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, 18. Dezember 2021

Inhaltsverzeichnis

1 Aufgabenbeschreibung

In dieser Aufgabe soll eine Thread-parallele Version von `Conway's Game-of-Life` in der Programmiersprache `C` implementiert werden.
Anschließend soll die Simulation mit verschieden großen Feldgrößen und Compiler durchgeführt und verglichen werden.

1.1 Conway's Game-of-Life

Das Game-of-Life ist ein vom Mathematiker John Horton Conway entworfenes Simulationsspiel [Gar70]. Es basiert auf einem zellulären Automaten. Häufig handelt es sich um ein Zweidimensionales Spielfeld, jedoch ist auch eine Dreidimensionale Simulation möglich.
Das Spiel besteht dabei aus einem Feld 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)` durch die C-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. Praktischerweise entspricht eine Zelle im Feld einem Byte im Speicher.
Beispiel: Möchte man auf die Zweite Zelle in der Zweiten Zeile (da die Nummerierung typischerweise bei 0 beginnt, also das erste Element) zugreifen, würde man das `columns + 1` Byte innerhalb des Speicherbereichs verwenden.
Der Datentyp `u_int8_t` benötigt dabei nur Ein Byte pro Zelle und ist für die Speicherung mehr als ausreichend, da ich nur den Zustand 0 - Zelle tot und 1 - Zelle lebendig speichern muss. Ein Byte ist typischerweise die kleinste adressierbare Einheit im Speicher. Das ist auch der Grund, warum kein noch kleiner Datentyp möglich ist.

Um zu Berücksichtigen, dass die Folgegeneration immer der aktuelle Generation ersetzt, allokiere ich einen zweiten Speicherbereich gleicher Größe. Vor dem Beginn einer neuen Berechnung, vertausche ich die beide Speicherbereiche, was dazu führt, dass die im vorhergehenden Schritt berechnete Folgegeneration zur aktuellen Generation wird und eine neue Generation berechnet werden kann.

2.1 Daten initialisieren

Gemäß den Startbedingungen muss nur eines der beiden Spielfelder mit Zufallswerten initialisiert werden. Um den Code möglichst einfach und effizient zu halten, verwende ich eine `for`-Schleife zur Iteration über jede Zelle des Arrays.
Für die Dateninitialisierung jeder Zelle mit Null oder Eins, habe ich mich für Pseudo-Zufallszahlengenerator `rand_r()` entschieden. Dieser ist, im Vergleich zu z.B. `rand()` Thread-sicher und kann Thread-parallel ausgeführt werden.
Für die eigentliche Parallelisierung verwende ich die `OpenMP` Direktive:

```
#pragma omp parallel for schedule(runtime)
```

Diese bewirkt, dass der Code innerhalb der Schleife parallel ausgeführt wird. `OpenMP` erzeugt beim betreten zusätzliche *slave* Threads. Jeder bekommt einen Teil der Arbeit zugewiesen und führt diesen unabhängig von den anderen Threads aus. Wenn alle Threads ihre Arbeit erledigt haben, der parallel auszuführende Code also abgearbeitet wurde, fährt der *master* Thread mit der seriellen Ausführung fort, bis er die nächste Direktive erreicht.
Über die Umgebungsvariable `OMP_THREAD_LIMIT` kann ein Thread Limit gesetzt werden. `OpenMP` verwendet dann maximal so viele Threads, wie angegeben. Wird die Umgebungsvariable nicht gesetzt, verwendet `OpenMP` eine optimale Anzahl an Threads. Typischerweise entspricht die der Zahl der Anzahl der Hardware-Threads auf dem System.
Durch `schedule(runtime)` ist es später möglich, über die Umgebungsvariable `OMP_SCHEDULE` das Schedulingverfahren zu wählen.

Bei der parallelen Ausführung ist darauf zu achten, dass jeder Thread mit einem unterschiedlich *seed* den Pseudo-Zufallszahlengenerator `rand_r()` startet. Um dieses Problem zu lösen, entschied ich mich, die Threads mit der `OpenMP` Direktive

```
#pragma omp parallel
```

vor der *seed* Generierung zu erzeugen. Dadurch werden Zwei Probleme gelöst:

1. Jeder Thread arbeitet auf seiner eigenen *seed* Variable. Dadurch wird verhindert, dass Threads auf der gleichen Variable arbeiten und folglich ein Flaschenhals entsteht.
2. Die *seed* Variablen können unterschiedliche Werte haben, was wiederum die Entropie des initialisierten Spielfeldes erhöht.

Die *seed* Variable ergibt sich bei mir aus der aktuellen Zeit in Sekunden und der Thread ID. Da bei jeder Ausführung die Zeit als auch die Thread ID variiert, erhält jeder Thread einen zufälligen *seed* mit geringem Rechenaufwand.

Anmerkung zu rand_r():
`rand_r()` wird in den `Linux Man Pages` als schwacher Pseudo-Zufallszahlengenerator geführt [lmp10]. Das soll an dieser Stelle keine Relevanz haben, da der Spielverlauf und Rechenaufwand nicht von der Güte des Zufallsgenerators abhängt.
Vielleicht Bild, wie so ein initialisiertes Feld aussieht?

