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, 5. 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 3.2 diskutiert.

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 Spielfeldes in gleich große Teilfelder.
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

Für den Nachrichtenaustausch zwischen den Prozessen verwende ich asynchrone (*non-blocking*) Kommunikation. Asynchrone Kommunikation bedeutet, dass nicht auf die Zustellung bzw. den Empfang der Nachricht gewartet wird. Die Funktionsaufrufe blockieren also nicht.
Das bietet sich aus mehren Gründen an:
Zum einen benötigen die Kanten und Ecken aufgrund des Torus-förmigen Spielfeldes eine separate Behandlung. Gleichzeit müssen aber auch nur Kanten und Ecken an andere Prozesse gesendet werden.
Zum anderen benötigt die Kommunikation viel Zeit, da jeder Prozess zu seinen 8 Nachbarn einen *send* und *receive* Aufruf pro Generation und einmalig für die Initialisierung durchführen muss. Daher bietet sich folgender Ablauf an:

1. Kanten und Ecken berechnen

2. Berechnete Kanten und Ecken an die 8 Nachbarn senden (umgekehrt 4 Kanten und 4 Ecken empfangen)

3. Während der Kommunikation im Hintergrund das innere Feld berechnen

4. Vor der Rückkehr aus der Funktion warten, bis die Kommunikation abgeschlossen ist

MPI bietet für die asynchrone Kommunikation die Funktionen `MPI_Isend()` und `MPI_Irecv()` an. Für das Warten auf den Abschluss der Kommunikation verwende ich `MPI_Waitall()`.
Beispiel für den Austausch einer Kante zwischen zwei ranks (vereinfacht dargestellt):

```
    // calculate edges and corners
    ...
    // afterwards, send and receive
    MPI_Status status[16];
    MPI_Request request[16];
    //bottom edge
    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]);
    // do this for the remaining 3 edges and 4 corners
    ...
    // calculate inner part
    ...
    // wait for communication to complete
    MPI_Waitall(16, request, status);

```

Listing 4: Asynchrone Kommunikation zwischen 2 ranks

2.1.2 Neighbour Matrix

Für den 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.

```
// Generierung der Matrix
// 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);
```

```
10
11 // Initialisieren der Matrix mit den ranks
12 void init_neighbour(u_int32_t *neighbour_matrix) {
13   u_int32_t z = 0;
14   for (int i = 1; i < nb_row - 1; i++) {
15     for (int j = 1; j < nb_col - 1; j++) {
16       neighbour_matrix[i * nb_col + j] = z;
17       z++;
18     }
19     //left
20     neighbour_matrix[i * nb_col] = neighbour_matrix[(i + 1) * nb_col - 2];
21     //right
22     neighbour_matrix[(i + 1) * nb_col - 1] = neighbour_matrix[i * nb_col + 1];
23   }
24   for (int i = 1; i < nb_col - 1; i++) {
25     //top
26     neighbour_matrix[i] = neighbour_matrix[(nb_row - 2) * nb_col + i];
27     //bottom
28     neighbour_matrix[(nb_row - 1) * nb_col + i] = neighbour_matrix[nb_col + i];
29   }
30   // top left corner
31   neighbour_matrix[0] = neighbour_matrix[(nb_row - 1) * nb_col - 2];
32   // top right corner
33   neighbour_matrix[nb_col - 1] = neighbour_matrix[(nb_row - 2) * nb_col + 1];
34   // bottom left
35   neighbour_matrix[(nb_row - 1) * nb_col] = neighbour_matrix[nb_col];
36   // bottom right
37   neighbour_matrix[nb_row * nb_col - 1] = neighbour_matrix[nb_col + 1];
38   return;
39 }
```

Listing 5: Generierung der Neighbour Matrix

Die Variable *rank_index* enthält dabei den Index des ranks innerhalb der Matrix. Dieser Index fungiert 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.
Wie in 2.1.1 erklärt, ist der Ablauf in mehrere Schritte unterteilt.
Für die Dateninitialisierung jeder Zelle mit Null oder Eins, verwende ich den Pseudo-Zufallszahlengenerator `rand_r()`. Der `seed` setzt sich aus `time()+rank` zusammen. Damit verändert sich der `seed` in Abhängigkeit der Zeit und dem rank.
Zuerst werden die Kanten und Ecken initialisiert, danach erfolgt Kommunikation. Währenddessen wird das innere Feld initialisiert und auf den Abschluss der Kommunikation gewartet.

