TECHNISCHE UNIVERSITÄT DRESDEN

FAKULTÄT INFORMATIK
INSTITUT FÜR TECHNISCHE INFORMATIK
PROFESSUR FÜR RECHNERARCHITEKTUR
PROF. DR. WOLFGANG E. NAGEL

Komplexpraktikum Paralleles Rechnen - Aufgabe B

Daniel Körsten

Dresden, 3. November 2021

Inhaltsverzeichnis

1 Aufgabenbeschreibung

In dieser Aufgabe soll eine Thread-parallele Version von `Conway's Game-of-Life` in der Programmiersprache `C` programmiert werden. Das Game-of-Life ist ein vom Mathematiker John Horton Conway entworfenes Spiel [Gar70]. Anschließend soll die Simulation mit verschieden großen Feldgrößen und Compiler durchgeführt und verglichen werden.

1.1 Conway's Game-of-Life

Das Spiel besteht dabei aus einem Spielbrett mit einer festgelegten, möglichst großen, Anzahl an Zeilen und Spalten. Eine Zelle kann dabei entweder Tot oder Lebendig sein. Dieses Spielfeld wird mit einer zufälligen Anfangspopulation initialisiert.
Ein Sonderfall stellen die Ecken und Kanten des Feldes dar, da dort nach den Spielregeln das Verhalten nicht festgelegt ist. Die Aufgabenstellung gibt vor, dass Spielfeld Torus-förmig sein soll. Alles was das Spielfeld auf einer Seite verlässt, kommt auf der gegenüberliegenden Seite wieder herein.
Anschließend wird durch die Befolgung der Spielregeln die nächste Generation berechnet. Dafür betrachtet man jede Zelle und ihre 8 Nachbarn, um ihre Entwicklung zu berechnen. Es gelten folgende Spielregeln:

1. Eine lebende Zelle mit zwei oder drei Nachbarn überlebt in der Folgegeneration.
2. Eine lebende Zelle mit vier oder mehr Nachbarn stirbt an der Überpopulation. Bei weniger als zwei Nachbarn stirbt sie an Einsamkeit.
3. Jede tote Zelle mit genau drei Nachbarn wird in der nächsten Generation geboren.

Wichtig ist, dass die Folgegenration für alle Zellen berechnet wird und anschließend die aktuelle Generation ersetzt. Es ist also nicht möglich die nachfolgende Generation im Spielfeld der Aktuellen zu berechnen.

1.2 Besonderheiten der Aufgabenstellung

Die Aufgabenstellung gibt vor, dass die Parallelisierung mittels `OpenMP` erfolgen soll. `OpenMP` ist eine API, welche es ermöglicht, Schleifen mithilfe von Threads zu parallelisieren [ope18]. Es eignet sich hervorragend für `Shared-Memory Systeme`, also Systeme, bei denen mehrere Threads auf einen gemeinsamen Hauptspeicher zugreifen.
Weitere Besonderheiten sind:

- Die Simulation soll variabel mit Feldgrößen von 128×128 bis 32768×32768 und 1 bis 32 Threads erfolgen.
- Das OpenMP Schedulingverfahren soll hinsichtlich des Einflusses auf die Ausführungszeit untersucht werden.
- Das Programm soll mit dem dem `GCC` und `ICC` kompiliert und anschließend getestet werden.

2 Implementierung

Zuerst habe ich mich mit der Abstraktion des Feldes in `C` beschäftigt. Meine Idee ist die Allokierung eines Speicherbereichs der Größe `columns * rows * sizeof(u_int8_t)` mittels der Funktion `malloc()`. Innerhalb des Speicherbereichs kann man sich nun frei bewegen. Dabei verwendet man die `columns` als Offset um an die entsprechende Stelle zu springen. Beispiel: Möchte man auf die Zweite Zelle in Zweiten Zeile (da die Nummerierung typischerweise bei 0 beginnt, also das erste Element) zugreifen, würde man das `columns + 1` Byte innerhalb des Speicherbereichs verwenden.

