Parallel Programming course. C++ threading

Obolenskiy Arseniy, Nesterov Alexander

Nizhny Novgorod State University

April 29, 2025

Contents

Introduction to C++ threading API

What is `std::thread`?

- Part of the C++11 thread support library (`<thread>`, `<mutex>`, etc.)
- Low-level, manual thread creation and management
- Relies on the OS native threads under the hood
- Provides:
  - `std::thread` for launching threads
  - Synchronization primitives (`std::mutex`, `std::condition_variable`, `std::atomic`)
  - Utilities for futures and promises (`std::future`, `std::promise`)

Threading before C++11

- POSIX threads (`pthread`): C-based API, widely used on Unix-like systems
- Windows threads: Win32 API with `CreateThread`, `CRITICAL_SECTION`, and events
- External libraries (e.g. Boost.Thread)

Challenges and inconveniences:

- non-standard APIs (towards C++)
- platform dependent APIs
- verbose initialization boilerplate
- manual resource management

std::thread history

- C++11 (2011): Introduced `std::thread`, `std::mutex`, `std::future`, etc.
- C++14/17/20: Incremental improvements (e.g., `std::hardware_constructive_interference_size`)
- Widely adopted as the base for custom thread pools and concurrency utilities
- Now the foundation of many higher-level C++ concurrency libraries

OpenMP vs TBB vs `std::thread`

- **OpenMP**
  - Directive-based, compiler-driven shared-memory parallelism
  - Simple loop and region parallelism
- **TBB**
  - Template library, task-based parallelism with work-stealing
  - High-level parallel patterns (`tbb::parallel_for`, `tbb::parallel_reduce`, etc.)
- **std::thread**
  - Low-level manual thread API
  - Full control over thread lifetime, but more boilerplate
  - Foundation for building custom task systems or thread pools

Pros and cons of `std::thread`

Pros

- Complete control over thread creation and destruction
- No hidden scheduler—behavior is predictable
- Part of standard C++, portable across platforms
- Good for learning fundamentals

Cons

- Manual management—risk of leaks, detach/join errors
- Verbose boilerplate for synchronization
- No built-in task scheduling or work-stealing
- Harder to scale and tune compared to higher-level APIs

C++ STL threading API (std::thread, std::jthread)

std::thread details

- Constructors:
  - `std::thread(callable, args...)`: starts a new thread executing `callable` with `args`
  - Default constructor: creates a thread object without an associated thread
- Member functions:
  - `join()`: blocks until the thread finishes execution
  - `detach()`: detaches the thread to run independently
  - `joinable()`: checks whether the thread can be joined
  - `get_id()`: returns the `std::thread::id` of the thread
- Properties:
  - Move-only: supports move construction and assignment; copy operations are deleted
  - Static `hardware_concurrency()`: returns the number of concurrent threads supported

Creating and launching threads

```
#include <thread>
#include <iostream>

void worker(int id) {
  std::cout << "Worker " << id << " running\n";
}

int main() {
  // Launch two threads with argument
  std::thread t1(worker, 1);
  std::thread t2(worker, 2);

  // Wait for them to finish
  if (t1.joinable()) {
    t1.join();
  }
  if (t2.joinable()) {
    t2.join();
  }

  return 0;
}
```

- Construct `std::thread` with a callable + optional args
- Must call `join()` or `detach()` before destruction
- Threads are move-only: no copy construction/assignment

Managing thread lifetime

- `join()`: waits for thread completion
- `detach()`: allows thread to run independently (daemon-like)

```cpp
std::thread t(worker);
// ...
if (t.joinable()) {
    t.join();
}

```

- `joinable()`: check if thread can be joined
- Detached threads continue after `main`—dangerous if resources go out of scope
- Always ensure each thread is either joined or detached

Passing arguments to threads

```cpp
#include <thread>
#include <string>
#include <iostream>

void greet(const std::string &name) {
  std::cout << "Hello, " << name << "!\n";
}

int main() {
  std::string user = "Alice";
  // Pass by reference: use std::ref
  std::thread t(greet, std::ref(user));
  t.join();
  return 0;
}
```

- By default, args are copied into the new thread
- Use `std::ref()` to pass references

Querying hardware concurrency capabilities

- `std::thread::hardware_concurrency()` returns the number of supported threads
- May potentially return 0

`unsigned n = std::thread::hardware_concurrency();` Usage example: to size thread pools or partition work

std::jthread

- Introduced in C++20 as a safer and more convenient alternative to `std::thread`.
- Automatically joins upon destruction, reducing the risk of detached threads.
- Supports cooperative cancellation via `std::stop_token`.

Example usage:

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

void task(std::stop_token stoken) {
  while (!stoken.stop_requested()) {
    std::cout << "Working...\n";
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
  }
  std::cout << "Task stopping.\n";
}

int main() {
  std::jthread jt(task);
  std::this_thread::sleep_for(std::chrono::seconds(1));
  // Request stop automatically during destruction or manually via jt.request_stop
     ();
  return 0;
}

```

`std::future`, `std::promise` and `std::async`

Future/Promise and Async Execution

- Future and promise provide a mechanism for asynchronous communication between threads
- A promise allows one thread to set a value or report an error
- The associated future retrieves the value, waiting if necessary
- std::async simplifies launching asynchronous tasks without explicit thread management
- These features promote decoupling of computation and enhance concurrency control

std::future and std::promise

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

int compute() {
  int result;
  ...  // heavy computation
  return result;
}

int main() {
  std::promise<int> prom;
  std::future<int> fut = prom.get_future();

  std::thread t([&prom]{
      prom.set_value(compute());
  });

  std::cout << "Result: " << fut.get() << "\n";
  t.join();
  return 0;
}
```

std::promise sets a value (or failure)

std::future retrieves it on demand

Using std::async

- Launches tasks asynchronously and returns a `std::future`
- Can run immediately in a new thread or be deferred until needed
- Simplifies asynchronous programming by handling thread management

