High Performance Computing für Maschinelle Intelligenz

Multithreading

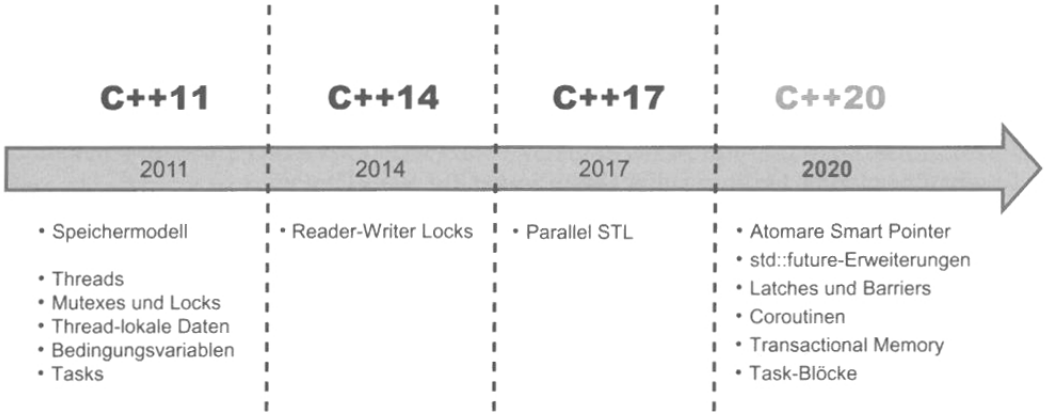
Martin Gottwald und Alice Hein

26. Oktober 2021

- Multithreading
- Beispiel: Paralleles Skalarprodukt
- Verschiedenes
- Fortgeschrittene Mutex Varianten
- Producer-Consumer Modell
- Hausaufgabe

Multithreading

Der C++ Standard



Quelle: „Modernes C++: Concurrency Meistern“ von Rainer Grimm

Stränge und Fäden

- Ein Betriebssystem ordnet einem Prozess den Adressraum und andere Betriebssystemmittel zu
- Innerhalb eines Prozesses können ein oder mehrere Threads existieren:
  - Geteilte Ressourcen (vor allem Arbeitsspeicher)
  - Schnellere Wechsel der Threads in einer CPU
- Ein Thread stellt den Ablauf eines Programms dar („Ausführungsstrang“)
- Es gibt zwei Arten:
  - Kernel-Threads, welche zum Kernel gehören und von dort gesteuert werden
  - User-Threads, die von einem Programm außerhalb verwaltet werden müssen

Beispiel Code

```cpp
#include <iostream>
#include <thread>  // Thanks to modern C++ ...

void call_from_thread()
{
  std::cout << "Hello, World" << std::endl;
}

int main()
{
  std::thread t(call_from_thread); // Launch a thread to run the function

  t.join(); // Join the main thread with the thread stored in t.
}
```

Kompilieren (Linux)

- Befehl: `g++ -std=c++11 -o foo main.cpp`
- Ergebnis:

```
g++ -std=c++11 main.cpp -o foo
  /tmp/tmp.o: In function `std::thread::thread<void (&)()>(void (&)())'
  main.cpp: undefined reference to `pthread_create'
  collect2: error: ld returned 1 exit status}
```

Warum tauchen hier `pthreads` auf?

Lösung (Linux)

- Befehl: `g++ -std=c++11 -o foo main.cpp -l pthread`
- Ergebnis:

```
g++ -std=c++11 main.cpp -o foo -l pthread
  [nix == kein Fehler]
```

- Ausführen: `./foo`

```
./foo
  Hello, World
```

Warum?

- `-l pthread` ist notwendig für den Linker: Linke gegen `pthread`
- „Das sollte nicht länger notwendig sein mit C++ 11! Ich sollte gar nicht wissen müssen, dass `pthread` die Implementierung von `std::thread` stellt.“
- C++11 abstrahiert Threads und versteckt Plattform-spezifische Details
- `std::thread` kann beliebig implementiert werden:
  - *Posix Threads*
  - Thread-Bibliothek des Betriebssystems: `#include <windows.h>`
  - *Boost*
- Man muss wissen, welche Threading-Bibliothek verwendet wird:
  - `g++` mit `-lpthread`
  - Visual C++ braucht einen Haken in den Einstellungen

Vorteil von C++ 11 und `std::thread`

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

void call_from_thread()
{
  std::cout << "Hello, World" << std::endl;

}

int main()
{
  std::thread t(call_from_thread);

  t.join();
}
```

Vorteil von C++ 11 und `std::thread`

```
#include <iostream>
#include <pthread.h>

void* call_from_thread(void*)
{
  std::cout << "Hello, World" << std::endl;
  return NULL;
}

int main()
{
  pthread_t t;
  pthread_create(&t, NULL, call_from_thread, NULL);
  pthread_join(t, NULL);
}
```

Was passiert, wenn `join()` fehlt?

