TECHNISCHE UNIVERSITÄT DRESDEN

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

Komplexpraktikum Paralleles Rechnen - Aufgabe C

Daniel Körsten

Dresden, 25. Januar 2022

Inhaltsverzeichnis

1 Aufgabenbeschreibung

In dieser Aufgabe soll eine SIMD-parallele Version von `Conway's Game-of-Life` in der Programmiersprache `C` implementiert werden.
Anschließend soll die Simulation mit verschieden großen Feldgrößen und Compilern 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 das 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 Folgegeneration 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` Compiler-Direktiven erfolgen soll. `OpenMP` ist eine API, die es ermöglicht, Schleifen mithilfe von Threads zu parallelisieren [ope18], was in Aufgabe B thematisiert wurde. Es eignet sich hervorragend für `Shared-Memory Systeme`, also Systeme, bei denen mehrere Threads auf einen gemeinsamen Hauptspeicher zugreifen.
`OpenMP` bietet auch die Möglichkeit, explizit SIMD Instruktionen für die Bearbeitung von Schleifen zu verwenden. Das soll in dieser Aufgabe bearbeitet werden.
Weitere Besonderheiten sind:

- Die Simulation soll variabel mit Feldgrößen von 128×128 bis 32768×32768 erfolgen.
- Es sollen Messungen mit aktiviertem und inaktivem `OpenMP` durchgeführt werden.
- Das Programm soll mit 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 kleinerer Datentyp möglich ist.

Um zu berücksichtigen, dass die Folgegeneration immer die 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 Initialisierung

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, verwende ich den Pseudo-Zufallszahlengenerator `rand()`. Den `seed` setzte ich mit der `time()` Funktion auf die aktuelle Uhrzeit. Damit beginnt das Spiel bei jedem Programmstart mit einer zufälligen Generation.

Der Zufallsgenerator kann nicht mit einer einfachen `OpenMP` Direktive SIMD-parallel ausgeführt werden. Allerdings habe ich mich aus Interesse damit auseinander gesetzt und bin über die `Xorshift` Generatoren gestoßen, genauer `Xorshift128+` [Vig17]. Dieser verwendet, wie der Name vermuten lässt, Addition statt Multiplikation, die in der Regel weniger Rechenintensiv ist.
Die von mir verwendete Implementierung ist auf GitHub verfügbar.

Zusammengesetzt ergibt sich daraus folgender Code:

```
1 void field_initializer(u_int8_t *state) {
2   //fills fields with random numbers 0 = dead, 1 = alive
3   #pragma omp parallel
4   {
5     unsigned tid = pthread_self();
6     unsigned seed = time(0) + tid;
7     #pragma omp parallel for schedule(runtime)
```

```
8       for (int i = 0; i < columns * rows; i++) {
9         state[i] = rand_r(&seed) % 2;
10      }
11    }
12    return;
13  }
```

Listing 1: Daten Initialisierung

Abbildung 1: Generiertes Spielfeld der Größe: 64×64

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.

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 die 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.

```
for (int i = 0; i < repetitions; i++) {
  calculate_next_gen(state_out, state_in);
  state_tmp = state_in;
  state_in = state_out;
  state_out = state_tmp;
}
```

Listing 2: Vertauschen der Pointer vor jedem Funktionsaufruf (vereinfacht)

Da es sich um ein Torus-förmiges Spielfeld handelt, benötigen die Kanten und Ecken eine separate Behandlung. Diese unterschiedet sich nur unwesentlich von der Berechnung des inneren Feldes.

2.2.1 Inneres Feld

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. Zusätzlich 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++) {
```