2.2 Berechnung der nächsten Generation

Die Berechnung der nächsten Generation erfolgt mithilfe beider Spielfelder. Die Funktion `calculate_next_gen()` erhält einen Pointer auf das Array mit der aktuellen Generation `*state_old` und einen auf das Array der Folgegeneration `*state`.
Bei jedem Simulationsschritt werden Pointer getauscht und die Funktion erneut aufgerufen. Damit wird die Forderung der Aufgabenstellung nach *double buffering* erfüllt, sprich die Folgegeneration in einem separatem Spielfeld berechnet.
Da es sich um ein Torus-förmiges Spielfeld handelt, benötigen die Kanten und Ecken eine separate Behandlung. Den Großteil stellt jedoch die Berechnung des inneren Feldes dar. Gleichzeitig unterscheiden sich die Schritte nur unwesentlich.

Der Zustand der Zelle in der nächsten Generation wird über die Spielregeln bestimmt und ist abhängig vom aktuellen Zustand der Zelle und ihren Acht Nachbarn. Da der Zustand mit Null (tot) oder Eins (lebendig) repräsentiert wird, kann die Zahl der Nachbarzellen aufsummiert werden. Die Summe entspricht dabei der Zahl lebender Nachbarn.
An dieser Stelle könnte mithilfe einer *if*-Verzweigung der Folgezustand entschieden werden. Allerdings entschied ich mich für die Verwendung von bitweisen Operatoren. Es handelt sich dabei aus schaltungstechnischer Sicht um die einfachsten Operationen auf den einzelnen Bits.
Der Grund liegt darin, dass die CPU bei *if*-Verzweigungen ihre Sprungvorhersage verwendet um die Pipeline möglichst sinnvoll auszulasten. Selbst mit einer guten Vorhersage werden falsche Entscheidungen

getroffen, die dann rückgängig gemacht werden müssen. Gleichzeit ist die CPU sehr schnell im abarbeiten von arithmetischen Operationen.
Daraus ergibt sich, bei der Verwendung bitweiser Operatoren, ein Performance Vorteil.

Im ersten Schritt werden alle Zellen berechnet, die nicht Teil einer Kante sind. Dafür verwende ich zwei geschachtelte `for`-Schleifen:

```
#pragma omp parallel for schedule(runtime)
for (int i = 1; i < rows - 1; i++) {
  for (int j = 1; j < columns - 1; j++) {
    //count up the neighbours
    u_int8_t sum_of_8 = state_old[(i - 1) * columns + (j - 1)] +
                        state_old[(i - 1) * columns + j] +
                        state_old[(i - 1) * columns + (j + 1)] +
                        state_old[i * columns + (j - 1)] +
                        state_old[i * columns + (j + 1)] +
                        state_old[(i + 1) * columns + (j - 1)] +
                        state_old[(i + 1) * columns + j] +
                        state_old[(i + 1) * columns + (j + 1)];
    state[i * columns + j] = (sum_of_8 == 3) | ((sum_of_8 == 2) & state_old[i *
    columns + j]);
  }
}

```

Listing 1: Berechnung der inneren Zellen

Die Erste Schleife iteriert dabei über jede Zeile und in jeder Zeile geht die Zweite durch jede Zelle. Ich habe dabei, wie schon bei der Daten Initialisierung, die `OpenMP` Direktive:

```
#pragma omp parallel for schedule(runtime)
```

verwendet.
Dabei wird jedoch nur die äußere Schleife parallelisiert. Das bewirkt, dass die Zeilen jeweils parallel berechnet, jedoch nicht aufgeteilt werden. `OpenMP` ist in der Lage mit `collapse(2)` zwei geschachtelte Schleifen zu parallelisieren, indem es daraus eine große Schleife erzeugt. Diese große Schleife wird dann in `chunks` zerlegt und den einzelnen Threads zur Bearbeitung zugewiesen. In meinen Tests führte dies zu einer enormen Verschlechterung der Performance, weswegen ich es nicht verwendet habe.

Die Gründe dafür können vielfältig sein. Ein Grund könnte der Fakt sein, dass `OpenMP` die Schleife ungünstig zerlegt.
Für die Berechnung einer Zelle benötigt man die Zelle selbst und ihre Acht Nachbarn. Geht man eine Zelle weiter, benötigt man Sechs der Neun Zellen aus dem vorherigem Schleifendurchlauf. Die Brechungen überlappen also. Verwendet man für die Berechnung einer Zeile einen Kern (= ein Thread; deaktiviertes Hyperthreading vorausgesetzt), kann man vom Cache profitieren. Jeder Kern arbeitet dann möglichst auf den Daten, die er schon einmal angefasst hat.

Zum anderen kann der Compiler, bei meiner Implementierung, die innere Schleife modifizieren und so eventuelle Optimierungen vornehmen. ->(kein Vektor (überprüft))

2.3 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 Thread Anzahl und OpenMP Schedulingverfahren einstellen.

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

[lmp10] *Linux Man Pages - rand*.
`https://linux.die.net/man/3/rand`. 2010

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