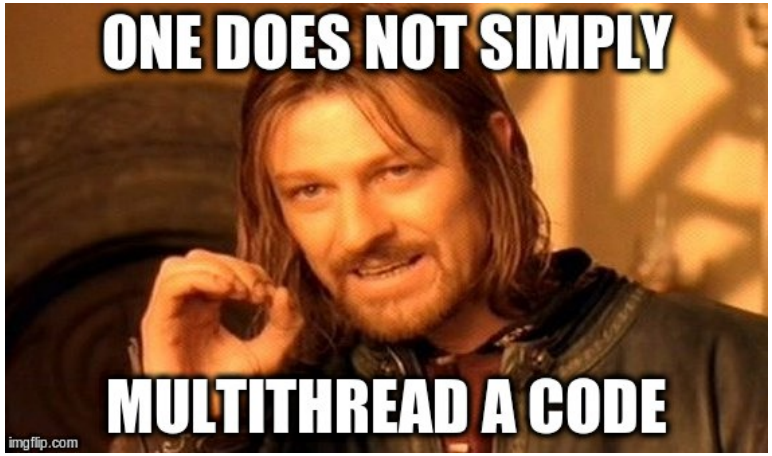
Parallelität und Nebenläufigkeit

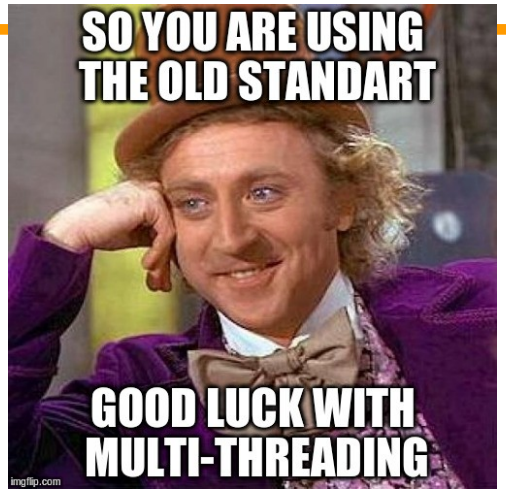
- Grundlagen
- Vektorisierung
- Parallelisierung
- Threads
- Futures
- Atomics



Grundlagen

- Es gibt 3 Arten von asynchronen Abläufen
- 1. Vektorisierung
  - Eine Operation wird für viele Daten angewandt (SIMD Instruktionen)
- 2. Parallelität
  - Viele gleiche Berechnungen werden auf mehrere CPU/Kerne verteilt
- 3. Nebenläufigkeit
  - Viele verschiedene Aufgaben werden gleichzeitig verarbeitet

Grundlagen



- Schwierigkeit ist, dass der Code korrekte Resultate liefert
- Seit C++11 bietet die Sprache mehrere Möglichkeiten für Nebenläufigkeit
- Ältere C++ Standards benötigen eine Externen Bibliotheken
- http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-concurrency

Vektorisierung

- Kann nicht Explizit angegeben werden
  - Einzige Möglichkeit über Befehle für spezielle Prozessorarchitektur
- Aber Compiler versucht mit -O3 Schleifen u.ä. zu Vektorisieren
- Deswegen
  - Einfache, verständliche Schleifen schreiben
  - for_each Schleifen verwenden
- https://godbolt.org/g/9B7a2u

Parallelisierung

- „Nativ“ erst ab C++17 möglich mit Execution policies (siehe Vorlesung über Algorithmen)
  - Explizite Angabe, wie der Algorithmus Parallelisiert werden soll
  - Wird bisher noch von keinem Compiler unterstützt, daher leider kein Codebeispiel
- Ansonsten
  - OpenMP parallelisiert über #pragma Anweisungen
    - Verfügbar in GCC, Clang und MSVC
  - Externe Bibliotheken

Threads

- Seit C++11 gibt es std::threads, die einen Thread in einem Vorgegebenen Threadpool startet
  - Konstruktor übernimmt eine Funktion, gefolgt von den Parametern
  - ID wir automatisch Vergeben
  - Std::join für ein blockierendes warten auf das Thread Ende
  - Std::detach um den Thread außerhalb des Threadpools laufen zu lassen
- Vereinfachte Sleeps mittels std::this_thread::sleep_for()
  - Nimmt Chrono Datentypen für die Zeitdauer
- https://godbolt.org/g/enPR7n