Zusammengesetzt ergibt sich daraus folgender Code. Da ich den Ablauf der Kommunikation schon in 2.1.1 erläutert habe, vereinfache ich das Beispiel an dieser Stelle.

```
void field_initializer(u_int8_t *state, u_int32_t *neighbour_matrix) {
  //fills fields with random numbers 0 = dead, 1 = alive
  // use different 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;
  }
  // MPI_Isend and MPI_Irecv 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;
    }
  }
  // Wait for all IPC to complete
  MPI_Waitall(16, request, status);
  return;
}
```

Listing 6: Daten Initialisierung

Abbildung 1: Mit `rand_r()` 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`, einen auf das Array der Folgegeneration `*state` und einen weiteren für die Neighbour Matrix.
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, neighbour_matrix);
  state_tmp = state_in;
  state_in = state_out;
  state_out = state_tmp;
}
```

Listing 7: Vertauschen der Pointer vor jedem Funktionsaufruf (vereinfacht)

2.3.1 Kanten und Ecken

Ähnlich der Feld Initialisierung, werden zuerst die Kanten und Ecken berechnet, damit diese im nächsten Schritt gesendet werden können. Da die Kanten jeweils nur aus einer Zeile bzw. Spalte bestehen, wird nur eine `for`-Schleife benötigt.
Ich habe mich willkürlich dazu entschieden, die Ecken mit der linken und rechten Spalte zu berechnen. Man hätte sie aber auch genauso gut mit der obersten und untersten Zeile berechnen können. Vereinfacht habe ich nur die Berechnung der linken und rechten Seite dargestellt.

```
//left & right + corners
for (int i = 1; i < block_row - 1; i++) {
  u_int8_t sum_of_l_edge = state_old[(i - 1) * block_col] +
  state_old[(i - 1) * block_col + 1] +
  state_old[(i - 1) * block_col + 2] +
  state_old[i * block_col] +
  state_old[i * block_col + 2] +
  state_old[(i + 1) * block_col] +
  state_old[(i + 1) * block_col + 1] +
  state_old[(i + 1) * block_col + 2];
  state[i * block_col + 1] = (sum_of_l_edge == 3) | ((sum_of_l_edge == 2) &
    state_old[i * block_col + 1]);

  u_int8_t sum_of_r_edge = state_old[i * block_col - 3] +
  state_old[i * block_col - 2] +
  state_old[i * block_col - 1] +
  state_old[(i + 1) * block_col - 3] +
  state_old[(i + 1) * block_col - 1] +
  state_old[(i + 2) * block_col - 3] +
  state_old[(i + 2) * block_col - 2] +
  state_old[(i + 2) * block_col - 1];

```