```
3   for (int j = 1; j < columns - 1; j++) {
4     //count up the neighbours
5     u_int8_t sum_of_8 = state_old[(i - 1) * columns + (j - 1)] +
6                         state_old[(i - 1) * columns + j] +
7                         state_old[(i - 1) * columns + (j + 1)] +
8                         state_old[i * columns + (j - 1)] +
9                         state_old[i * columns + (j + 1)] +
10                        state_old[(i + 1) * columns + (j - 1)] +
11                        state_old[(i + 1) * columns + j] +
12                        state_old[(i + 1) * columns + (j + 1)];
13    state[i * columns + j] = (sum_of_8 == 3) | ((sum_of_8 == 2) & state_old[i *
      columns + j]);
14  }
15 }
```

Listing 3: Berechnung der inneren Zellen

Die erste Schleife iteriert dabei über jede Zeile und in jeder Zeile die Zweite durch jede Zelle. Ich habe dabei, wie schon bei der Dateninitialisierung, 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` wäre 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 (gleichbedeutend mit einem Thread; deaktiviertes Simultaneous Multithreading vorausgesetzt), kann man vom Cache profitieren. Jeder Kern arbeitet dann möglichst auf den Daten, die er schon einmal geladen hat.

Zum anderen kann der Compiler, bei meiner Implementierung, die innere Schleife modifizieren und so eventuelle Optimierungen vornehmen. Ich habe mir deshalb mit `objdump` den Quellcode anzeigen lassen, konnte jedoch keine `SIMD`-Instruktionen finden.

2.2.2 Kanten

Wie bereits erwähnt, unterscheidet sich die Art und Weise der Berechnung der Kanten nur unwesentlich von der des inneren Feldes. Da die Kanten jeweils nur aus einer Zeile bzw. Spalte bestehen, wird nur eine `for`-Schleife benötigt. Außerdem muss in der Berechnung beachtet werden, dass Felder von der

gegenüberliegenden Seite benötigt werden. Auch hier wurden die Kanten wieder mit

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

parallelisiert.

```
void calculate_top(u_int8_t *state, u_int8_t *state_old) {
  #pragma omp parallel for schedule(runtime)
  for (int i = 1; i < columns - 1; i++) {
    u_int8_t sum_of_t_edge = state_old[i - 1] +
                             state_old[i + 1] +
                             state_old[2 * columns + (i - 1)] +
                             state_old[2 * columns + i] +
                             state_old[2 * columns + (i + 1)] +
                             state_old[(rows - 1) * columns + i] +
                             state_old[(rows - 1) * columns + i + 1] +
                             state_old[(rows - 1) * columns + i - 1];
    state[i] = (sum_of_t_edge == 3) | ((sum_of_t_edge == 2) & state_old[i]);
  }
}
```

Listing 4: Berechnung der obersten Zeile

2.2.3 Ecken

Bei den Ecken ist keine Parallelisierung möglich und auch nicht notwendig, da vier Ecken bei einem Feld mit mehr als 16.000 Felder nicht ins Gewicht fallen.
Die Berechnung basiert wieder auf der vorher aufgeführten Methode.

```
void calculate_corner(u_int8_t *state, u_int8_t *state_old) {
  u_int8_t corner_sum;
  // top left
  corner_sum = state_old[1] +
  state_old[columns] +
  state_old[columns + 1] +
  state_old[(rows - 1) * columns] +
  state_old[(rows - 1) * columns + 1] +
  state_old[columns - 1] +
  state_old[2 * columns - 1] +
  state_old[rows * columns - 1];
  state[0] = (corner_sum == 3) | ((corner_sum == 2) & state_old[0]);
}
```

Listing 5: Berechnung der Ecke oben links

2.3 Zeitmessung

Für die Zeitmessung habe ich auf die `clock()` und `omp_get_wtime()` Funktion zurückgegriffen.

Die `clock()` Funktion misst dabei die CPU Zeit des Prozesses und nicht die reale vergangene Zeit (*wall-clock time*). Das Ergebnis ist also die aufaddierte Zeit aller Threads in der Funktion.

Die `omp_get_wtime()` Funktion gibt hingegen die *wall-clock time* zurück; unabhängig von der Anzahl der Threads.

Ich entschied mich dafür, nicht nur die Funktion `calculate_next_gen()` zu messen, sonder auch die `field_initializer()` Funktion. Dadurch lässt sich später eine genauere Aussage über die Parallelisierbarkeit des Problems treffen. Für die Berechnung der Ausführungszeit wird vor und nach Ausführung der zu untersuchenden Funktion `clock()` und `omp_get_wtime()` aufgerufen. Die Differenz aus den beiden Momentaufnahmen entspricht der jeweiligen Zeit.

