TECHNISCHE UNIVERSITÄT DRESDEN

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

Komplexpraktikum Paralleles Rechnen - MPI parallele Programmierung

Daniel Körsten

Dresden, 3. März 2022

Inhaltsverzeichnis

1 Aufgabenbeschreibung

In dieser Aufgabe soll eine MPI-parallele Variante von `Conway's Game-of-Life` in der Programmiersprache `C` implementiert werden.
Anschließend soll die Simulation mit verschieden großen Feldern und Anzahl von ranks 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 `MPI` erfolgen soll. `MPI` ist eine API, die die parallele Berechnung über mehrere getrennte Computer ermöglicht. Ich verwendete in diesem Praktikum die Open-Source Implementierung `OpenMPI`.
Im Gegensatz zu `OpenMP` benötigt MPI kein *Shared-Memory System*, sondern kann auf dedizierten Rechnern innerhalb eines Clusters ausgeführt werden.

`MPI` steht dabei für *Message Passing Interface*. Konkret bedeutet das, dass die parallel laufenden Prozesse Daten über Nachrichten austauschen. Dadurch wird es möglich, das Problem aufzuteilen "*divide and conquer*". Der Datenaustausch geschieht nicht automatisch, sondern der Programmierer muss explizit angeben, welcher Prozesse zu welchem Zeitpunkt Daten an einen anderen Prozess sendet.

Der Nachrichtenaustausch kann dabei über verschiedene Wege erfolgen.
Innerhalb eines Sockets, oder auch zwischen Sockets auf einem Dual Socket Board, können Nachrichten über den gemeinsamen Speicher oder Interconnects, wie Infiniband oder Infinity Fabric mit sehr hoher Geschwindigkeit und geringer Latenz ausgetauscht werden.
Kommen mehrere Compute Nodes in einem Cluster zum Einsatz, kann die Kommunikation über das Netzwerk, in der Regel über TCP/IP, erfolgen.
Die genaue Testumgebung mit Kommunikationsmöglichkeiten wird später in 3.1 diskutiert.BELEGE

Weitere Besonderheiten:

- Für 1, 4, 16, 64, 128 und 256 ranks die Ausführungszeiten messen und vergleichen
- Quadratische Feldgrößen von 2048×2048, 8192×8192, 32768×32768 und 131072×131072
- Die Felder sollen auf die ranks aufgeteilt werden
- benachbarte ranks sollen möglichst nah beieinander gescheduled sein

Begriffsklärung rank: Jeder Prozess, der am Nachrichtenaustausch teilnimmt, muss eine eindeutige Kennung besitzen, um gezielt Nachrichten versenden und empfangen zu können.
`MPI` erstellt dabei Prozessgruppen (typischerweise eine, wenn nicht anders spezifiziert), in der ein Prozess über seinen *rank* identifiziert wird.
Der rank ist dabei eine Zahl im Intervall $[0, N-1]$, $N \ldots$ Anzahl der gestarteten Prozesse.

2 Implementierung

Bei Aufgabe B & C allokierte ich einen Speicherbereichs der Größe
`columns * rows * sizeof(u_int8_t)` durch die C-Funktion `malloc()`.
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.
Bei MPI muss dieses Feld nun so aufgeteilt werden, dass jeder Prozess einen Teil des Feldes bearbeitet. Ich entschied mich dafür für ein chessboard Layout, also der Aufteilung des Feldes in gleich große Quadrate.
Der Grund ist, dass hier der Kommunikationsaufwand minimal wird, da jeder Prozess seinen direkten Nachbarn nur die äußerste Reihe senden (und umgekehrt von ihnen empfangen) muss.
Im ersten Schritt muss MPI initialisiert werden. Das erfolgt direkt am Anfang:

```
1 static int rank, cluster;
2
3 int main(int argc, char *argv[]) {
4   // MPI
5   MPI_Init(&argc, &argv);
6   MPI_Comm_size(MPI_COMM_WORLD, &cluster);
7   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
8 }
```

Listing 1: MPI Initialisierung mit Bestimmung von rank und cluster

Die beiden Variablen rank und cluster enthalten dabei direkt die wichtigsten Informationen: Den rank des Prozesses und die Gesamtzahl der Prozesse.
Da die Größe des Feldes ebenfalls bekannt ist, lässt sich nun das Feld aufteilen.