```
22    state[(i + 1) * block_col - 2] =
23    (sum_of_r_edge == 3) | ((sum_of_r_edge == 2) & state_old[(i + 1) * block_col - 2])
        ;
24  }
25  // top & bottom
26  for (int i = 2; i < block_col - 2; i++) {
27    ...
28  }
```

Listing 8: Berechnung der Kanten und Ecken

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.

2.3.2 Inneres Feld

Im ersten Schritt werden alle inneren Zellen berechnet. Dafür verwende ich zwei geschachtelte `for`-Schleifen:

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

Listing 9: Berechnung der inneren Zellen

Diese Berechnung findet während der Kommunikation statt und wird nicht mit anderen ranks geteilt. Die Berechnung der Kanten und Ecken wäre auch in diesem Schritt möglich, in dem man die Zählvariablen um eins früher beginnen und eins später enden lassen würde.
Dann könnte jedoch erst nach Abschluss der Berechnung der Datenaustausch stattfinden, was den Performance Vorteil der asynchronen Kommunikation kaputt machen würde.

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 und 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]. Ein EPYC 7702 besteht aus 64 physischen Kernen. *Simultaneous Multithreading* ist deaktiviert. Ein Node verwendet 2 EPYC Prozessoren (2x 64 Cores) auf einem Dual-Socket Board und 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.
Ich kompilierte das Programm mit dem `GCC` (**G**NU **C**ompiler **C**ollection, Version 11.2) unter Verwendung von OpenMPI (Version 4.1.2) mit den Compiler-Flags:

- `O3` - Optimierungsflag des Compilers (u.a. Verwendung von SIMD Instruktionen)
- `mavx2` - Verwendung von AVX2
- `mfma` - Verwendung von FMA
- `lm` - Notwendig für das einbinden der *math library*

Um die NUMA Eigenschaften optimal zu nutzen, verwendete ich bei der Slurm Ausführung die Parameter:

- `--cpu_bind=cores`
- `--distribution=block:block`

Der CPU bind bewirkt, dass jedem Prozess ein (physischer) Core zugewiesen wird.