```cpp
#include <iostream, thread>  // Uses less space, this is not valid C++

void call_from_thread()
{
  std::cout << "Hello from thread" << std::endl;

  // Some work to keep a thread busy
  for (unsigned int i = 0; i < 100000; i++) std::cout << i << std::endl;
}

int main()
{
  std::thread t(call_from_thread);
  std::cout << "Hello from main" << std::endl;
  // t.join(); //<- Removed for demonstration
}
```

Was passiert, wenn `join()` fehlt?

```
./build/demo4.exe
  Hello from main
  terminate called without an active exception
  Hello from thread
  Aborted (core dumped)
```

Keine Garantie,

- dass der Thread überhaupt arbeiten kann
- dass das Programm sauber beendet wird (z.B. kein Dekonstruktoraufruf)

Mehrere Threads

```cpp
#include <iostream, thread, vector> // Shortcut to free some lines

void call_from_thread(){ /* code as before */ }

int main()
{
  std::vector<std::thread> threads;
  const int num_threads = 10;

  // Launch a group of threads
  for (int i = 0; i < num_threads; ++i)
    threads.push_back(std::thread(call_from_thread));

  for (auto& t : threads) t.join(); // Join all threads
}
```

Mehrere Threads

- Erstellen eines Array (= `std::vector`) zum Speichern von Thread-Instanzen
  - Standardkonstruktor würde eine Thread-Instanz erstellen
  - Es wird noch nichts gestartet oder ein echter Thread im Code repräsentiert
- Erstellen und Starten der Threads per Schleife
  - Die Zuweisung, welche im `push_back` versteckt ist, triggert den Zustandswechsel des Threads im `std::vector`
  - Der temporäre Thread zwischen den Klammern vom `push_back` wird beim Verlassen des Scopes wieder gelöscht
  - Hier liegt der Teufel im Detail (später mehr)
- Warten auf alle Threads per Schleife

Mehrere Threads

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

// The number of threads is required at several spots in this example
const int num_threads = 10;

// Use some custom integer as Thread-Id.
// std::this_thread::get_id() is also possible,
// but this is a huge and unwieldy number
void call_from_thread(int id)
{
  std::cout << "Hello from thread " << id << std::endl;
}
```

Mehrere Threads

```
// code of last slide

int main()
{
  // C-Style arrays also work, but better stick to modern C++ ...
  std::thread threads[num_threads];

  for (int i = 0; i < num_threads; ++i)
    // Use the loop counter as id for the thread
    threads[i] = std::thread(call_from_thread, i);

  std::cout << "Hello from main" << std::endl;

  for (int i = 0; i < num_threads; ++i) threads[i].join();
}
```

Mehrere Threads - Output

```
./foo
  Hello from thread 0
  Hello from thread Hello from thread 12
  Hello from thread 3

  Hello from thread 4
  Hello from thread 5
  Hello from thread Hello from thread 7
  6
  Hello from main
  Hello from thread 8
  Hello from thread 9
```

- Anzahl der Threads: 11
- Ergebnis wirkt zufällig
- Vermischter Text obwohl `std::endl` den Speicher spült (`std::flush`)
- „Hello from thread“ als Einheit → unteilbarer C-String
- `main`-Thread läuft weder als erstes noch als letztes

⇒ Synchronisation liegt bei euch

Synchronisation

- Code braucht Regeln (`std::mutex`), wenn Threads auf eine gemeinsame Ressource (hier `std::cout`) zugreifen
- Alternativ können getrennte Datenstrukturen verwendet werden
- Beispiel: Bilder falten
  - Ein Bild als Input
  - Ein weiteres, um das Ergebnis zu speichern
  - Simultanes lesen ist kein Problem
  - Wenn das Bild in n-Teile für n-Threads zerlegt wird, gibt es keine Kollision
  - Jeder schreibt in seinen eigenen Bereich