Es gilt nun zu bestimmen, in wie viel Zeilen und Spalten das Feld zerlegt werden muss. Da die Felder quadratisch sind und genau wie die ranks mit einer Zweierpotenz ausgedrückt werden können, ist eine Aufteilung immer möglich.
Dafür verwendete ich eine Funktion, die den *Logarithmus Dualis* der rank Größe berechnet und so die Zeilen und Spalten berechnet.
Dabei stellt sich ein Problem: Besitzt die Zweierpotenz einer rank Größe einen ungeraden Exponenten, bspw. $128 = 2^7$, entspricht die Anzahl der Zeilen nicht der der Spalten.
In diesem Fall entschied ich mich mehr Blocks pro Zeile statt Spalte zu verwenden. Dies ist eine willkürliche Festlegung und sollte keinen Einfluss auf die Performance haben.

```
1 void init_chessboard() {
2   u_int32_t exponent = (u_int32_t) log2((double) cluster);
3   if (exponent & 1) {
4     // ungerade
5     blocks_per_col = pow(2, (exponent / 2));
6     blocks_per_row = blocks_per_col * 2;
7   } else {
```

```
8       // gerade
9       blocks_per_row = blocks_per_col = pow(2, (exponent / 2));
10    }
11    block_row = rows / blocks_per_row + 2;
12    block_col = columns / blocks_per_col + 2;
13    return;
14  }
```

Listing 2: Bestimmung der Zeilen und Spalten des chessboard Layouts

In Zeile 11 und 12 bestimme ich nun direkt die Zeilen und Spalten jedes Blockes, in dem ich die Zeilen und Spalten des Feldes durch die jeweilige Anzahl der Blöcke teile.
Anschließend addiere ich noch 2. Wie eingangs erwähnt, benötigt jeder Block die äußerste Reihe seiner Nachbarn, um das Game-of-Life spielen zu können.
Deshalb vergrößere ich das Feld um jeweils 2 Zeilen und 2 Spalten.

Auf dieser Grundlage kann jeder rank nun sein eigenen Spielfelder initialisieren:

```
1  // initializing states and pointers
2  u_int8_t *state_1 = (u_int8_t *) malloc(block_row * block_col * sizeof(u_int8_t));
3  u_int8_t *state_2 = (u_int8_t *) malloc(block_row * block_col * sizeof(u_int8_t));
4  u_int8_t *state_in = state_1;
5  u_int8_t *state_out = state_2;
6  u_int8_t *state_tmp = NULL;
7
```

Listing 3: Initialisierung der Spielfelder

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 beiden Pointer, was dazu führt, dass die im vorhergehenden Schritt berechnete Folgegeneration zur aktuellen Generation wird und eine neue Generation berechnet werden kann.

2.1 MPI Kommunikation

2.1.1 Nachrichtenaustausch

MPI Isend usw erklären

2.1.2 Neighbour Matrix

Für den späteren Datenaustausch über MPI ist es notwendig, dass ein Prozess mit einem gegeben rank seine Nachbarn kennt. Da es sich um ein Torus-förmiges Spielfeld handelt, hat ein rank in der letzten Spalte als rechten Nachbarn nicht den $rank + 1$, da dieser in der ersten Spalte der nächsten Zeile liegt usw.
Um dieses Problem zu lösen und die Kommunikation später zu erleichtern, generiere ich eine *Neighbour Matrix*, bei der die Kanten und Ecken entsprechend gespiegelt werden.

```
1  // Generierung der Matrix
2  // 2 Zeilen und 2 Spalten groesser, um die Spiegelung zu ermoeglichen
```

```
nb_row = blocks_per_row + 2;
nb_col = blocks_per_col + 2;
u_int32_t *neighbour_matrix = (u_int32_t *) malloc(nb_row * nb_col * sizeof(
    u_int32_t));

// position in neighbour_matrix
rank_index = (rank / blocks_per_col + 1) * nb_col + (rank % blocks_per_col + 1);
init_neighbour(neighbour_matrix);

// Initialisieren der Matrik mit den ranks
void init_neighbour(u_int32_t *neighbour_matrix) {
  u_int32_t z = 0;
  for (int i = 1; i < nb_row - 1; i++) {
    for (int j = 1; j < nb_col - 1; j++) {
      neighbour_matrix[i * nb_col + j] = z;
      z++;
    }
    //left
    neighbour_matrix[i * nb_col] = neighbour_matrix[(i + 1) * nb_col - 2];
    //right
    neighbour_matrix[(i + 1) * nb_col - 1] = neighbour_matrix[i * nb_col + 1];
  }
  for (int i = 1; i < nb_col - 1; i++) {
    //top
    neighbour_matrix[i] = neighbour_matrix[(nb_row - 2) * nb_col + i];
    //bottom
    neighbour_matrix[(nb_row - 1) * nb_col + i] = neighbour_matrix[nb_col + i];
  }
  // top left corner
  neighbour_matrix[0] = neighbour_matrix[(nb_row - 1) * nb_col - 2];
  // top right corner
  neighbour_matrix[nb_col - 1] = neighbour_matrix[(nb_row - 2) * nb_col + 1];
  // bottom left
  neighbour_matrix[(nb_row - 1) * nb_col] = neighbour_matrix[nb_col];
  // bottom right
  neighbour_matrix[nb_row * nb_col - 1] = neighbour_matrix[nb_col + 1];
  return;
}
```