Die *distribution* bestimmt die Verteilung der Prozesse über die Sockets und Nodes.

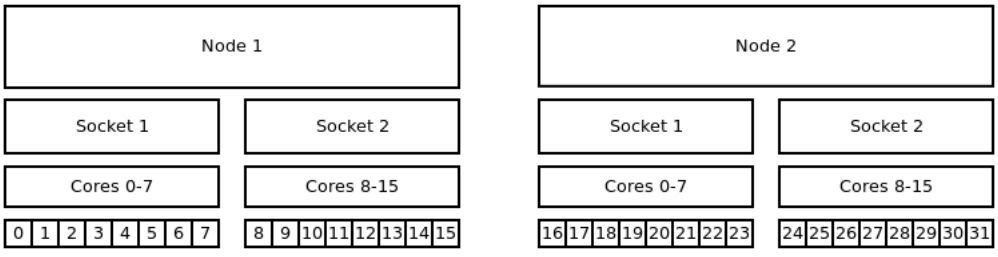

Abbildung 2: Aufteilung der ranks über mehrere Sockets und Nodes mit `block:block`.
Quelle: ZIH HPC Compendium

Es werden also zuerst alle Cores innerhalb eines Sockets und anschließend innerhalb eines Nodes verwendet. Je näher die ranks beieinander liegen, desto schneller ist die Kommunikation. Darauf möchte ich unter Betrachtung der AMD EPYC CPU einmal genauer eingehen.

3.2 AMD EPYC CPU Layout

Bevor ich zur Testmethode und den Ergebnissen übergehe, möchte ich erst auf die Struktur der AMD EPYC CPU eingehen, um Ergebnisse später besser deuten zu können.

Abbildung 3: CPU Layout eines AMD EPYC 7002.
Quelle: HPC Tuning Guide for AMD EPYC 7002 Series Processors

Der verwendete AMD EPYC 7702 ist ein 64 Kern Prozessor. Jeder Socket besteht dabei aus 4 *Zeppelins* (Orange in der Abbildung), die über ein *Infinity Fabric* untereinander und mit den 4 Zeppelins des anderen Sockets sehr schnell Daten austauschen können [AMD20]. Jeder Zeppelin besteht damit aus 16 Cores.

Das zu erwartende Ergebnis ist, dass der Geschwindigkeitsvorteil geringer wird, sobald man die 16 Cores übersteigt. Ab da müssen die Zeppelins über Infinity Fabric miteinander kommunizieren, was mehr Zeit benötigen sollte, als die Kommunikation innerhalb eines Zeppelins.

Die nächste interessante Schwelle liegt bei 64 Cores. Sobald sie überschritten wird, müssen die ranks über den Socket hinweg kommunizieren, was aufgrund der längeren Leitungswege ebenfalls Zeit kostet.

Mit der Überschreitung von 128 Cores muss dann zwischen zwei Nodes kommuniziert werden. Hier sind die größten Geschwindigkeitseinbrüche zu erwarten, da die Kommunikation über das Netzwerk im Vergleich zur Kommunikation auf einem Board höhere Latenzen mit sich zieht.

3.3 Testmethode

Die Tests wurden automatisiert mit `sbatch`-Skripten ausgeführt. Um Schwankungen auszugleichen, wurde jede Messung 20 mal wiederholt. Ich allokierte jeden bei den Messungen den Knoten exklusiv, um Einflüsse durch koexistierende Prozesse auszuschließen.