Der Teufel liegt im Detail

```cpp
#include <iostream, thread, vector> // Free some lines

void call_from_thread(){ /* code as before */ }

int main()
{
  std::vector<std::thread> threads;
  const int num_threads = 10;

  for (int i = 0; i < num_threads; ++i) // Launch a group of threads
    threads.push_back(std::thread(call_from_thread));

  // Join the threads with the main thread
  for (auto& t : threads) t.join();
}
```

Der Teufel liegt im Detail

```
#include <iostream, thread, vector> // Free some lines

void call_from_thread(){ /* code as before */ }

int main()
{
  [...]

  for (int i = 0; i < num_threads; ++i) // Launch a group of threads
  {
    std::thread t(call_from_thread)
    threads.push_back(t);
  }
  [...]
}
```

Der Teufel liegt im Detail

```
make demo3.exe
  mkdir -p build
  g++ -o build/demo3.exe demo3.cpp -l pthread
  In file included from /usr/include/x86_64-linux-gnu/c++/9/bits/c++allocator.h:33,
                   [...]
                   from demo3.cpp:1:
  /usr/include/c++/9/ext/new_allocator.h: In instantiation of ‘void
  __gnu_cxx::new_allocator<_Tp>::construct(_Up*, _Args&& ...) [with _Up = std::thread;
   _Args = {const std::thread&}; _Tp = std::thread]’:
  [...]
  /usr/include/c++/9/ext/new_allocator.h:145:20: error: use of deleted function
  ‘std::thread::thread(const std::thread&)’
    145 |  noexcept(noexcept(::new((void *)__p)
        |                    ^~~~~~~~~~~~~~~~~
    146 |         _Up(std::forward<_Args>(__args)...)))
  In file included from demo3.cpp:2:
  /usr/include/c++/9/thread:142:5: note: declared here
    142 |     thread(const thread&) = delete;
        |     ^~~~~~
  make: *** [Makefile:15: demo3.exe] Error 1
```

Der Teufel liegt im Detail

- Der Compiler beschwert sich, dass ein `std::thread` keinen Copy-Konstruktor hat: `142 | thread(const thread&) = delete;`
- `threads.push_back(t);` muss aber `t` kopieren → Problem
- `threads.push_back(std::thread(call_from_thread));` verwendet das überladene `std::vector::push_back` für r-Value Referenzen, es wird nichts kopiert, sondern nur bewegt
- `threads.push_back(std::move(t));` verwendet explizit die „C++ move semantics“, der Compiler macht wieder mit

Wie viele Threads braucht man?

- Keine pauschale Antwort, hängt von Daten, Hardware und Code ab
- Faustregel: Pro (virtuellem) Kern ein Thread
- Ein Thread pro echtem Kern kann schneller sein als einer pro virtuellem

Hyper-Threading

- Es kann nur ein Thread pro echtem Kern zur selben Zeit arbeiten
- Ein zweiter kann auf Bereitschaft in der CPU verbleiben
- Dies nennt man Hyper-Threading (Intel)
- Wenn einer der beiden warten muss, kann der andere schnell Einspringen
- Dadurch werden weniger CPU-Zyklen verschwendet

Wann wartet ein Thread?

- Bei I/O Operationen:
  - Kommunikation mit Peripherie
  - Festplatten, welche erst anlaufen müssen
- Bei einem Cache-Miss mit anschließendem Warten auf den RAM
- Netzwerkkommunikation
- Bei homogenen Aufgaben, z.B. parallele Matrix-Vektor Multiplikation, ist es besser nur einen Thread im Kern zu haben, diesen aber permanent

Beispiel: Paralleles Skalarprodukt

Beispiel: Skalarprodukt

- Für zwei Vektoren $u, v \in \mathbb{R}^n$ und ein (großes) $n \in \mathbb{N}$ gilt

$$\langle u, v \rangle = \sum_{i=0}^{n} u_i \cdot v_i$$

- Wähle $m << n$ Blöcke, summiere parallel in jedem Block

$$\{1, 2, 3, 4, 5, \ldots, n-1, n\} \Rightarrow \left\{ \underbrace{(1, 2, 3, 4)}_{B_1}, \underbrace{(5, 6)}_{B_2}, \underbrace{(7, 8, 9)}_{B_3}, \underbrace{(10)}_{B_4}, \ldots, \underbrace{(n-42, \ldots, n)}_{B_m} \right\}$$