Listing 4: Generierung der Neighbour Matrix

Die Variable *rank_index* enthält dabei den Index des ranks innerhalb der Matrix. Dieser Index fungiert später als konstantes Offset, um Nachrichten an seine Nachbarn adressieren zu können.

2.2 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]. Die `Xorshift` Generatoren sind eine Familie von Pseudozufallszahlengeneratoren, die sich durch eine hohe Geschwindigkeit und einer anpassbaren Periodenlänge auszeichnen. `Xorshift128+` 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.

Da die AMD Rome EPYC 7702 Prozessoren nur AVX2 und nicht AVX512 unterstützen (vergleiche 3.1), verwende ich die Funktion `avx_xorshift128plus`, welche auf 256 Bit Registern arbeitet.
Meine Idee war, die 256 generierten Bits auf die 8 Bit großen Zellen des Spiels aufzuteilen. Damit lassen sich pro Durchgang 32 Zellen mit Zufallszahlen füllen, was zu einer deutlichen Geschwindigkeitssteigerung führen sollte.
Zusammengesetzt ergibt sich daraus folgender Code:

```
void field_initializer(u_int8_t *state, u_int32_t *neighbour_matrix) {
  //fills fields with random numbers 0 = dead, 1 = alive
  // use diffrent seed for every rank
  unsigned seed = time(0) + rank;
  // top & bottom
  for (int i = 2; i < block_col - 2; i++) {
    state[block_col + i] = rand_r(&seed) % 2;
    state[(block_row - 2) * block_col + i] = rand_r(&seed) % 2;
  }
  //left & right + corners
  for (int i = 1; i < block_row - 1; i++) {
    state[i * block_col + 1] = rand_r(&seed) % 2;
    state[(i + 1) * block_col - 2] = rand_r(&seed) % 2;
  }
  // send everything
  MPI_Status status[16];
  MPI_Request request[16];
  //bottom
  MPI_Isend(&state[(block_row - 2) * block_col + 1], block_col - 2, MPI_UINT8_T,
            neighbour_matrix[rank_index + neighbour_col], 0,
            MPI_COMM_WORLD, &request[0]);
  MPI_Irecv(&state[1], block_col - 2, MPI_UINT8_T,
            neighbour_matrix[rank_index - neighbour_col], 0,
            MPI_COMM_WORLD, &request[1]);
  // and so on for every edge and corner

  // do the middle, while sending/receiving
  for (int i = 2; i < block_row - 2; i++) {
    for (int j = 2; j < block_col - 2; j++) {
      state[i * block_col + j] = rand_r(&seed) % 2;
    }
```

```
32    }
33    // Wait for all IPC to complete
34    MPI_Waitall(16, request, status);
35  }
```

Listing 5: Daten Initialisierung

Abbildung 1: Mit `Xorshift128+` generiertes Spielfeld der Größe: 64×64

2.3 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.

```
1  for (int i = 0; i < repetitions; i++) {
2    calculate_next_gen(state_out, state_in);
3    state_tmp = state_in;
4    state_in = state_out;
5    state_out = state_tmp;
6  }
```

Listing 6: 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 unterscheidet sich nur unwesentlich von der Berechnung des inneren Feldes.