```
double time_calc = 0;
double omp_calc = 0;
for (int i = 0; i < repetitions; i++) {
  t = clock();
  t_omp = omp_get_wtime();
  // function call
  t_omp = omp_get_wtime() - t_omp;
  t = clock() - t;
  omp_calc += t_omp;
  time_calc += ((double) t) / CLOCKS_PER_SEC;
}
printf("Calculation took %f seconds to execute (all threads added).\n", time_calc);
printf("Calculation took %f seconds to execute (real time).\n", omp_calc);
```

Listing 6: Berechnung der Ausführungszeit eines *function calls*

Bei der Darstellung der Testergebnisse 4 beziehe ich mich nur auf die Ergebnisse der `omp_get_wtime()` Funktion, da die Ausführungszeiten (*wall-clock time*) verglichen werden sollten.

2.4 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 lässt sich das Ergebnis über *pbm*-Files visualisieren.
Alle Funktionen sowie die Syntax lassen sich über den Parameter `--help` ausgeben.

3 Zeitmessung auf Taurus

3.1 Testumgebung

Alle Messungen wurden auf dem Hochleistungsrechner Taurus der TU Dresden durchgeführt.
Verwendet habe ich die Romeo-Partition, die auf AMD Rome EPYC 7702 Prozessoren basiert [Mar21]. Hier reservierte ich für die Messungen einen kompletten Node, um Schwankungen durch andere Prozesse auf dem Knoten auszuschließen. Ein Node ist mit 512 GB RAM ausgestattet. Als Betriebssystem kommt Centos 7 zum Einsatz.

Vor Beginn der Messung muss noch die Topologie des unterliegenden Systems betrachtet werden. Für die optimale Performance, sollten die größten Vektorregister zum Einsatz kommen. Bei den EPYC Prozessoren entspricht das AVX2. Die Breite der Register liegt hier bei 256 Bit.
Außerdem wird FMA unterstützt. FMA steht für Fused-multiply-add und steigert die Leistung durch verbesserte Ausnutzung von Registern und einen kompakteren Maschinencode. Das wird durch das Zusammenfassen einer Addition und Multiplikation zu einem Befehl erreicht.