$$\langle u, v \rangle = \sum_{B_k} \sum_{i \in B_k} u_i \cdot v_i$$

Beispiel: Skalarprodukt

```
#include <iostream, thread, vector> // Free some lines

//Split "range" into "parts": parts = 4 and range = 10 -> bnd=[0,2,4,6,10]
void bounds(std::vector<int>& bnd, int parts, int range)
{
  int delta = range / parts, reminder = range % parts;
  int N1 = 0, N2 = 0;
  bnd.push_back(N1);
  for (int i = 0; i < parts; ++i){
    N2 = N1 + delta;
    if (i == parts - 1) N2 += reminder;
    bnd.push_back(N2);
    N1 = N2;
  }
}
```

Beispiel: Skalarprodukt

```
// code from last slide

// Partial inner product starting from L and running until R.
void dot_product(const std::vector<int>& v1,
                 const std::vector<int>& v2,
                 int& result,
                 const int L, const int R)
{
  for (int i = L; i < R; ++i)
  {
    result += v1[i] * v2[i];
  }
}
```

Beispiel: Skalarprodukt

```
// code from last slides
int main()
{
  int nr_elements = 1e5, nr_threads = 5, result = 0;
  std::vector<std::thread> threads;

  std::vector<int> v1(nr_elements, 1), v2(nr_elements, 2);
  std::vector<int> bnd; bounds(bnd, nr_threads, nr_elements);

  for (int i = 0; i < nr_threads; ++i)
    threads.push_back(std::thread(dot_product, std::ref(v1), std::ref(v2),
                                  std::ref(result), bnd[i], bnd[i + 1]));

  for (auto& t : threads) t.join(); std::cout << result << std::endl;
}
```

Beispiel: Skalarprodukt

Kompilieren und Ausführen:

```
./foo
  82618
./foo
  65658
./foo
  70506
./foo
  57912
./foo
  62290
```

- Ergebnis ist nicht $2 \cdot 10^5$!
- Output variiert, warum?
- Wir haben zwar getrennte Blöcke, aber `result` wird asynchron verwendet!

Race Condition

- Problematisch ist die Zeile: `result += v1[i] * v2[i];`
- Abhängig von der zeitlichen Reihenfolge wird die Variable von verschiedenen Threads geschrieben ohne vorher den neuen Wert zu laden
- Dies wird als *race condition* bezeichnet
- Man spricht gerne auch von einem „Heisenbug“:
  $\rightarrow$ Bug verschwindet beim Zuschauen (Debuggen)
- Tritt zum Beispiel auch in logischen Schaltungen auf

Race Condition

Thread 1	Thread 2		Variable
			0
read		←	0
add 2			0
write		→	2
	read	←	2
	add 2		2
	write	→	4

Thread 1	Thread 2		Variable
			0
read		←	0
	read	←	0
add 2			0
	add 2		0
write		→	2
	write	→	2

Zeit läuft von oben nach unten

Mutual Exclusion

- Die Variable `result` muss synchronisiert verwendet werden
- Man definiert dazu einen Bereich im Code, in dem nur ein Thread zur selben Zeit sein darf
- Das sich gegenseitige Ausschließen wird mit der Klasse `std::mutex` erreicht
- **Mut**ual **Ex**clusion $\rightarrow$ Mutex

Mutual Exclusion

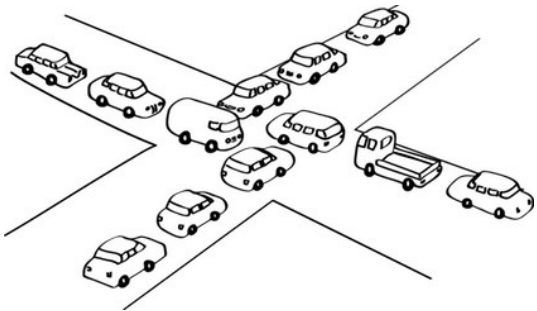
- Um an einer Mutex Instanz vorbeizukommen, muss diese per Funktion `lock()` gesperrt werden
- Ist dies nicht möglich (ein anderer Thread war zuerst da), ist `lock()` ein blockierender Aufruf
- Sobald die kritische Arbeit fertig ist, muss per Funktion `unlock()` die Mutex Instanz wieder frei gegeben werden und ein anderer Thread kann weiter machen
- `try_lock()` ist immer nicht-blockend, es muss der Rückgabewert (`true` / `false`) beachtet werden

std::mutex

```cpp
#include <mutex>
std::mutex mutex;  // Use meaningful names if you have more than one ...

void dot_product(const std::vector<int>& v1, const std::vector<int>& v2,
                 int& result, const int L, const int R)
{
  int partial_sum = 0;    // One per thread -> private memory to work with

  for (int i = L; i < R; ++i)
    partial_sum += v1[i] * v2[i];

  mutex.lock();           // Other threads wait here
  result += partial_sum; // synchronized adding
  mutex.unlock();         // Unlock by hand
}
```