2.3.1 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 diese bitweisen Operatoren sich gut bei SIMD Operationen verwenden lassen.

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

```
for (int i = 1; i < rows - 1; i++) {
  #pragma omp simd
  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 7: Berechnung der inneren Zellen

Die erste Schleife iteriert dabei über jede Zeile und die Zweite in jeder Zeile durch jede Zelle. Ich habe dafür die `OpenMP` Direktive

```
#pragma omp simd
```

verwendet.
Dabei wird jedoch nur die innere Schleife parallelisiert. Das bewirkt, dass die Spalten jeweils parallel berechnet werden. `OpenMP` wäre in der Lage, mit `collapse(2)` zwei geschachtelte Schleifen zu parallelisieren. In meinen Tests führte dies zu einer Verschlechterung der Performance, weswegen ich es nicht verwendet habe.
Zum anderen kann der Compiler, bei meiner Implementierung, die äußere Schleife modifizieren und so eventuelle Optimierungen vornehmen.

2.3.2 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 simd
```

parallelisiert.

```
void calculate_top(u_int8_t *state, u_int8_t *state_old) {
  #pragma omp simd
  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 8: Berechnung der obersten Zeile

2.3.3 Ecken

Bei den Ecken ist keine Parallelisierung möglich und auch nicht notwendig, da vier Ecken bei einem Feld mit mehr als 16.000 Zellen 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 9: Berechnung der Ecke oben links

2.4 Zeitmessung

Ich habe mich für eine Zweistufige Zeitmessung entschieden.

In erster Instanz messe ich die Ausführungszeiten der Funktionen `calculate_next_gen()` und `field_initializer()`.
Die vergangene Zeit messe ich mit der Funktion `clock_gettime()`.
Für die Berechnung der Ausführungszeit wird vor und nach Ausführung der zu untersuchenden Funktion `clock_gettime()` aufgerufen. Die Differenz aus den beiden Momentaufnahmen entspricht der jeweiligen Zeit.

Zusätzlich messe ich die Ausführungszeit des kompletten Programms mit dem Linux Befehl *time*. Da die Funktionen `calculate_next_gen()` und `field_initializer()` den größten Abteil der Programm haben, lässt sich, zumindest näherungsweise, der MPI Overhead zum spawnen initialisieren der Prozesse berechnen.

