TECHNISCHE UNIVERSITÄT MÜNCHEN
Fakultät für Elektrotechnik und Informationstechnik
Lehrstuhl für Datenverarbeitung
Prof. Dr.-Ing. K. Diepold

High Performance Computing für Maschinelle Intelligenz

Hausaufgabe 1

Erstellt am 26. Oktober 2021

1 Barnsley Farn

Der Barnsley Farn ist ein Fraktal, benannt nach dem Mathematiker Michael Barnsley. Eine mögliche Realisierung (passend zu den Werten der Tabelle) ist in Abbildung 1 zu sehen. Das Fraktal entsteht durch das Iterieren eines linearen dynamischen Systems

$$f(\vec{x}) = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix} \tag{1}$$

mit vier verschiedenen Sätzen $f_i = [a, b, c, d, e, f]$ aus Werten für die sechs Parameter. Welcher Satz gewählt wird, wird nach jeder Iteration (ausführen von Gleichung (2)) gewürfelt. Einen Überblick über alle Parameter und deren Wahrscheinlichkeit gibt es in der Tabelle weiter unten. Als Startpunkt wird immer $\vec{x}_0 = [0, 0]$ gewählt. Alle Nachfolger werden durch die Abbildung

$$\vec{x}_{j+1} = f_i(\vec{x}_j) \quad i \in \{1, 2, 3, 4\} \tag{2}$$

bestimmt. Der Wert von i wird gemäß den Wahrscheinlichkeiten in der Tabelle vor dem Iterieren gewürfelt. Der Farn nimmt eine Fläche von $-2.2 < x_1 < 2.7$ in x_1-Richtung und $0.0 < x_2 < 10.0$ in x_2-Richtung ein (wichtig für das Zeichnen).

Abbildung 1: Barnsley Farn von Wikipedia

f_i	a	b	c	d	e	f	$p(f_i)$
f_1	0	0	0	0.16	0	0	0.01
f_2	0.85	0.04	−0.04	0.85	0	1.6	0.85
f_3	0.2	−0.26	0.23	0.22	0	1.6	0.07
f_4	−0.15	0.28	0.26	0.24	0	0.44	0.07

2 Producer-Consumer Modell

Das Producer-Consumer Modell ist ein typischer Ansatz in Multithreading-Anwendungen. Hierbei muss die Synchronisation mehrerer Threads explizit gelöst werden.

Einer oder mehrere Produzenten erzeugen Daten und stellen sie den Verbrauchern über eine Warteschlange mit endlicher Größe (dem „Buffer“) zur Verfügung. Wiederum ein oder mehrere Verbraucher leeren die Warteschlange nach und nach indem die Daten aus der Warteschlange entfernt und verarbeitet werden.

Produzenten dürfen nur neue Daten hinzufügen wenn Platz ist und Verbraucher können nur laufen, solange unverarbeitete Daten vorliegen. Andernfalls müssen die jeweiligen Threads blockiert werden.

Für die Hausaufgabe soll das Producer-Consumer Modell für den Barnsley Farn umgesetzt werden. Erstellt eine saubere Hierarchie, in welcher die jeweiligen Funktionen in entsprechenden Klassen „versteckt“ werden. Dadurch erhaltet ihr ein Programm, welches von der eigentlichen Problemstellung entkoppelt und somit universal einsetzbar ist.

Jede der folgenden Seite widmet sich einer Komponente. Lest euch die Beschreibung durch und gebt dem Compiler etwas zu tun. Die Reihenfolge ist größtenteils egal.

2.1 Worker

Sowohl Produzenten als auch Verbraucher müssen in einem eigenen Thread laufen. Es bietet sich an, eine gemeinsame Basisklasse `Worker` zu erzeugen, welche sich um das Starten, Stoppen und regelmäßige Aufrufen einer Funktion kümmert. Erstellt hierfür eine Klasse `Worker` mit:

- einer privaten Variable `m_thread`, welche den Thread speichert.
- einer privaten Variable `m_terminate`, sobald diese auf `true` gesetzt wird soll der Thread zum Ende kommen.
- einer privaten Variable `m_running`, welche genutzt werden kann, um mehrfaches Starten zu verhindern.
- öffentliche Funktionen zum Starten und (vorzeitigen) Stoppen des `Worker`. Man kann auf zwei Arten Stoppen: Warten bis die Arbeit regulär erledigt ist indem man auf den internen Thread per `join()` wartet oder vorzeitig terminieren indem man `m_terminate` überschreibt.
- einer rein virtuellen Funktion `bool step()` mit Zugriffsmodifikation `protected`, ohne Argumente und einem Boolean als Rückgabewert. Der Rückgabewert soll angeben, ob ein `Worker` zum Ende kommen soll weil die Aufgabe erledigt ist. Diese Funktion muss demnach von der Kind-Klasse implementiert werden und der `Worker` ist dadurch eine rein abstrakte Basisklasse.
- einer privaten Funktion `void work()`, welche eine Schleife beinhaltet und wiederholt `step()` aufruft bis der Rückgabewert `false` ist. Diese Funktion läuft in dem privaten Thread der Klasse (vgl. Vorlesung 2 Folie 46). Wird `m_terminate` von außen gesetzt wird die Schleife unabhängig vom Rückgabewert beendet.
- Achtet auf einen vollständigen Konstruktor und Destruktor, damit alles initialisiert und sauber aufgeräumt wird.

2.2 Producer

Diese Klasse kümmert sich um das Verarbeiten der erzeugten Daten, sprich die Manipulation des Buffers. Die eigentliche Berechnung der Daten (die Punkte des Farns) findet in einer anderen Klasse statt (siehe Abschnitt 2.3).

Es werden später mehrere Instanzen der `Producer` existieren, welche parallel Punkte erzeugen und in den gemeinsamen Buffer schieben.

Damit dieser Vorgang für beliebige Datentypen funktioniert muss der `Producer` eine Template Klasse sein:

- Erstellt eine Template-Klasse `Producer` mit dem Template-Typ `T`, welche von `Worker` erbt.
- Der Produzent speichert eine private Referenz `Buffer<T>& m_buffer`, welche im Konstruktor auf einen passenden existierenden Puffer gesetzt wird. Die Referenz ist privat, damit Kinder dieser Klasse nicht hinein pfuschen können.
- Im Produzenten wird die rein virtuelle Methode `step()` des `Worker` implementiert. Darin wird ein neuer Datenpunkt per `produce(...)` erstellt und dem Buffer übergeben. Gegenbefalls wird über den Rückgabewert der Thread in der Basisklasse gestoppt.
- `bool produce(T& data)` ist eine neue rein virtuelle Methode der `Producer`, welche passend zu `step()` über den Rückgabewert das Ende der Arbeitsschleife kontrolliert. Das einzige Argument ist eine Referenz vom Typ `T`. Ein Kind von `Producer` kann in dieser Funktion die eigentliche Arbeit implementieren, entscheiden wann es fertig ist und über die Referenz neue Daten transportieren. Unnötiges kopieren wird dadurch vermieden.

2.3 Der echte Produzent

In dieser Klasse findet die eigentliche Berechnung des Farns statt. Hier werden alle Parameter und der Zustand für das iterierte Funktionensystem (1) gespeichert.

Zunächst müsst ihr euch für eine Darstellung eines Punktes entscheiden. Als Datentyp ist denkbar:

- ein `std::pair<float,float>`
- ein `struct` mit zwei Feldern für die Koordinaten
- ein `unsigned long long int`, dessen erste 32 Bit die eine Koordinate darstellen und die zweiten 32 Bit die andere
- …

Eventuell bietet sich ein `typedef`, an um die Schreibarbeit zu reduzieren.

- Diese Klasse erbt von `Producer`, der Typ für das Template kann nun explizit angegeben werden. Es werden Punkte in den Buffer geschrieben.
- Alle 24 Parameter ($4 \times 6 = 24$), die Wahrscheinlichkeiten und der Zustandsvektor sind private Variablen dieser Klasse. Hinzu kommt noch ein Zähler für die generierten Punkte. Eine Instanz dieser Klasse besitzt somit alles um unabhängig von dem restlichen Code in einem Thread laufen zu können (Stichwort Kapselung und Thread-Sicherheit).
- Der Konstruktor von `Producer` braucht wiederum die Referenz auf einen existierenden Buffer (mit gleichem Template-Typ), um den vererbten Konstruktor aufrufen zu können. Zusätzlich könnt ihr die Parameter das Farns übergeben, um bequem euren eigenen zeichnen zu können. Vergesst nicht den Startpunkt der Iteration und den Zähler zu initialisieren.
- Implementiert `produce()`, legt das Ende der Arbeit über die Anzahl der zu erzeugenden Punkte fest ($\sim 10^8$ pro Produzent um schöne Farne zu bekommen, deutlich weniger zum Testen). In dieser Funktion findet das Würfeln der Parameter statt, nutzt den `std::random_device` Mechanismus um aus den vier Fällen gemäß der Wahrscheinlichkeiten auszuwählen. Anschließend entsteht der nächste Punkt des Farns per Gleichung (2).
- Achtet darauf, wo `std::random_engine`, `std::random_device` und die Wahrscheinlichkeitsverteilung definiert werden, um einerseits keine korrupten Zufallszahlen zu bekommen und um andererseits nicht unnötig Rechenleistung zu verschwenden. Der Ansatz per `rand()` und `RAND_MAX` ist definitiv kein moderner C++ Code.