Wie in der Aufgabenstellung gefordert, kompilierte ich das Programm mit dem `GCC` (**G**NU **C**ompiler **C**ollection, Version 11.2) und dem `ICC` (**I**ntel **C**ompiler **C**ollection, Version 19.0.5.281); jeweils mit den Compiler-Flags:

- `O3` - Optimierungsflag des Compilers
- `mavx2` - Verwendung von AVX2
- `fma` - Verwendung von FMA
- `fopenmp` - Verwendung nur für Tests mit OpenMP

Da für die Messungen nur ein Core verwendet werden soll, ist es nicht notwendig Threads an bestimmte Cores zu pinnen, wie das noch bei Aufgabe B erforderlich war.

Bemerkung zur `O3` Flag:
Die Optimierungsflag teilt dem Compiler mit, dass er die Performance auf Kosten der Programmgröße und Kompilationszeit erhöht. Das schließt SIMD Instruktionen ein. Ohne diese Flag hat der `GCC` auch mit aktiviertem `OpenMP` keine SIMD Instruktionen verwendet, was ich mit `objdump` verifiziert habe. Da die Aufgabenstellung vorgibt, man solle aktiviertes und deaktiviertes `OpenMP` vergleichen, habe ich die Optimierungsflag bei beiden aktiviert gelassen.
Das führt zu dem Ergebnis, dass der `GCC` nur SIMD Instruktionen in Kombination mit der Optimierungsflag verwendet.
Der Intel Compiler verwendet hingegen SIMD Instruktionen nur, wenn die `fopenmp` Flag gesetzt wurde.

3.2 Testmethode

Die Tests wurden automatisiert mit einem `sbatch`-Skript ausgeführt. Um Schwankungen auszugleichen, wurde jede Messung 20 mal wiederholt.

Dabei ist zu beachten, dass die Ausführungszeit (logischerweise) mit der Feldgröße linear ansteigt. Interessanter für das Praktikum ist jedoch das Verhalten bei der Verwendung von SIMD Instruktionen. Deshalb passte ich die Anzahl der Wiederholungen an die Feldgröße an. Die genauen Details lassen sich den Tabellen 1 und 2 entnehmen.