```
clockid_t clk_id = CLOCK_MONOTONIC;
double time_calc = 0;
struct timespec calc_s, calc_e;
for (int i = 0; i < repetitions; i++) {
  clock_gettime(clk_id, &calc_s);
  // function call
  clock_gettime(clk_id, &calc_e);
  time_calc += (double) (calc_e.tv_nsec-calc_s.tv_nsec) / 1000000000 +
               (double) (calc_e.tv_sec-calc_s.tv_sec);
}
printf("Calculation took %f seconds to execute.\n", time_calc);
```

Listing 10: Berechnung der Ausführungszeit eines *function calls*

2.5 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. Dabei fügt jeder rank seiner Ausgabedatei seinen rank an. Das Bild ließe sich aus diesen Einzelbildern konstruieren, ist jedoch nicht Schwerpunkt dieser Aufgabe.
Alle Funktionen sowie die Syntax lassen sich über den Parameter `--help` ausgeben.

Um mehrfache Ausgaben auf der Konsole zu vermeiden, werden Ausgaben nur vom Prozess mit dem rank 0 ausgeführt.

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 einem 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
- `mfma`/`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 entscheiden, 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.

Anmerkung zur Notation: Mit einer Feldgröße von 128 meine ich ein Feld mit 128×128 Zellen. Da hier im Praktikum alle Feldgrößen quadratisch sind, ist die Angabe der zweiten Größe redundant, weswegen ich auch auf diese verzichte.

Anmerkung zu den Diagrammen: In den Diagrammen finden sich bei einigen Feldgrößen keine Datenpunkte. Ist das der Fall, war das Ergebnis der Zeitmessung 0 Sekunden. Da der Logarithmus von 0 jedoch $-\infty$ entspricht, ist eine Darstellung nicht sinnvoll.
Das Ergebnis von 0 Sekunden entspricht natürlich nicht ganz der Realität, aber es lässt sich festhalten, dass die Berechnung so schnell abgeschlossen war, dass diese faktisch nicht messbar war.

4 Testergebnisse

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

4.1 Dateninitialisierung

Zuerst möchte ich auf die Ausführungszeiten der Funktion `field_initializer()` des Spiels eingehen.
Diese Funktion füllt das Spielfeld mit zufälligen Werten. Wie Eingangs in 2.2 beschrieben, lässt sich diese Funktion nicht mit SIMD Instruktionen parallelisieren.
Ich implementierte jedoch einen `Xorshift128+` Pseudozufallszahlengenerator, um das Problem SIMD parallel auszuführen.
Die Ergebnisse habe ich hier dargestellt.

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

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

Betrachtet man die Werte zwischen der seriellen Version und der SIMD parallelen Version, fällt ein deutlicher Speedup von etwa 9 auf. Diese Parallelisierung war folglich sehr effektiv.
Der Unterschied zwischen beiden Compilern ist hingegen vernachlässigbar klein.

4.2 Berechnung

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

Betrachtet man die Ergebnisse des `GCC`, fällt auf, dass sich die beiden Graphen kaum unterschieden. Das hängt damit zusammen, dass der `GCC` unabhängig von der `OpenMP` Compiler Direktive immer SIMD Instruktionen verwendet (Vergleiche 3.1).
Die Verwendung von `OpenMP` verschlechterte die Ausführungszeiten in diesem Fall minimal.

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

Beim `ICC` wird erstmals deutlich, dass die Verwendung von SIMD Instruktionen einen erheblichen Performance Vorteil bewirkt. Der Speedup liegt bei circa 24.
Vergleicht man die Zeiten des `ICC` mit denen des `GCC`, fällt auf, dass der Programmcode des `GCC` etwas schneller ist.

A Tabellen

Hinweis zur Notation: *without SIMD* bedeutet nicht, dass keine SIMD Instruktionen verwendet werden. Es bedeutet, dass das Programm gemäß der Aufgabenstellung ohne `OpenMP` Compiler Direktiven compiliert wurde. Wie bereits in 3.1 festgestellt, verwendet der `GCC` dennoch SIMD Instruktionen.

Compiler	Size	SIMD	Repetitions	Initialization in s
gcc	128	without SIMD	100	0.0
gcc	128	SIMDxorshift	100	0.0
gcc	512	without SIMD	100	0.0
gcc	512	SIMDxorshift	100	0.0
gcc	2048	without SIMD	100	0.046
gcc	2048	SIMDxorshift	100	0.0
gcc	8192	without SIMD	100	0.788
gcc	8192	SIMDxorshift	100	0.081
gcc	32768	without SIMD	100	12.583
gcc	32768	SIMDxorshift	100	1.399
icc	128	without SIMD	100	0.0
icc	128	SIMDxorshift	100	0.0
icc	512	without SIMD	100	0.0
icc	512	SIMDxorshift	100	0.0
icc	2048	without SIMD	100	0.05
icc	2048	SIMDxorshift	100	0.0
icc	8192	without SIMD	100	0.849
icc	8192	SIMDxorshift	100	0.079
icc	32768	without SIMD	100	13.846
icc	32768	SIMDxorshift	100	1.351

Tabelle 1: Testergebnisse der Funktion `field_initializer()`. Zeiten gerundet auf 3 Nachkommastellen.

Compiler	Size	SIMD	Repetitions	Calculation in s
gcc	128	without SIMD	100	0.0
gcc	128	SIMD	100	0.0
gcc	512	without SIMD	100	0.009
gcc	512	SIMD	100	0.01
gcc	2048	without SIMD	100	0.082
gcc	2048	SIMD	100	0.093
gcc	8192	without SIMD	100	1.38
gcc	8192	SIMD	100	1.49
gcc	32768	without SIMD	100	18.752
gcc	32768	SIMD	100	21.272
icc	128	without SIMD	100	0.002
icc	128	SIMD	100	0.0
icc	512	without SIMD	100	0.15
icc	512	SIMD	100	0.006
icc	2048	without SIMD	100	2.47
icc	2048	SIMD	100	0.09
icc	8192	without SIMD	100	42.054
icc	8192	SIMD	100	1.722
icc	32768	without SIMD	100	633.289
icc	32768	SIMD	100	25.341

Tabelle 2: Testergebnisse der Funktion `calculate_next_gen()`. Zeiten gerundet auf 3 Nachkommastellen.

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

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

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