Gefahr beim Sperren von Hand

- Auf jedes `lock()` muss ein `unlock()` folgen
- Passiert nicht bei Sprüngen im Programmablauf (z.B. `try` und `catch`)
- Dadurch kann ein Deadlock entstehen:
  - Thread wartet auf sich selbst
  - Alle Threads warten auf andere



std::lock_guard

```cpp
#include <mutex>
std::mutex mutex;  // Use meaningful names if you have more than one ...

void dot_product(const std::vector<int>& v1, const std::vector<int>& v2,
                 int& result, const int L, const int R)
{
  int partial_sum = 0;   // One per thread -> private memory to work with

  for (int i = L; i < R; ++i)
    partial_sum += v1[i] * v2[i];

  std::lock_guard<std::mutex> some_name_to_avoid_anonymous_objects(mutex);
  result += partial_sum; // synchronized adding

} // Leaving the scope destroys the guard -> mutex is unlocked
```

std::lock_guard

- `std::lock_guard` übernimmt Sperren und Freigeben eines Mutex
- Der kritische Bereich `result += partial_sum;` ist threadsicher
- Wird die Instanz zerstört wird `unlock()` aufgerufen
- Beim Verlassen des Bereichts (Exception, Goto, per `return`, …) passiert dies automatisch
- `mutex.unlock()` kann nicht mehr übersprungen werden $\Rightarrow$ kein Deadlock

```
void some_function()
{
  mutex.lock();

  try { some_work(); }
  catch (std::string e)
  {
    mutex.unlock();
    throw e;
  }
  mutex.unlock();
}
```

std::lock_guard

- Scopes können jederzeit erstellt werden
- Einfach ein Paar aus geschweifte Klammern platzieren
- Sauberer Code
- Schaut manchmal komisch aus …

```
void some_function()
{
  some_parallel_work();

  {
    std::lock_guard<std::mutex> guard(mutex);
    some_synchronized_work()
  }

  some_parallel_work();
}
```

RAII

- Das zugrunde liegende Konzept heißt:

Ressource **A**quisition **i**s **I**nitialization

- Das Belegen einer Ressource ist an die Lebenszeit eines Objekts gebunden
- Bei einem `std::lock_guard` bedeuted dies:
  - Im Konstruktor findet `mutex.lock()` statt
  - Im Dekonstruktor befindet sich `mutex.unlock()`
  - C++ garantiert den Aufruf des Dekonstruktors am Ende der Lebenszeit
  - Das Objekt, sprich der `std::lock_guard`, **muss** auf dem Stack erzeugt werden!

Atomic Types

```
#include <atomic>
void dot_product(const std::vector<int>& v1, const std::vector<int>& v2,
                 std::atomic<int>& result, const int L, const int R)
{
  int partial_sum = 0;
  for (int i = L; i < R; ++i) partial_sum += v1[i] * v2[i];
  result += partial_sum; // synchronized adding
}
int main()
{
  int nr_elements = 100000, nr_threads = 5;
  std::atomic<int> result(0);

  // Remaining Code as before
}
```

Atomic Types

- Hier (und nur hier) die deutlich einfachere Lösung
- *Atomic Types* sind spezielle Datentypen, welche von alleine threadsicher sind
- Paralleles Lesen/Schreiben wird durch den Compiler synchronisiert
- Nur für Integer-Datentypen im *atomic* Header vordefiniert:
  - `std::atomic<char>`
  - `std::atomic<unsigned long long>`
  - `std::atomic<bool>`
  - ...

