock

void set_exception( std::exception_ptr p );
• Indicate that there won’t be any value, but an exception instead
Communicating tasks:
std::packaged_task<R(Args...)>  
• Solves the problem of managing futures and promises

Important functions
• template <class F> explicit packaged_task( F&& f );
• std::future<R> get_future();
• void operator()( ArgTypes... args );
double accum(double* beg, double* end, double init)
    // compute the sum of [beg:end) starting with the initial value init
{
    return accumulate(beg,end,init);
}

double comp2(vector<double>& v)
{
    using Task_type = double(double*,double*,double);    // type of task
    packaged_task<Task_type> pt0 {accum};                 // package the task (i.e., accum)
    packaged_task<Task_type> pt1 {accum};

    future<double> f0 {pt0.get_future()};                  // get hold of pt0's future
    future<double> f1 {pt1.get_future()};                  // get hold of pt1's future

    double* first = &v[0];
    thread t1 {move(pt0),first,first+v.size()/2,0};        // start a thread for pt0
    thread t2 {move(pt1),first+v.size()/2,first+v.size(),0}; // start a thread for pt1

    // ...

    return f0.get()+f1.get();                              // get the results
}
Communicating tasks: std::async

- Async is used to specify tasks that run asynchronously, potentially in other threads
- No need to even think about threads, C++ manages them possibly as part of a thread pool
- There is no synchronization between the async tasks. Don’t use it if you need synchronization!
- For specialized parallel executions, C++’s algorithm library offers execution policies such as std::execution::seq, std::execution::par, std::execution::par_unseq, and std::execution::unseq
double comp4(vector<double>& v) {
    // spawn many tasks if v is large enough
    if (v.size() < 10000) {  // is it worth using concurrency?
        return accum(v.begin(), v.end(), 0.0);
    }

    auto v0 = &v[0];
    auto sz = v.size();

    auto f0 = async(accum, v0, v0 + sz/4, 0.0);  // first quarter
    auto f1 = async(accum, v0 + sz/4, v0 + sz/2, 0.0);  // second quarter
    auto f2 = async(accum, v0 + sz/2, v0 + sz*3/4, 0.0);  // third quarter
    auto f3 = async(accum, v0 + sz*3/4, v0 + sz, 0.0);  // fourth quarter

    return f0.get() + f1.get() + f2.get() + f3.get();  // collect and combine the results
}
Multithreading in action: Implementing a thread pool

• C++ does not offer a native thread pool library

• We will review the implementation of two third-party libraries:
  • ThreadPool: https://github.com/progschj/ThreadPool
  • C++ Thread Pool Library: https://github.com/vit-vit/CTPL (referred only for function resize)
ThreadPool: Public functions

• ThreadPool::ThreadPool(size_t threads);
  • Create a thread pool with \textit{threads} number of threads

• template<class F, class... Args>
  auto ThreadPool::enqueue(F&& f, Args&&... args)
    -> std::future<typename std::result_of<F(Args...)>::type>
  • Add a new task to the pool

• ThreadPool::~ThreadPool()
  • Non-trivial destructor since we are working with threads
ThreadPool: Data members

- `std::vector< std::thread > workers;`
  - collection of threads in the pool

- `std::queue< std::function<void()> > tasks;`
  - collection of tasks that need to be completed

- `std::mutex queue_mutex;`
  - `std::condition_variable condition;`
  - `bool stop;`
    - For synchronization
// the constructor just launches some amount of workers
inline ThreadPool::ThreadPool(size_t threads)
       : stop(false)
{
   for(size_t i = 0;i<threads;++i)
       workers.emplace_back(
           [this]
           {
               for(;;)
               {
                   std::function<void()> task;
                   {
                       std::unique_lock<std::mutex> lock(this->queue_mutex);
                       this->condition.wait(lock,
                           [this]{ return this->stop || !this->tasks.empty(); });
                       if(this->stop && this->tasks.empty())
                           return;
                       task = std::move(this->tasks.front());
                       this->tasks.pop();
                   }
                   task();
               }
           });
}
// add new work item to the pool
template<class F, class... Args>
auto ThreadPool::enqueue(F&& f, Args&&... args)
   -> std::future<typename std::result_of<F(Args...)>::type>
{
    using return_type = typename std::result_of<F(Args...)>::type;

    auto task = std::make_shared<
        std::packaged_task<return_type>()>
    >(std::bind(std::forward<F>(f), std::forward<Args>(args)...));

    std::future<return_type> res = task->get_future();
{
    std::unique_lock<std::mutex> lock(queue_mutex);