Threads

- Std::Mutex als Elementarer Datentyp um Zugriffe zu schützen
  - Hat bereits fertige lock/unlock implementierung
- Std::lock_guard übernimmt ein Mutex und sperrt es bis zum Desturktoraufruf
  - Hilfreich vorallem in Kombination mit RAII
  - https://godbolt.org/g/2Ay1hY
- Std::Condition_variable übernimmt ein Mutex das extern gelöst werden kann
  - cv.wait übernimmt das lock
  - cv.notify_once löst ein lock
  - cv.noitfiy_all löst alle locks
  - CV kann auch mit einem Timeout versehen werden (wait_until)
  - https://godbolt.org/g/RgijFW

Futures

- In C++11 gibt es Futures für asynchrone Tasks
  - Futures enthalten einen Rückgabetyp eines Tasks, der möglicherweise noch nicht fertig ist
- Gestartet wird der Task mit std::async
- Der Datentyp einens Futures ist ein promise
  - Mittels promise.get_future() wird das promise einem Future zugewiesen
  - Ein promise.set_value() gibt dem Future bescheid, dass der Wert berechnet wurde
- https://godbolt.org/g/is1FYE

Futures

- Futures ab C++20 (oder https://github.com/rpz80/cf)
  - Verkettung von Futures
  - Wenn einer oder mehrere Futures fertig sind, werden danach andere Threads gestartet
- Bisher im Proposal (http://en.cppreference.com/w/cpp/experimental/future)
  - Then(): startet einen Thread danach
  - When_any: wenn ein Thread aus einem Array aus Futures fertig ist
  - when_all: wenn alle Thread aus einem Array aus Futures fertig sind

Atomics

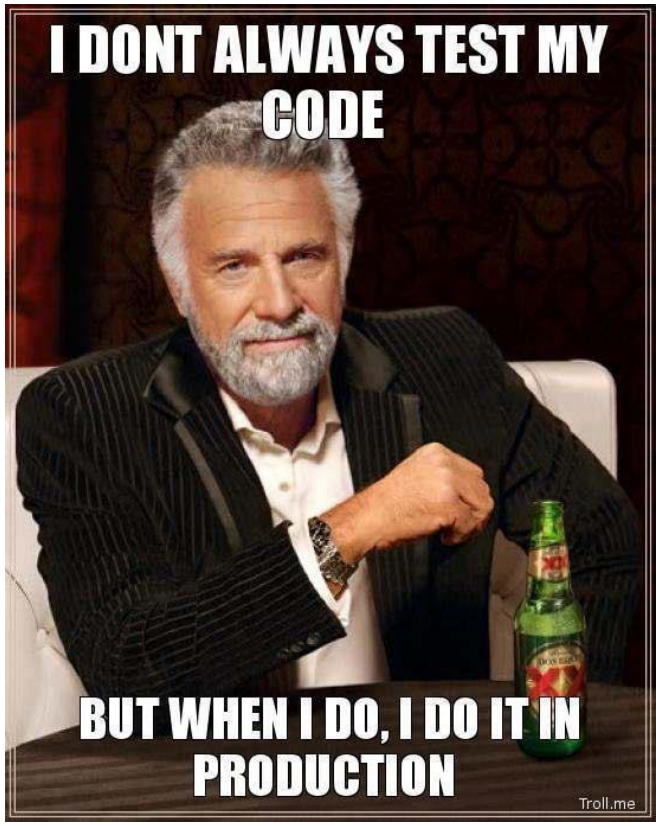
- Seit C++11 Atomics (sind auch in C11 implementiert!!)
- Std::atomic<T> ist ein Template Datentyp, der alle Operationen auf den Datentyp nur atomar erlaubt
  - Vorteil: Variabel generiert keine Dataraces
  - Nachteil: Overhead ist immer Vorhanden
  - Seit C++11: Alle static Variablen sind atomic
- https://godbolt.org/g/fj5sdz

Atomics

- std::atomic_thread_fence ist eine Barriere, deren Eigenschaften mit Memory_orders angegeben werden
  - memory_order_relaxed: keine Synchronisierung
  - memory_order_acquire: sperrt die Barriere für alle nachfolgenden Threads
  - memory_order_release: entsperrt die Barriere
  - Gibt noch ein paar mehr (http://en.cppreference.com/w/cpp/atomic/memory_order)

Sonstiges

- std::atomic_flag für Atomic<bool> mit test_and_set Funktion
- Std::call_once wird nur einmal Aufgerufen, auch wenn andere Threads an die Stelle kommen



Nächtes Mal:

CI / CD / Testing