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

int computeSquare(int x) {
  // Simulate heavy computation
  std::this_thread::sleep_for(std::chrono::seconds(1));
  return x * x;
}

int main() {
  // Launch computeSquare asynchronously. Using std::launch::async forces immediate
     execution
  std::future<int> fut = std::async(std::launch::async, computeSquare, 5);
  std::cout << "Square: " << fut.get() << std::endl;
  return 0;
}

```

Different launch policies available in std::async

```cpp
#include ...

int computeCube(int x) { ... }

int main() {
  // Using deferred launch: execution is postponed until get() is called
  std::future<int> futDeferred = std::async(std::launch::deferred, computeCube, 3);
  std::cout << "Deferred result: " << futDeferred.get() << std::endl;

  // Using async launch: task is executed immediately in a new thread
  std::future<int> futAsync = std::async(std::launch::async, computeCube, 3);
  std::cout << "Async result: " << futAsync.get() << std::endl;

  // Using default launch policy: implementation defined behavior
  std::future<int> futDefault = std::async(computeCube, 3);
  std::cout << "Default launch result: " << futDefault.get() << std::endl;

  return 0;
}
```

Policies:

- `std::launch::deferred`: Execution is delayed until `get()` is called
- `std::launch::async`: Execution starts immediately in a separate thread
- Default policy: The decision is left to the implementation

Synchronization primitives (mutexes, condition variables, ...)

Synchronization primitives overview

- C++ provides several mechanisms to coordinate concurrent operations:
  - Mutual exclusion: `std::mutex`, `std::lock_guard`, `std::unique_lock`
  - Condition variables: `std::condition_variable`
  - Atomic operations: `std::atomic`
- Choose the appropriate primitive based on the required control and performance
- Proper synchronization is key to ensuring thread safety and avoiding data races

std::mutex

```cpp
#include <mutex>
#include <thread>
#include <iostream>

std::mutex mtx;
int counter = 0;

void increment() {
  mtx.lock();
  {
    ++counter; // protected section
  }
  mtx.unlock();
}

int main() {
  std::thread t1(increment);
  std::thread t2(increment);

  t1.join();
  t2.join();

  std::cout << "Final counter value: " << counter << std::endl;
  return 0;
}
```

- Manual locking with `mtx.lock()` and unlocking with `mtx.unlock()`
- Be cautious with exceptions to avoid deadlocks

There is a way to ensure that mutex will be unlocked...

std::mutex and std::lock_guard

```cpp
#include <mutex>

std::mutex mtx;
int counter = 0;

void increment() {
  std::lock_guard<std::mutex> lock(mtx);
  ++counter;  // protected section
}
```

- `std::mutex`: exclusive lock
- `std::lock_guard`: RAII wrapper—locks on construction, unlocks on destruction

std::unique_lock and std::condition_variable

```cpp
#include <mutex>
#include <condition_variable>

std::mutex mtx;
std::condition_variable cv;
bool ready = false;

void worker() {
  std::unique_lock<std::mutex> lock(mtx);
  cv.wait(lock, []{ return ready; });
  // proceed once ready == true
}

void notifier() {
  {
    std::lock_guard<std::mutex> lock(mtx);
    ready = true;
  }
  cv.notify_one();
}

```

`std::unique_lock`: provides flexible locking mechanisms such as deferred locking, timed locking, and manual unlocking/relocking, unlike `lock_guard` which locks immediately and strictly scopes the lock.

`std::condition_variable`: allows one or more threads to wait until a particular condition is met. It's typically used with `unique_lock` to enable the thread to wait and then be notified.

std::atomic

Lock-free primitives for simple data types. Avoid mutex overhead for simple counters and flags Supported operations list:

- `store`: assign a new value
- `load`: retrieve the current value
- `exchange`: replace the value and obtain the old value
- `compare_exchange_weak`/`compare_exchange_strong`: atomically compare and set
- `fetch_add`: add to the value and return the previous value
- `fetch_sub`: subtract from the value and return the previous value
- `fetch_and`: perform bitwise AND and return the previous value
- `fetch_or`: perform bitwise OR and return the previous value
- `fetch_xor`: perform bitwise XOR and return the previous value

Atomic operations usage example

```cpp
#include <atomic>
#include <thread>
#include <iostream>

std::atomic<int> counter(0);

void increment() {
  for (int i = 0; i < 100000; ++i) {
    counter.fetch_add(1, std::memory_order_relaxed);
  }
}

int main() {
  std::thread t1(increment);
  std::thread t2(increment);

  t1.join();
  t2.join();

  std::cout << "Final counter value: " << counter.load() << std::endl;
  return 0;
}
```

`std::atomic` provides lock-free operations. Use `fetch_add` method and load to operate on atomic variables safely

Best practices and recommendations

Best practices and recommendations

- Always join or detach threads before destruction
- Prefer RAII wrappers (`std::lock_guard`, `std::unique_lock`)
- Minimize shared state; prefer message passing or futures
- Be cautious with detached threads, manage lifetimes carefully
- Consider higher-level thread pools (e.g., `std::async`)

When and where to use `std::thread` in real world scenarios

- Fine-grained control over threading
- Building custom schedulers or thread pools
- Interfacing directly with OS thread APIs
- Performance critical sections where you avoid scheduler overhead
- When introducing 3rdparty library is an overkill

Thank You!

References

- cppreference.com: `https://en.cppreference.com/w/cpp/thread`