Die Anzahl von Simulationsschritten ist auf 100 für alle Feldgrößen festgelegt. Ich habe diese Größe gewählt, da so ein Vergleich zwischen verschiedenen Feldgrößen möglich wird, um zu entschieden, wie sich die Ausführungszeit in Abhängigkeit dieser verändert.
Für noch genauere Ergebnisse wären mehr Simulationsschritte nötig gewesen. Bei einer Feldgröße von 128 mit mehr als 100.000 Wiederholungen befindet sich die Ausführungszeit immer noch im Millisekunden Bereich. Aufgrund begrenzter CPU Zeit ist eine so hohe Anzahl an Simulationsschritten nicht möglich.

Aus den 20 Wiederholungen pro Messung habe ich den Mittelwert gebildet und Logarithmische Diagramme erzeugt. Diese Darstellung bietet sich aufgrund der exponentiell ansteigenden Feldgröße an. Ansonsten wären die Werte der Feldgrößen unter 8192 im Diagramm nicht unterscheidbar.

4 Testergebnisse

In diesem Abschnitt habe ich die Messwerte aus den Tabellen A visualisiert.

4.1 GCC kompilierte Version

Zuerst möchte ich auf die `GCC` kompilierte Version des Spiels eingehen.

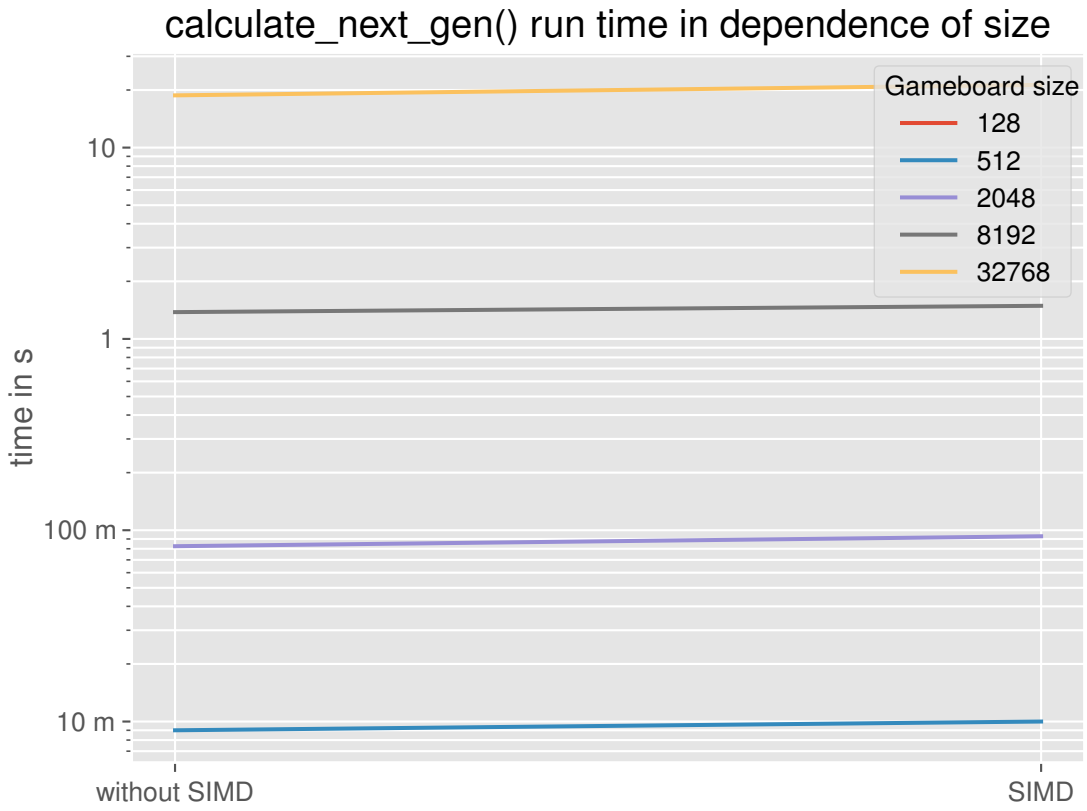

Abbildung 2: Logarithmische Darstellung der Ausführungszeit der Funktion `calculate_next_gen()`, kompiliert mit `GCC`.

Die gezeigte Grafik berücksichtigt nur die Ausführung der Funktion `calculate_next_gen()`. Da sich sowohl die Feldgröße, als auch die Anzahl der Wiederholungen exponentiell erhöht, bietet sich eine doppellogarithmische Darstellung an.
Man erkennt eindeutig die zu erwartende Tendenz: Mehr Threads bewirken eine Verkürzung der Laufzeit. Allerdings lässt sich bei den Feldgrößen 128 deutlich und 512 in abgeschwächter Form erkennen, dass ab einer gewissen Thread Anzahl kein Speedup mehr erreicht werden kann. Schlimmer noch: Bei der Größe 128 steigt die Ausführungszeit mit 32 Threads wieder an.
Eine Begründung ist, dass die einzelnen Schleifendurchläufe dieser Feldgrößen sehr schnell abgeschlossen sind. Mit steigender Anzahl an Threads steigt auch der Overhead durch Thread spawning, Synchronisation und Zuteilung der Arbeit. Ab einem gewissen Punkt ist dieser Overhead im Vergleich zu dem