Verschiedenes

Member Funktion in eigenem Thread

```
struct SayHello
{
  void call_in_thread(const std::string& name)
    { std::cout <<"Hello from thread " << name << std::endl; }
};

int main(int argc, char* argv[])
{
  SayHello x;

  //Syntax: member function pointer, pointer to instance, arguments
  std::thread t(&SayHello::call_in_thread, &x, std::cref("Foo"));

  t.join();
}
```

Lambda Funktionen und Threads

```
#include <iostream, thread>

using namespace std;

int main(int argc, char* argv[])
{
  //Syntax for anonymous (lambda) function: function, arguments
  std::thread t([](int a, int b) { cout << "Result: " << a * b << endl; },
                42, 24);

  t.join();
}
```

Call once

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

std::once_flag flag;

void do_something(){
    std::call_once(flag, [](){std::cout << "Called once" << std::endl;});
    std::cout << "Called each time" << std::endl;
}

int main(){
    std::thread t1(do_something), t2(do_something), t3(do_something);

    t1.join(); t2.join(); t3.join();
}
```

Fortgeschrittene Mutex Varianten

Recursive Locking

```
struct MyData
{
  std::mutex mtx; int data;

  MyData(): data(0) {}

  void add(int x){
    std::lock_guard<std::mutex> l(mtx);
    data += x;
  }
  void s_add(int c, int x){
    std::lock_guard<std::mutex> l(mtx);
    data *= c; add(x);
  }
};
```

- Guter Stil: Kein globales `std::mutex`
- Einfaches Programm:

```
int main(){
  MyData d;
  std::thread t(&MyData::s_add,&d,2,3);
  t.join();
  std::cout << d.data;
}
```

- Output in der Konsole:
  Nichts, Deadlock!
- Das Mutex wird gesperrt, danach blockiert sich der Thread selbst

Recursive Locking

```
struct MyData
{
  std::recursive_mutex mtx; int data;

  MyData(): data(0) {}

  void add(int x){
    std::lock_guard<std::recursive_mutex> l(mtx);
    data += x;
  }
  void s_add(int c, int x){
    std::lock_guard<std::recursive_mutex> l(mtx);
    data *= c; add(x);
  }
};
```

- Ein Standard Mutex kann nicht mehrmals gesperrt werden
- `std::recursive_mutex` können dies, aber nur beim selben Thread
- Das Programm von vorhin läuft nun

Timed Locking

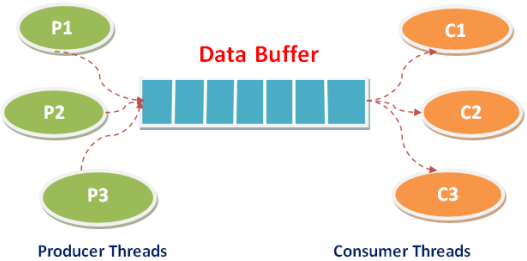
```
std::timed_mutex mutex;

void work(){
  std::chrono::milliseconds timeout(42);

  while(true){
    if(mutex.try_lock(timeout)){
      // Work with mutex
      mutex.unlock();
    }
    else{
      // Work without mutex
    }
  }
}
```

- Manchmal soll ein Thread nicht unbegrenzt warten
- Die Zeit, bis eine Ressource frei wird, kann sinnvoll genutzt werden
- Hängt vom Problem ab ob dies sinnvoll/machbar ist

Producer-Consumer Modell

Schema

- Viele Anwendungsgebiete
  - ROS mit Netzwerkkommunikation
  - MQTT für das Internet der Dinge
- Gleichzeitiges Produzieren der Daten
- Gleichzeitiges Konsumieren der Daten
- Eine Form von Warteschlange zum Entgegennehmen und Verteilen



Condition Variables

- Eine `std::condition_variable` erlaubt Threads:
  - Mutex sperren und freigeben
  - Warten, bis Daten eines anderen Thread vorliegen
  - Zeitlich begrenztes Blocken
- Stellen explizite inter-thread Kommunikation dar
- Condition Variables sind eigentlich Instanzen der Klasse:
  `std::condition_variable cv;`
- Warten / Testen einer Bedingung mit `cv.wait(...);`
- Kommunizieren, dass Daten etc. verfügbar sind:
  - `cv.notify_one();`
  - `cv.notify_all();`

Producer-Consumer mit `std::condition_variable`