Hinweis: Das erzeugen eines einzelnen Punktes ist keine rechenintensive Aufgabe. Deswegen entstehen sehr viele und teure Aufrufe für den Buffer. Die optimale Anzahl an Produzenten und Verbrauchern wird unterschiedlich sein und muss durch Ausprobieren herausgefunden werden.

Im schlimmsten Fall läuft der parallelisierte Code langsamer als wenn man es „direkter“ Implementiert. Das Ziel der Hausaufgabe ist es, praktische Erfahrung mit den Threads zu bekommen, **nicht** optimal den Farn zu zeichnen.

2.4 Consumer

Ein `Consumer` ist quasi identisch zu den Produzenten, nur dass der Buffer eben geleert wird. Das eigentliche Verarbeiten der Daten (hier das Zeichnen des Farns) findet in einer anderen Klasse statt (siehe Abschnitt 2.5).

- Erstellt eine Template-Klasse `Consumer` mit dem Template-Typ `T`, welche von `Worker` erbt.
- Der Verbraucher speichert eine private Referenz `Buffer<T>& m_buffer`, welche im Konstruktor auf einen passenden existierenden Puffer gesetzt wird. Die Referenz ist privat, damit Kinder dieser Klasse nicht hinein pfuschen können.
- Im Verbraucher wird die rein virtuelle Methode `step()` des `Worker` implementiert. Darin wird ein neuer Datenpunkt aus dem Buffer genommen und per `consume(...)` verarbeitet. Gegenfalls wird über den Rückgabewert der Thread in der Basisklasse gestoppt.

  Da die Verbraucher meistens nicht wissen, wann die Produzenten fertig sind, wird das Ende zusätzlich über den Buffer gesteuert. Wenn dieser für eine gewisse Zeit leer bleibt, wirft der Buffer einen `std::runtime_error`. Fangt diese Ausnahme in `step()` des Verbrauchers auf und stoppt den Thread der Basisklasse, indem ein `true` zurückgegeben wird. Andernfalls steuert der Rückgabewert von `consume()` den Thread.
- `bool consume(const T& data)` ist eine neue rein virtuelle Methode der Verbraucher, welche passend zu `step()` über den Rückgabewert das Ende der Arbeitsschleife kontrolliert. Das einzige Argument ist eine konstante Referenz vom Typ `T`. Ein Kind von `Consumer` kann in dieser Funktion die eigentliche Arbeit implementieren und über die Referenz neue Daten empfangen. Unnötiges kopieren wird dadurch vermieden und es ist sichergestellt, dass die Daten nicht versehentlich verändert werden können.

2.5 Der echte Verbraucher

In dieser Klasse werden die Punkte des Farns in ein gemeinsames Bild eingezeichnet. Mehrere Verbraucher holen sich Koordinaten aus dem Buffer, konvertieren diese in Pixelwerte (siehe Bereichsangabe am Anfang) und erhöhen den Farbwert an dieser Stelle um eins (solange der Wert kleiner 255 ist).

Das Bild sollte groß sein ($10k \times 20k$ Pixel) um das Fraktal gescheit aufzulösen. Mit dem Datentyp `unsigned char` sind das ungefähr 500 MB Arbeitsspeicher. Das resultierende `.png` ist wegen der Komprimierung deutlich kleiner. Für die Bildverarbeitung in C++ gibt es die Cool Image Bibliothek. Auf Moodle steht im `.zip` Archiv von heute ein kleines Beispiel bereit.

Da mehrere Verbraucher auf das Bild zugreifen muss natürlich auf die Synchronisation geachtet werden. Jedes einzelne Pixel mit einem Mutex zu versehen ist sicherlich zu viel des Guten (ein Array aus $2 \cdot 10^8$ `std::mutex` braucht bereits 8 GB Arbeitsspeicher), während ein Mutex für das gesamte Bild den Vorteil mehrerer Verbraucher zunichte macht. Überlegt euch, wie das Bild (grob) in Regionen eingeteilt werden kann und schützt jede Region mit einem Mutex. Es geht hier um die Balance zwischen Speicherverbrauch und Performanz, welche nur durch Ausprobieren optimiert werden kann.