zu lösenden Problem nicht mehr vernachlässigbar klein und wirkt sich negativ auf den Speedup aus.
Wie in allen folgenden Grafiken zeichnet sich das Bild ab, dass eine Verdopplung der Threads, aus den genannten Gründen, nicht zur Halbierung der Laufzeit führt. Das ist eine theoretische Annahme, die in der Praxis nie erreicht werden kann.

Ein anderes Bild zeichnet sich jedoch beim Ausführen der Funktion `field_initializer()` ab.

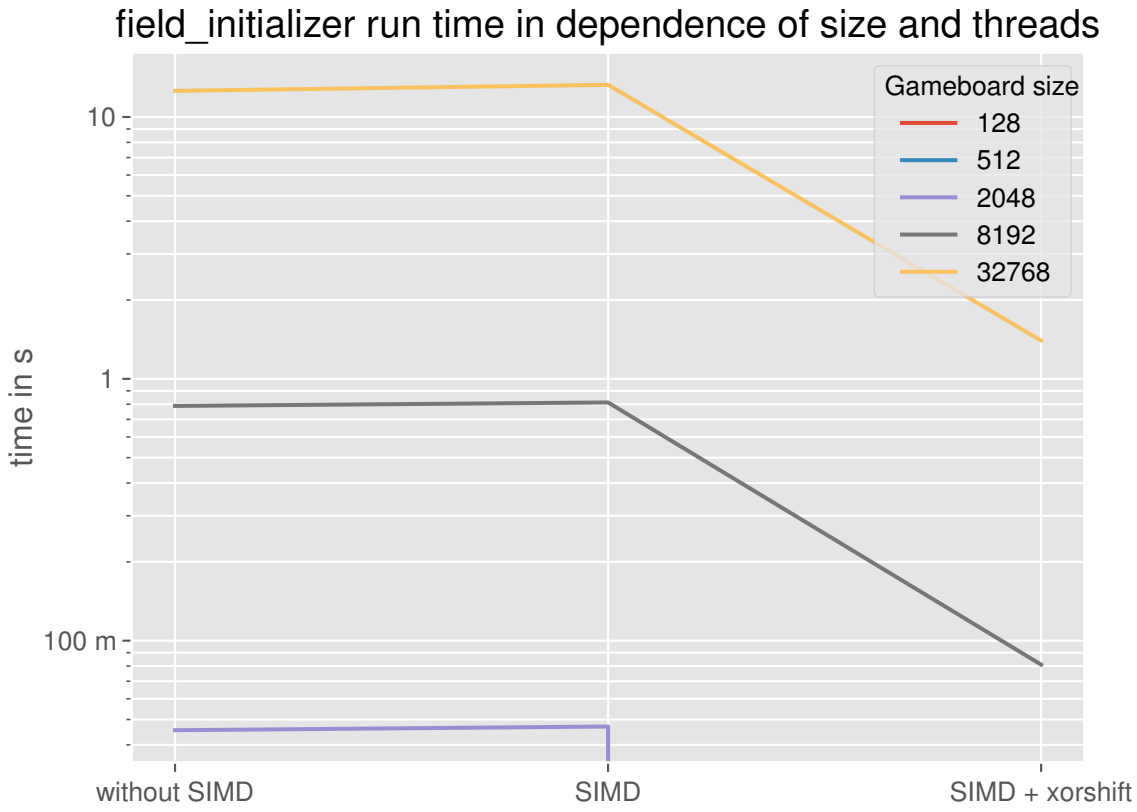

Abbildung 3: Logarithmische Darstellung der Ausführungszeit der Funktion `field_initializer()`, kompiliert mit `GCC`.

Erwartungsgemäß benötigt ein großes Feld mehr Zeit zum Initialisieren, als ein kleines Feld. Betrachtet man, dass ein 32768 Spielfeld 16 mal so groß ist wie 8192 und nach den Daten auch etwa 16 mal so viel Zeit in Anspruch nimmt, sind die Zeiten sehr plausibel und skalieren etwa linear mit der Feldgröße. Ausnahmen bilden hier wieder die kleinen Feldgrößen, bei denen die Werte auch extrem schwanken.
Erklären lässt sich das mit der äußerst kurzen Ausführungszeit von teilweise deutlich unter 10 ms bei einer Feldgröße von 128. Vermutlich ist hier wieder der Overhead im Verhältnis zum zu lösenden Problem deutlich größer. Bei einer großen Problemgröße benötigt die Initialisierung etwa gleich viel Zeit unabhängig von der Zahl der Threads.
Die Parallelisierung bietet an dieser Stelle keinen nennenswerten Geschwindigkeitsvorteil.

4.2 ICC kompilierte Version

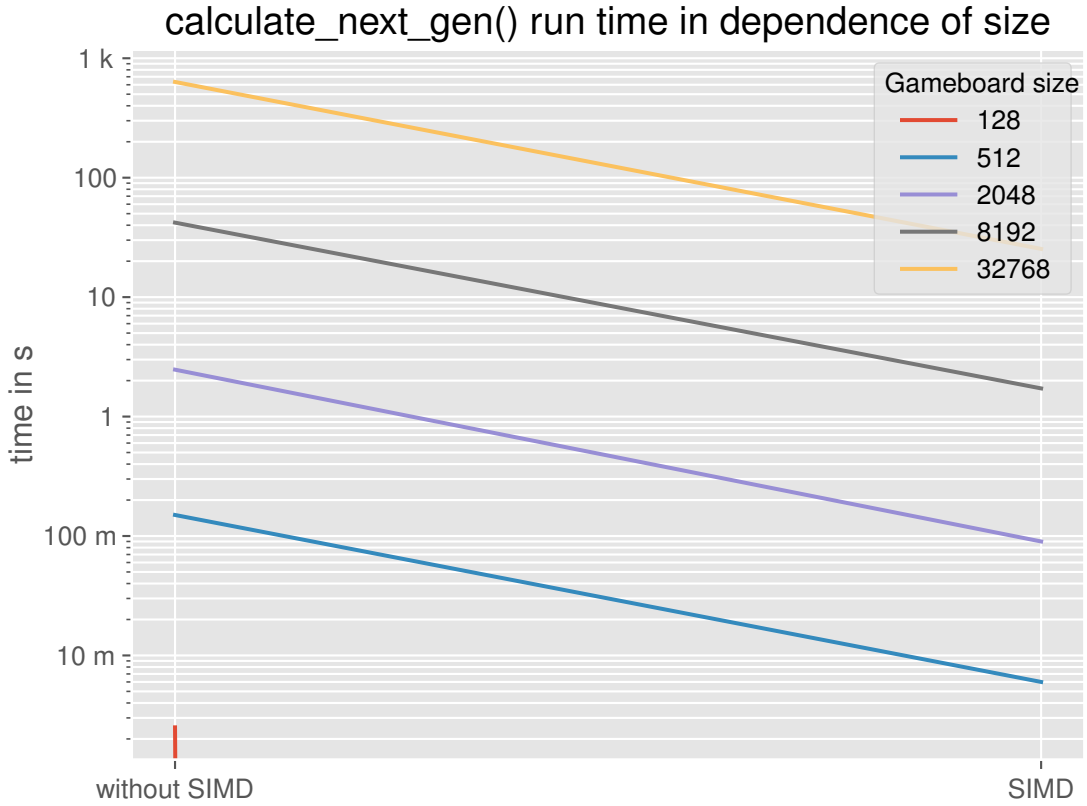

Abbildung 4: Logarithmische Darstellung der Ausführungszeit der Funktion `calculate_next_gen()`, kompiliert mit `ICC`.

Ähnlich zur `GCC` kompilierten Version des Spiels sind hier die selben Effekte erkennbar. Mehr Threads verkürzen generell die Ausführungszeit und bei kleinen Feldgrößen verringert sich zunehmend der Speedup.