```
// Somewhere in MyQueue Class
std::mutex a_mutex;

void MyQueue::put_on_queue(const std::vector<int>& data)
{
  std::lock_guard<std::mutex> lock(a_mutex);

  if (data.size() + m_count > CAPACITY ) { /* code to wait for capacity */ }

  handle_new_values(data); // new elements must be copied at some point
}
MyQueue::pop_from_queue(std::vector<int>& data)
{
  std::lock_guard<std::mutex> lock(a_mutex);

  if (m_count - data.size() <= 0 ) { /* code to wait for more data */ }

  handle_removing_of_values(data); // Copy elements in 'data'
}
```

MyQueue:put_on_queue

```
// Somewhere in MyQueue
std::condition_variable hasCapacity, hasData;

void MyQueue::put_on_queue(const std::vector<int>& data)
{
  std::unique_lock<std::mutex> lock(a_mutex);

  // Lambda function which checks for free space,
  // true or false determines whether wait() finishes
  hasCapacity.wait(lock, [this, &data] {
    if (data.size() + m_count > CAPACITY) return false; else return true;
  });

  handle_new_values(data);

  // Notify one of the threads that new data is available
  // -> One thread checks the wait condition
  hasData.notify_one();
}
```

MyQueue:pop_from_queue

```
// Somewhere in MyQueue
std::condition_variable hasCapacity, hasData;

MyQueue::pop_from_queue(std::vector<int>& data)
{
  std::unique_lock<std::mutex> lock(a_mutex);

  // Lambda function which checks for enough data,
  // true or false determines whether wait() finishes
  hasData.wait(lock, [this, &data] {
    if (m_count - data.size() <= 0) return false; else return true;
  });

  handle_removing_of_values(data);

  // Notify one of the threads that capacity has increased
  // -> One thread checks the wait condition
  hasCapacity.notify_one();
}
```

`std::lock_guard` vs. `std::unique_lock`

- Gleiche Verwendung:
  ```
  std::lock_guard guard1(mutex_1);
  std::unique_lock guard2(mutex_2);
  ```
- Gleiches verhalten: Mutex gesperrt in Konstruktor und freigegeben in Destruktor
- Unterschied: `std::unique_lock` kann man von Hand (ent)sperren
- `guard2.unlock();` entsperrt (vorzeitig)
- `guard2.lock();` um es (wieder) zu sperren
- Die Funktionen einer `std::condition_variable` erfordern ein `std::unique_lock`
- Generell: Nur verwenden wenn notwendig, `std::lock_guard` ist robuster

Hausaufgabe

Barnsley Farn

- Ein Fraktal benannt nach Michael Barnsley
- Wird erzeugt durch das dynamische System

$$f(\vec{x}) = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix}$$

- Es gibt vier verschiedene Sätze an Parametern
- Welcher verwendet wird, wird nach jeder Iteration gewürfelt

Barnsley Farn auf Wikipedia

Parallele Berechnung des Barnsley Farns

- Eine genaue Beschreibung gibt es als PDF auf Moodle
- Ziel der Hausaufgabe ist es ein Producer-Consumer-Modell umzusetzen:
  - Mehrere Produzenten (Threads) generieren Punkte des Farns
  - Diese Punkte werden in einen Buffer geschrieben (Synchronisation)
  - Verbraucher (Threads) holen sich die Punkte aus dem Buffer und tragen sie in ein Bild ein (Synchronisation)
- Zeichnet euren eigenen Farn indem ihr die Parameter und Farbe verändert

Barnsley Farn

- Basisklasse um die Threads zu verwalten
- Rein abstrakte Template-Basisklasse für Produzenten und Verbraucher um den Buffer zu verwalten
- Die eigentliche Funktionalität kommt per Vererbung hinzu
- Der Buffer soll auch als Template-Klasse angelegt werden damit beliebige Datentypen möglich sind
- Um in C++ ein Bild zu bearbeiten kommt die „Cool Image“ Bibliothek zum Einsatz
  - Sie steht auf der Webseite zum herunterladen bereit
  - Wirbt mit „thread-safe“, dennoch solltet ihr die benötigten Funktionen (Pixel verändern) anschauen und überprüfen

Barnsley Farn

- Ihr habt demnächst Zugang zum Praktikumsraum (virtuell)
- Das Eikon ist seit gestern wieder verfügbar
- Die Hausaufgaben müssen auf den unseren Rechnern auf Anhieb kompilieren und laufen
- Eine `ssh` Anleitung ist in unserem Wiki, der Link ist auch auf Moodle
- Abgabe ist in zwei Wochen via Moodle

Bis nächste Woche