C++ Thread Fundamentals

2025

André Pereira

- **Concurrency vs. Parallelism**
  - **Concurrency**: Multiple tasks processing in overlapping time periods (not necessarily simultaneously)
  - **Parallelism**: Multiple tasks running at the **exact same time** (requires multiple cores/processing units)

- **Threads in HPC**
  - Shared-memory parallelism on multi-core architectures.
  - Typically used to exploit **intra-node** parallelism (complementary to MPI for **inter-node** parallelism).

- **Why C++ Threads?**
  - Standardised since **C++11**, ensuring portability.
  - Direct, low-level control over thread management and synchronization.
  - High-level abstractions for low-level control
  - Can be combined with higher-level frameworks like **OpenMP** or **TBB** if needed.
  - Fine-grain control over thread behaviour (great for **load balancing**)
    - They will be important later

- **`std::thread`** overview
  - Pass a function, functor, or lambda to a thread constructor
    - `std::thread t([]{/* work */});`

- Thread lifecycle
  - **`join()`**: Blocks until the thread finishes; ensures safe cleanup.
  - **`detach()`**: Thread runs independently; cannot be joined later.

```cpp
void worker(int id) {
    std::cout << "Thread " << id << " is working\n";
}

int main() {
    std::thread t(worker, 1);
    // Must join or detach before exiting
    t.join();
    return 0;
}
```

- **`std::thread`** overview
  - Pass a function, functor, or lambda to a thread constructor
    - `std::thread t([]{/* work */});`
- Thread lifecycle
  - **`join()`**: Blocks until the thread finishes; ensures safe cleanup.
  - **`detach()`**: Thread runs independently; cannot be joined later.

```cpp
void worker(int id) {
    std::cout << "Thread " << id << " is working\n";
}

int main() {
    std::thread t(worker, 1);
    // Must join or detach before exiting
    t.join();
    return 0;
}
```

- Forgetting to **join or detach** => `std::terminate` will be called
- Exiting main while threads are still running => undefined behaviour

- Passing arguments
  - By **value**: Makes a copy.
  - By **reference**: Uses `std::ref` or capture references carefully with lambdas.
  - By **pointer**: Copies the address provided.
  - **Lambda captures**: `[&]`, `[=]`, or selective captures to control data access.

- Returning Results
  - Use **shared data** (protected by mutex or atomic operations).
  - **`std::promise`** + **`std::future`** to send back results or exceptions.

```cpp
std::promise<int> p;
std::future<int> f = p.get_future();

std::thread t([&p](){
    int result = compute_some_value();
    p.set_value(result);
});
// ...
int value = f.get(); // blocks until
                     // set_value is called
t.join();
```

Synchronisation Primitives - Locks

- **`std::mutex`** and variations
  - **`std::timed_mutex`**: Allows timeout-based lock attempts.
  - **`std::recursive_mutex`**: Can be locked multiple times by the same thread (careful!).

- Locking mechanisms
  - **RAII** (Resource Acquisition Is Initialization) with:
    - **`std::lock_guard<std::mutex> lock(mtx)`**: Simple, acquires on construction, releases on destruction.
    - **`std::unique_lock<std::mutex>`**: More flexible, can unlock/lock multiple times.
    - **`std::scoped_lock`**: C++17 feature for multiple mutexes with no deadlock.

- Common pitfalls
  - **Deadlock**: Acquiring multiple locks in an inconsistent order.
  - **Double Locking**: Attempting to lock the same mutex twice from the same thread without using `std::recursive_mutex`.

```cpp
std::mutex m;
int sharedCounter = 0;

void increment() {
    m.lock();
    ++sharedCounter;
    m.unlock();
}

int main() {
    std::thread t1(increment), t2(increment);
    t1.join(); t2.join();
    std::cout << sharedCounter << "\n";
}
```

Synchronisation Primitives - Locks

- Common pitfalls
  - **Deadlock**: Acquiring multiple locks in an inconsistent order.
  - **Double Locking**: Attempting to lock the same mutex twice from the same thread without using `std::recursive_mutex`.

```cpp
std::mutex m;
int sharedCounter = 0;

void increment() {
    std::lock_guard<std::mutex> lock(m);
    ++sharedCounter;
}

int main() {
    std::thread t1(increment), t2(increment);
    t1.join();
    t2.join();
    std::cout << "Counter = " << sharedCounter << "\n";
}
```

Synchronisation Primitives – Condition Variables

- **TBC**