Auffällig ist, dass die Version des Intel Compilers bei der jeweilig selben Anzahl von Threads und Feldgröße nur ca. 25% der Zeit benötigt, also zu jeder Zeit etwa vier mal so schnell ist. Der Intel Compiler produziert folglich besser optimierten Code.
Ich habe mir mittels `objdump` den disassemblierten Code beider Binaries angeschaut, da ich vermutete, dass der Intel Compiler möglicherweise `SIMD`-Instruktionen verwendet. Das ist jedoch nicht der Fall.

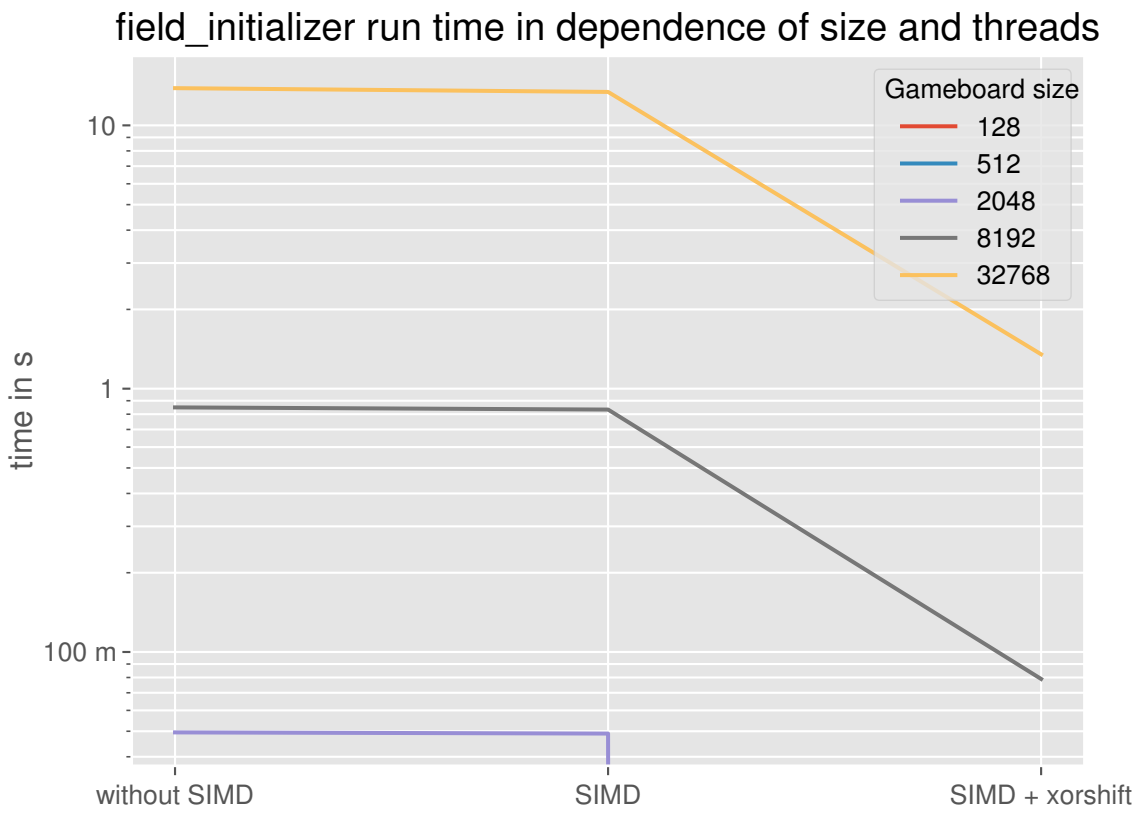

Abbildung 5: Logarithmische Darstellung der Ausführungszeit der Funktion `field_initializer()`, kompiliert mit `ICC`.

Bei der Dateninitialisierung zeigt sich beim Intel Compiler ein etwas anderes Bild. Im Allgemeinen schwanken die Werte der `ICC` Version weniger stark.
Bei Feldgrößen ab 2048 ist er ebenfalls schneller als die Version des `GCC` Compilers. Bei den Werten darunter ist ein Vergleich aufgrund der Schwankungen schwierig. Tendenziell lässt sich wieder ein Anstieg der Zeit mit steigender Zahl an Threads verbuchen.

A Tabellen

Compiler	Size	#Threads	Repetitions	Initialization in s	Calculation in s
GCC	128	1	100 000	0.0003	30.7874
GCC	128	2	100 000	0.0007	17.171
GCC	128	4	100 000	0.0011	9.5063
GCC	128	8	100 000	0.0036	6.8678
GCC	128	16	100 000	0.0032	5.4929
GCC	128	32	100 000	0.0038	5.8989
GCC	512	1	10 000	0.0039	48.383
GCC	512	2	10 000	0.0181	25.4004
GCC	512	4	10 000	0.0147	12.9686
GCC	512	8	10 000	0.0088	7.1592
GCC	512	16	10 000	0.0088	4.0574
GCC	512	32	10 000	0.0115	2.6117
GCC	2048	1	1 000	0.0623	77.1689
GCC	2048	2	1 000	0.0791	39.0673
GCC	2048	4	1 000	0.0692	19.6133
GCC	2048	8	1 000	0.1033	10.0464
GCC	2048	16	1 000	0.0971	5.174
GCC	2048	32	1 000	0.1038	2.7467
GCC	8192	1	100	0.9902	125.295
GCC	8192	2	100	1.1225	63.2725
GCC	8192	4	100	1.0953	31.7264