2.1 Vorbereitung des Codes für eine parallele Ausführung

Für die Berechnung der nächsten Generation muss nun jede Zelle einzeln betrachtet werden und ihr Zustand in der nächsten Generation gemäß den Spielregeln berechnet werden. Ein möglicher Ansatz ist über jede Zeile und anschließend jede Spalte zu iterieren. Realisieren lässt sich das über zwei geschachtelte `for`-Schleifen. Das Ergebnis dieser Berechnung muss in einer zweiten Matrix gespeichert werden um die Berechnungen der Nachbarzellen nicht zu verfälschen.
Dieser Ansatz bietet den Vorteil, dass er mit OpenMP relativ einfach parallelisiert werden kann, denn die Berechnung jeder einzelnen Zelle ist unabhängig von den Berechnungen anderer Zellen.
Besonderes Augenmerk muss man jedoch auf die Kanten und Ecken legen. Diese sollen, laut Aufgabenstellung, mit `periodic boundary conditions` implementiert werden. Jedoch kann auch hier OpenMP zur Parallelisierung der Kanten verwendet werden.

2.2 Ein und Ausgabe

Da die Messung später in verschiedenen Feldgrößen durchgeführt wird, habe ich mich für den Einsatz von `getopt` entschieden. Es ermöglicht die Anzahl der Schleifendurchläufe, die Feldgröße und eine optionale Fortschrittsanzeige über Argumente beim Programmstart einzustellen. Ebenso lassen sich Threadanzahl und OpenMP Schedulingverfahren einstellen.

2.3 Probleme und ihre Lösung

Die Erzeugung des 2D-Arrays soll dynamisch erfolgen, damit man, ohne Anpassung des Programmcodes, die Feldgröße festlegen kann. Gemäß der Aufgabenstellung umfasst das größte Feld 32768×32768 Zellen. Damit ist das Array zu groß für den Stack des Programms. Mein Lösungsansatz war die Allokation von Speicher mittels `malloc` und `double pointers`: Ein erstes Array wurde mit Pointern gefüllt, die jeweils wiederum auf die einzelnen Zeilen verweisen, welche ebenfalls mit `malloc` allokiert wurden. Später habe ich die `double pointers` durch ein einzelnes `malloc` ersetzt, indem die Anzahl von Spalten als Offset dient, um sich im 2D-Array zu bewegen. Dadurch vermeidet man einen Speicherzugriff und folglich eine Adressübersetzung bei jeder Datenabfrage und -manipulation.
Nach der Parallelisierung mit OpenMP änderte sich jedoch nichts an der Ausführungszeit, was eindeutig den Erwartungen widersprach. Durch Experimentieren fand ich heraus, dass die Parallelisierung der Funktion, welche das Array mit Zufallswerten füllt, die Performance massiv beeinflusst. Die Funktion `rand` ist nicht für die parallele Ausführung geeignet. Deshalb habe ich mich hier einfach für die sequentielle Ausführung entschieden.

2.4 Optimierung des Codes

Eine Möglichkeit zur Einsparung von Rechenressourcen ist, den Speicherverbrauch des Programms zu reduzieren. So verwendete ich für die Zellen der Felder den Datentyp `u_int8_t` statt `int`. Dadurch reduziert sich der Speicherverbrauch jeder Zelle von 4 auf 1 Byte. Bei einem Feld der Größe 32768×32768 entspricht dies einer stattlichen Einsparung von über 3 GiB.
Die Verwendung von von bitweisen Operatoren in den Berechnung bringen ebenfalls einen geringen Geschwindigkeitsvorteil, da bitweise Operationen in der Schaltungstechnik die Einfachsten darstellen. Alle höheren Operationen lassen sich auf sie zurückführen.

Literatur

[Gar70] GARDNER, Martin. *MATHEMATICAL GAMES - The fantastic combinations of John Conway's new solitaire game "life"*. https://web.stanford.edu/class/sts145/Library/life.pdf. 1970

[ope18] *OpenMP Application Programming Interface 5.0*. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf. 2018