    // don't allow enqueueing after stopping the pool
    if(stop)
      throw std::runtime_error("enqueue on stopped ThreadPool");

    tasks.emplace([task] { (*task); });
    condition.notify_one();

    return res;
}
// the destructor joins all threads
inline ThreadPool::~ThreadPool()
{
    std::unique_lock<std::mutex> lock(queue_mutex);
    stop = true;
}
condition.notify_all();
for(std::thread &worker: workers)
    worker.join();
void resize(int nThreads) {
    if (!this->isStop && !this->isDone) {
        int oldNThreads = static_cast<int>(this->threads.size());
        if (oldNThreads != nThreads) { // if the number of threads is increased
            this->threads.resize(nThreads);
            this->flags.resize(nThreads);
            for (int i = oldNThreads; i < nThreads; ++i) {
                this->flags[i] = std::make_shared<std::atomic<bool>>(false);
                this->set_thread(i);
            }
        } else { // the number of threads is decreased
            for (int i = oldNThreads - 1; i >= nThreads; --i) {
                *(this->flags[i]) = true; // this thread will finish
                this->threads[i]->detach();
            }
            // stop the detached threads that were waiting
            std::unique_lock<std::mutex> lock(this->mutex);
            this->cv.notify_all();
            this->threads.resize(nThreads); // safe to delete because the threads are detached
            this->flags.resize(nThreads); // safe to delete because the threads have copies of shared_ptr of the flags, not originals
        }
    }
}
Part II: Lazy Evaluation in C++

- Chapter 6 of *Functional Programming in C++* by Ivan Čukić
- C++ does not provide lazy evaluation like Haskell does
  - auto P = A * B; for matrices A and B will be evaluated immediately
- But we can use C++’s functional programming features for lazy eval.
- For example, one can define
  ```cpp
  auto P = [A, B] { return A * B; };
  ```
- Now, P can be called when the value is needed
- What if the value is needed multiple times?
Laziness in C++

- Define a class lazy_val with the following data members

```cpp
template <typename F>
class lazy_val {
private:
    F m_computation;
    mutable bool m_cache_initialized;
    mutable decltype(m_computation()) m_cache;
    mutable std::mutex m_cache_mutex;

public:
    ...
};
```

- Declaring cache-related members as mutable means that the member functions can be declared const
Implementation of implicit cast of lazy_val

```cpp
operator const decltype(m_computation())& () const
{
    std::unique_lock<std::mutex> lock{m_cache_mutex};
    if (!m_cache_initialized) {
        m_cache = m_computation();
        m_cache_initialized = true;
    }

    return m_cache;
}
```
We don’t even need the mutex with std::call_once!

template <typename F> class lazy_val {
private:
    F m_computation;
    mutable decltype(m_computation()) m_cache;
    mutable std::once_flag m_value_flag;

public:
    ...
    operator const decltype(m_computation())& () const {
        std::call_once(m_value_flag, [this] {
            m_cache = m_computation();
        });
        return m_cache;
    }
};
Laziness as an optimization technique

- Sorting collections lazily
  - What if you only need the top k elements?
  - For example, displaying results page by page for a web query
  - Lazy quicksort: Don’t sort the partitions that are not part of the result
Laziness as an optimization technique

- Pruning recursion trees by caching function results
  - Fibonacci is a classic example
  - Also applicable to dynamic programming through memoization

```cpp
std::vector<unsigned int> cache{0, 1};

unsigned int fib(unsigned int n) {
    if (cache.size() > n) {
        return cache[n];
    } else {
        const auto result = fib(n - 1) + fib(n - 2);
        cache.push_back(result);
        return result;
    }
}
```
Generalized memoization

• Question: Can we write a generalized function wrapper that can provide caching? We don’t need to be smart about what to cache.

template <typename Result, typename... Args>
auto make_memoized(Result (*f)(Args...)) {
  std::map<std::tuple<Args...>, Result> cache;

  return [f, cache](Args... args) mutable -> Result {
    const auto args_tuple = std::make_tuple(args...);
    const auto cached = cache.find(args_tuple);

    if (cached == cache.end()) {
      auto result = f(args...);
      cache[args_tuple] = result;
      return result;
    } else {
      return cached->second; }
  };
}
What about recursive functions?

• Refer to the book for an implementation
• Makes the following possible with automatic memoization:

```cpp
auto fibmemo = make_memoized_r<
    unsigned int(unsigned int)>
    [](auto& fib, unsigned int n) {
        std::cout << "Calculating " << n << "!\n";
        return n == 0 ? 0
            : n == 1 ? 1
            : fib(n - 1) + fib(n - 2);
    };
```
Expression templates and lazy string concat.

• Consider the following expressions:

  ```cpp
  std::string fullname = title + " " + surname + ", " + name;
  ```

• + is a left-associative binary operator, so it’s evaluated as

  ```cpp
  std::string fullname = (((title + " ") + surname) + ", " + name;
  ```

• This generates and destroys strings that are not needed. This is not efficient.
Solution: Define a class that can hold multiple strings together using variadic templates

template <typename... Strings> class lazy_string_concat_helper;
template <typename LastString, typename... Strings>
class lazy_string_concat_helper<LastString, Strings...> { 

private:
    LastString data;
    lazy_string_concat_helper<Strings...> tail;

public:
    ...
}

template <> class lazy_string_concat_helper<> { ...}
Definition of operator +

```cpp
lazy_string_concat_helper<
    std::string,
    LastString,
    Strings...>
operator+(const std::string& other) const {
    return lazy_string_concat_helper
        <
            std::string,
            LastString,
            Strings...
        >(other,
            *this);
}
```
Final remarks about template expressions

• We can hold the operands (strings in case of string concatenation) by references to avoid copying
• But we need to make sure that we only access the expression as long as the strings are in scope