Da nicht genügend Rechenressourcen zur Verfügung standen, machte ich die Anzahl der Simulationsschritte von der Feldgröße abhängig.
Ich entschied mich dabei für einen exponentielle Abnahme der Wiederholungen, da die Feldgröße exponentiell ansteigt.

Dafür testete ich die höchste Anzahl an ranks (256) mit der kleinsten Feldgröße (2048). Die minimal Ausführungszeit sollte bei etwa einer Sekunde (ohne den Overhead von MPI) liegen, um verlässliche Werte zu erhalten. Diese wurde bei 10.000 Simulationsschritten erreicht. Mit der nächst höheren Feldgröße sinkt die Anzahl der Wiederholungen um eine Dekade.

Die Messwerte sind daher ungeeignet, um zu beurteilen, wie sich die Ausführungszeit verändert, wenn man die Anzahl der ranks gleich belässt, aber die Feldgröße ändert.
Dass die Ausführungsgeschwindigkeit mit der Feldgröße ansteigt ist logisch. Interessanter für das Praktikum ist jedoch das Verhalten bei MPI-paralleler Ausführung, was sich beurteilen lässt.

Aus den 20 Wiederholungen pro Messung habe ich den Mittelwert gebildet und Logarithmische Diagramme erzeugt. Diese Darstellung bietet sich aufgrund der exponentiell ansteigenden Feldgröße und ranks 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 zur exklusiven Nutzung von Knoten: Aufgrund der begrenzten Rechenressourcen stand ich vor der Entscheidung Knoten exklusiv zu allokieren oder die Anzahl der Simulationsschritte von der Feldgröße abhängig zu machen.
Ich entschied mich für letzteres, wie oben beschrieben. Warum zeige ich X STELLE.

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.

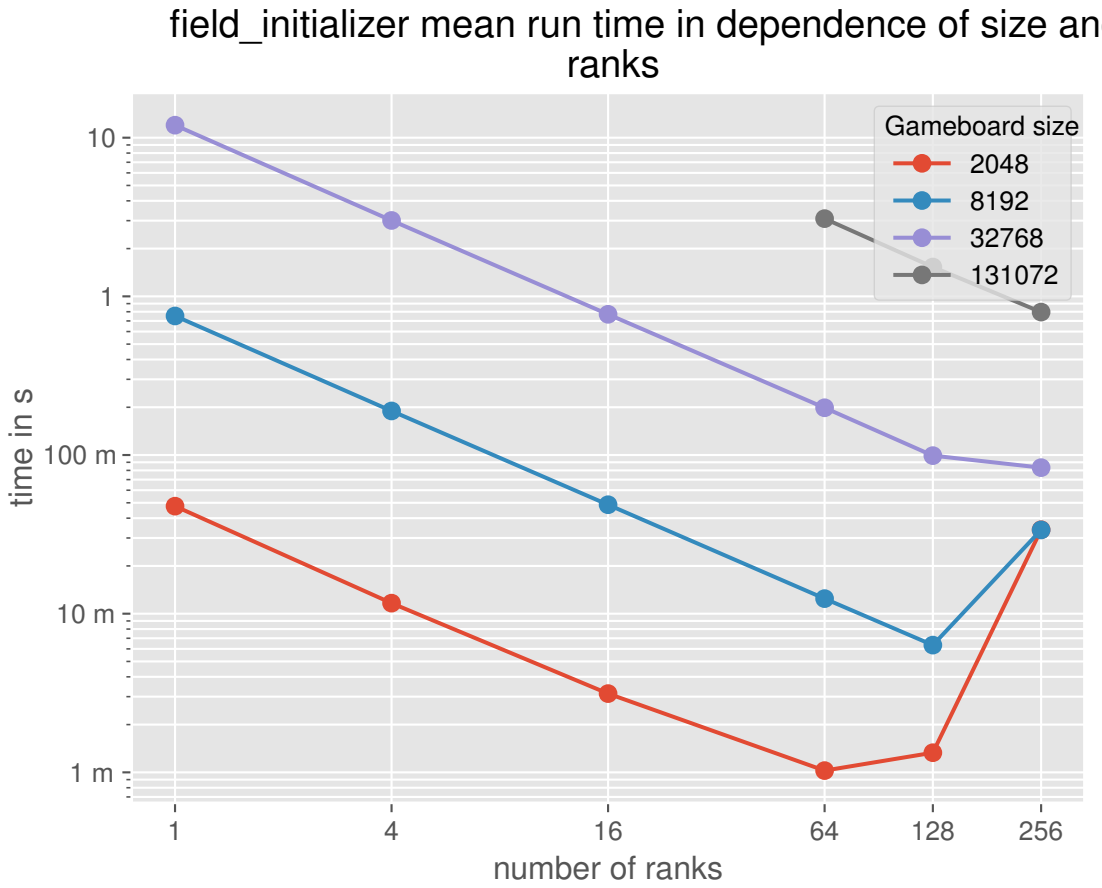

Abbildung 4: Logarithmische Darstellung der Ausführungszeit der Funktion `field_initializer()`.

Die allgemeine Tendenz zeigt, was zu erwarten war: Mehr ranks verbessern die Ausführungsgeschwindigkeit.