Der echte Verbraucher ist ähnlich zu einem echten Produzenten aufgebaut:

- Diese Klasse erbt von `Consumer` und verwendet den gleichen Datentyp wie der Produzent.
- Es gibt eine private und statische Variable für das Bild. Diese Variable wird zwischen allen Instanzen geteilt (da sie statisch ist), der Schreibzugriff muss also synchronisiert werden. Genau ein Konsument muss das Bild am Ende speichern (z.B. wenn am Ende der `main` Funktion aufgeräumt wird, der letzte Verbraucher in der Liste speichert alles)
- `consume()` kann nicht entscheiden ob der Verbraucher fertig ist (solange kein Fehler Auftritt, z.B. `I/O-Error`). Daher ist der Rückgabewert einfach immer `false` und die Basisklasse soll sich um das Beenden via `std::runtime_error` kümmern.

Hinweis: Das Verarbeiten der Punkte stellt einen anderen Aufwand als das Erzeugen dar. Die optimale Anzahl an Produzenten und Verbrauchern wird unterschiedlich sein und muss durch Ausprobieren herausgefunden werden.

Auch hier gilt wieder, dass der Arbeitsaufwand beim Zeichnen vermutlich zu gering ist, um den extra Aufwand zu rechtfertigen. Dennoch sind die Gestaltung der Synchronisation und der Umgang mit dem Buffer eine gute Übung.

2.6 Buffer

Der Datenaustausch zwischen Produzenten und Verbraucher läuft über eine Warteschlange mit endlicher Größe, dem „Buffer“. Der Buffer muss groß genug sein, damit die Verbraucher immer etwas zu tun haben falls die Produzenten mal länger brauchen, aber nicht so groß, dass man erst alle Punkte generieren kann und anschließend verarbeitet. Das würde die Synchronisation hinfällig machen. Eintausend Elemente sollten passen. Der Buffer soll ein Fifo sein. Daten die zuerst rein kommen, kommen auch zuerst wieder raus.

Der Buffer:

- verwendet einen geeigneten Container der STL um die Daten zu speichern.
- realisiert einen Ringspeicher, sodass exzessives Kopieren vermieden wird (keine Vorhandene Implementierung, z.B. die in `Boost`)
- besitzt ein `push()` und `pop()` mit der intuitiven Bedeutung. In diesen Funktionen werden die Schreib- und Lese-Pointer verschoben und der Füllstand berechnet.
- soll unabhängig vom Datentyp sein → Template-Klasse wie `Producer` und `Consumer`.
- muss thread-sicher sein → Mutex und Synchronisation in den entsprechenden Funktionen.
- hat eine endliche Größe. Diese kann über `std::condition_variable` realisiert werden, orientiert euch an den Vorlesungsfolien (Vorlesung 2 Folien 54 ff.).
- lässt Threads, welche auf Daten im `pop()` warten, nur für eine begrenzte Zeit warten. Werft einen `std::runtime_error`, wenn zu lange keine neuen Daten mehr ankommen (z.B. weil alle Produzenten fertig sind). Die Basisklasse `consumer()` kann auf diesen Fall reagieren und die Verbraucher-Threads anhalten.

Hinweis: Die Arbeit mit `std::condition_variable` im Buffer ist sehr teuer und wird ein Flaschenhals sein. Um effizient mit mehreren Threads und kleinen Datenmengen zu arbeiten müsste man bei dem Farn Punkte im Paket durch den Buffer schieben und nicht einer nach dem anderen.

3 Allgemeines

- Die Deadline ist der 09. November 2021 um 23:55
- Der Code wird in Moodle als `.zip` Archiv abgegeben
- Es dürfen keine fertig kompilierten Programme vorliegen
- Es darf kein Compiler-Output vorhanden sein, wir werden nicht erst aufräumen
- Erstellt keine unnötigen Unterordner im Archiv
- Nennt das Projekt „HA“, vor allem die ausführbare Datei soll am Ende diesen Namen haben
- Der Code muss fehlerfrei kompilieren und starten mit dem gewohnten Ablauf im `build` Ordner:

  `cmake ..` → `make` → `./HA`

- Bei Fragen fragt jederzeit nach! Es gibt immer Fälle, welche wir nicht vorhergesehen haben
- Zeichnet euren eigenen Farn indem ihr die Parameter und Farbe verändert und legt ein `.png` der Abgabe bei