GCC	8192	8	100	1.1748	15.8625
GCC	8192	16	100	1.2046	7.9815
GCC	8192	32	100	1.2601	4.0859
GCC	32768	1	10	15.8826	199.181
GCC	32768	2	10	20.1785	100.1272
GCC	32768	4	10	18.0209	50.0713
GCC	32768	8	10	17.1956	25.0854
GCC	32768	16	10	17.4827	12.5836
GCC	32768	32	10	17.7285	6.3286

Tabelle 1: Testergebnisse der GCC kompilierten Version des Game-of-Life. Zeiten gerundet auf vier Stellen.

Compiler	Size	#Threads	Repetitions	Initialization in s	Calculation in s
ICC	128	1	100 000	0.0021	8.5322
ICC	128	2	100 000	0.0022	4.5041
ICC	128	4	100 000	0.0025	2.5492
ICC	128	8	100 000	0.0035	1.9608
ICC	128	16	100 000	0.0054	1.8089
ICC	128	32	100 000	0.0127	1.9434
ICC	512	1	10 000	0.0061	13.4002
ICC	512	2	10 000	0.0063	6.742
ICC	512	4	10 000	0.0072	3.4289
ICC	512	8	10 000	0.0082	1.7867
ICC	512	16	10 000	0.0099	0.9776
ICC	512	32	10 000	0.0195	0.5929
ICC	2048	1	1 000	0.0496	21.5322
ICC	2048	2	1 000	0.0485	10.7675
ICC	2048	4	1 000	0.052	5.4267
ICC	2048	8	1 000	0.0519	2.7251
ICC	2048	16	1 000	0.0546	1.3692
ICC	2048	32	1 000	0.0638	0.6911
ICC	8192	1	100	0.7441	37.5091
ICC	8192	2	100	0.7437	19.5603
ICC	8192	4	100	0.7483	9.9158
ICC	8192	8	100	0.7696	5.1219
ICC	8192	16	100	0.7756	2.5619
ICC	8192	32	100	0.7851	1.2802
ICC	32768	1	10	11.8049	55.2352
ICC	32768	2	10	11.8445	27.6332
ICC	32768	4	10	11.8567	13.8855
ICC	32768	8	10	12.2583	6.9565
ICC	32768	16	10	12.2543	3.4985
ICC	32768	32	10	12.3533	1.7492

Tabelle 2: Testergebnisse der ICC kompilierten Version des Game-of-Life. Zeiten gerundet auf vier Stellen.

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

[Mar21] MARKWARDT, Dr. U. *Introduction to HPC at ZIH*.
https://doc.zih.tu-dresden.de/misc/HPC-Introduction.pdf. 2021

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

[Vig17] VIGNA, Sebastiano. *Further scramblings of Marsaglia's xorshift generators*.
https://vigna.di.unimi.it/ftp/papers/xorshiftplus.pdf. 2017