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

High Performance Computing für Maschinelle Intelligenz

Hausaufgabe 1

10. November 2020

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

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

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 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

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

bestimmt. Der Wert von i wird gemäß den Wahrscheinlichkeiten in der Tabelle vor dem Iterieren gewürfelt. Der Farn nimmt eine Fläche von $-2.2 < x < 2.7$ in x-Richtung und $0.0 < y < 10.0$ in y-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 jeweilige 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 eine Klasse 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 privated Variable `m_running`. Diese Information is notwendig um mehrfaches Starten zu verhindern.
- öffentlichen Funktionen zum Starten und Stoppen des Threads. Man kann auf zwei Arten Stoppen: Warten bis die Arbeit erledigt ist oder vorzeitig terminieren.
- einer rein virtuellen Funktion `step()` mit Zugriffsmodifikation `protected`, ohne Argumente und einem Boolean als Rückgabewert. Der Rückgabewert soll angeben, ob der Thread beendet werden soll weil die Aufgabe erledigt ist. Diese Funktion muss demnach von der Kind-Klasse implementiert werden und der `Worker` ist eine rein abstrakte Basisklasse.
- einer privaten Funktion `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 41).
- 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 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 dem `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()` implementiert. Darin wird ein neuer Datenpunkt per `produce(...)` erstellt und dem Buffer übergeben. Gegenfalls wird über den Rückgabewert der Thread in der Basisklasse gestoppt.
- `produce(...)` ist eine neue rein virtuelle Methode der Produzenten, 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 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 Funktionen System (1) gespeichert.

Zunächst müsst ihr euch für eine Darstellung eines Punktes entscheiden, denkbar sind:

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

Nutzt ggf. ein `typedef` um die Schreibarbeit zu reduzieren.

- Diese Klasse erbt von `Producer`, der Typ für das Template kann nun explizit angegeben werden (siehe oben)
- Der Konstruktor braucht wiederum die Referenz auf einen Buffer (mit gleichem 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 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.

Hinweis: Das erzeugen der Punkte ist keine rechenintensive Aufgabe. Die optimale Anzahl an Produzenten und Verbrauchern wird unterschiedlich sein und muss durch Ausprobieren herausgefunden werden.

2.4 Consumer

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

- Erstellt eine Template-Klasse `Consumer` mit dem Template-Typ `T`, welche von dem `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()` 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 allerdings die Verbraucher meistens nicht wissen wann die Produzenten fertig sind, wird das Ende zusätzlich über den Buffer gesteuert. Wenn er für eine gewisse Zeit leer bleibt wirft der Buffer einen `std::runtime_error`. Fangt diese Ausnahme in `step()` auf und stoppt den Thread, indem ein `true` zurückgegeben wird. Andernfalls steuert der Rückgabewert von `consume()` den Thread.
- `consume(...)` 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.

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 Speicher), während ein Mutex für das gesamte Bild den Vorteil mehrerer Verbraucher zunichte macht. Wählt ein grobes Gitter mit einem Mutex pro Zelle als Kompromiss.

Der Rest ist ähnlich zum Produzenten:

- 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, der Schreibzugriff muss synchronisiert werden. Genau ein Konsument muss das Bild am Ende speichern.
- `consume()` kann nicht entscheiden ob der Thread fertig ist (solange kein Fehler Auftritt, z.B. I/O-Error). Daher ist der Rückgabewert immer `false` und die Basisklasse soll sich um das Beenden 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.

2.6 Buffer

Der Datenaustausch zwischen Produzenten und Verbraucher läuft über eine Warteschlange mit endlicher Größe, dem „Buffer“. Der Buffer soll ein Fifo sein, Daten die zuerst rein kommen, kommen auch zuerst wieder raus.

- Verwendet einen geeigneten Container der STL um die Daten zu speichern. Exzessives Kopieren muss vermieden werden, realisiert am besten einen Ringpuffer (kein `Boost`).
- `push()` und `pop()` mit der intuitiven Bedeutung. In diesen Funktionen werden die Schreib- und Lese-Pointer verschoben und der Füllstand berechnet.
- Der Puffer soll unabhängig vom Datentyp sein → Template-Klasse wie die `Producer-` und `Consumer`-Basisklassen.
- Macht den Puffer thread-sicher → Mutex in den entsprechenden Funktionen.
- Die endliche Größe kann über `std::conditional_variable` realisiert werden. Orientiert euch an den Vorlesungsfolien.
- Threads, welche auf Daten im `pop()` warten, sollten nur für eine Begrenzte zeit warten. Werft einen `std::runtime_error`, wenn zu lange keine Daten mehr ankommen (weil die Produzenten fertig sind). Die Basisklasse `consumer()` kann auf diesen Fall reagieren und die Verbraucher-Threads anhalten.

Hinweis: Die Synchronisation im Buffer ist sehr teuer und wird ein Flaschenhals sein. Um effizient mit mehreren Threads zu arbeiten müsste man beim Farn Punkte im Paket durch den Buffer schieben und nicht einer nach dem anderen.

3 Allgemeines

- Der Code wird in Moodle als `.zip` Archiv abgegeben.
- Es dürfen keine fertig kompilierten Programme vorliegen.
- Nennt das Projekt „HA1“, 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` → `./HA1`

- Erstellt keine unnötigen Unterordner.
- Bei Fragen fragt jederzeit nach! Es gibt immer Fälle die wir nicht vorhergesehen haben.