Es ist zu erkennen, dass bei 64 Cores für kleine Feldgrößen ein Minimum erreicht wird. Mehr Cores verschlechtern wieder die Performance. Warum? Die Kommunikation nimmt hier also mehr Zeit in Anspruch, als der eigentliche Vorgang.
Das zeigt sich auch sehr gut daran, dass die Werte der Feldgrößen 2048 und 8192 bei 256 ranks auf einen Punkt fallen. Dadurch wird deutlich, dass der Datenaustausch die Ausführungszeit dominiert und nicht das füllen der Felder mit Zufallswerten.
Beim Sprung von 64 auf 128 ranks findet Kommunikation zwischen den Sockets statt, was die Latenz er-

höht. Bei 256 ranks wird ein zweiter Node benötigt. Die Zeit steigt hier dramatisch an. Dieses Verhalten habe ich bereits in 3.2 erklärt.

4.2 Berechnung

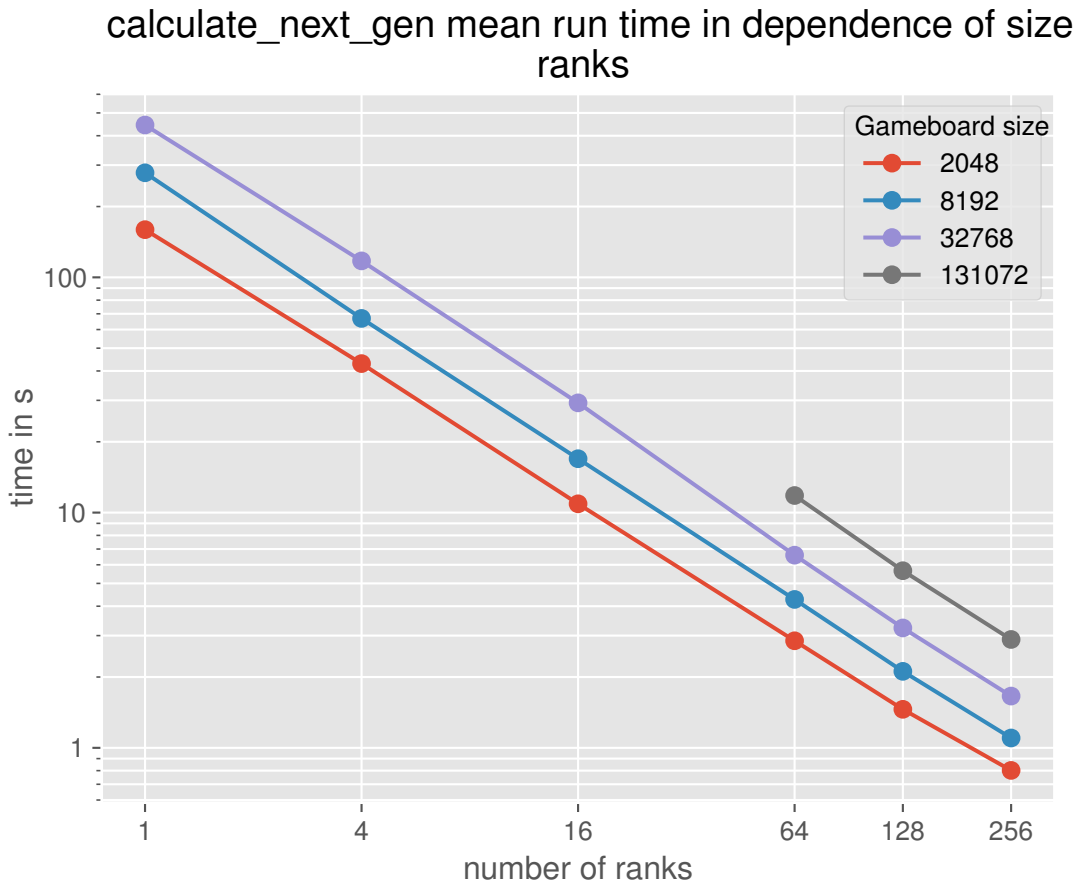

Abbildung 5: Logarithmische Darstellung der Ausführungszeit der Funktion `calculate_next_gen()`.

Auch hier sinkt die Ausführungszeit mit Anzahl der ranks ab. Es gibt jedoch kein Abknicken, wie noch bei der Initialisierung. Eine mögliche Begründung dafür ist, dass die Berechnung des inneren Feldes soviel Zeit in Anspruch nimmt, dass währenddessen die Kommunikation abgeschlossen wird. Damit verringert sich die Ausführungszeit ohne erkennbare Einbrüche.

4.3 Komplettes Programm

Hier betrachte ich die benötigte Zeit des Programms, inklusive spawnen der Prozesse die Initialisierung von MPI. In 3.2 diskutierte ich, wann Einbrüche zu erwarten sind. Es zeigt sich damit, dass die Ausführungszeit ab 16 ranks immer stärker vom Overhead von MPI dominiert wird. Erkennen lässt sich das daran, dass die Ausführungszeiten ab dieser Grenze stagnieren und anschließend wieder steigen. Gleichzeitig rücken die Feldgrößen immer näher zusammen. Das verdeutlicht, dass sie nicht mehr der dominierende Faktor sind.

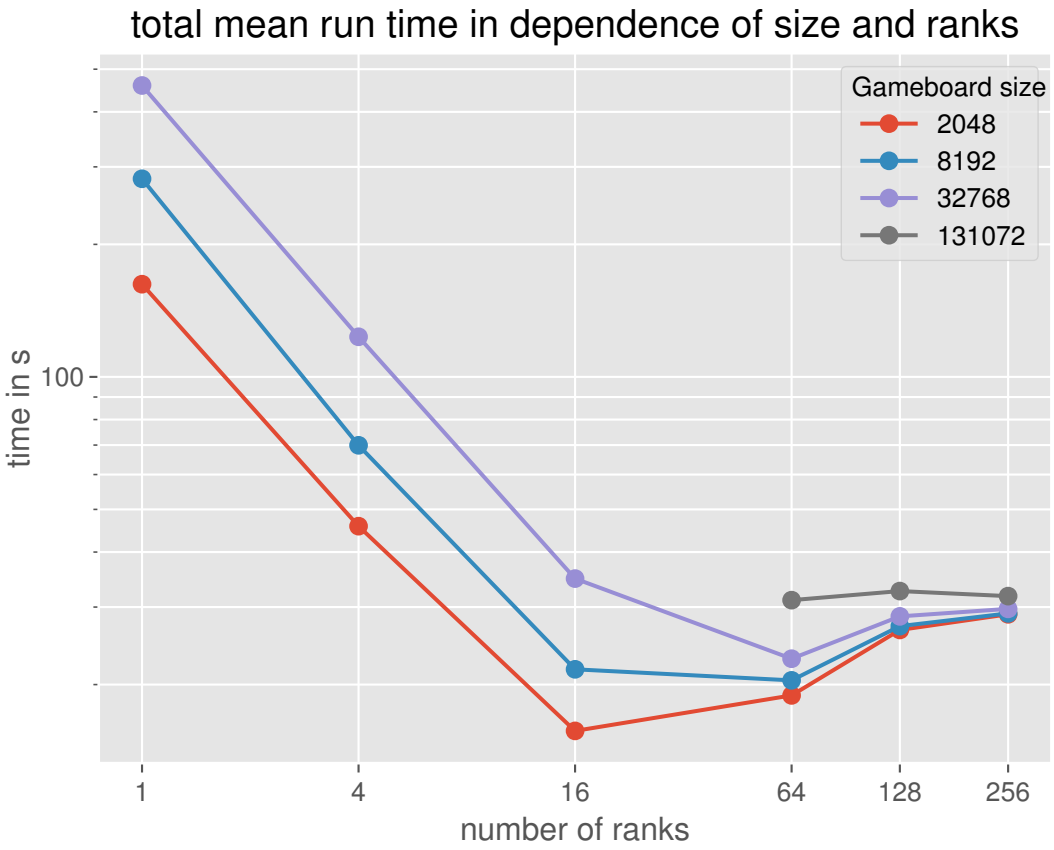

Abbildung 6: Logarithmische Darstellung der Ausführungszeit des Programms.

Subtrahiert man von der Ausführungszeit des kompletten Programms die Zeiten der Daten Initialisierung und Berechnung ergibt sich folgendes Bild.

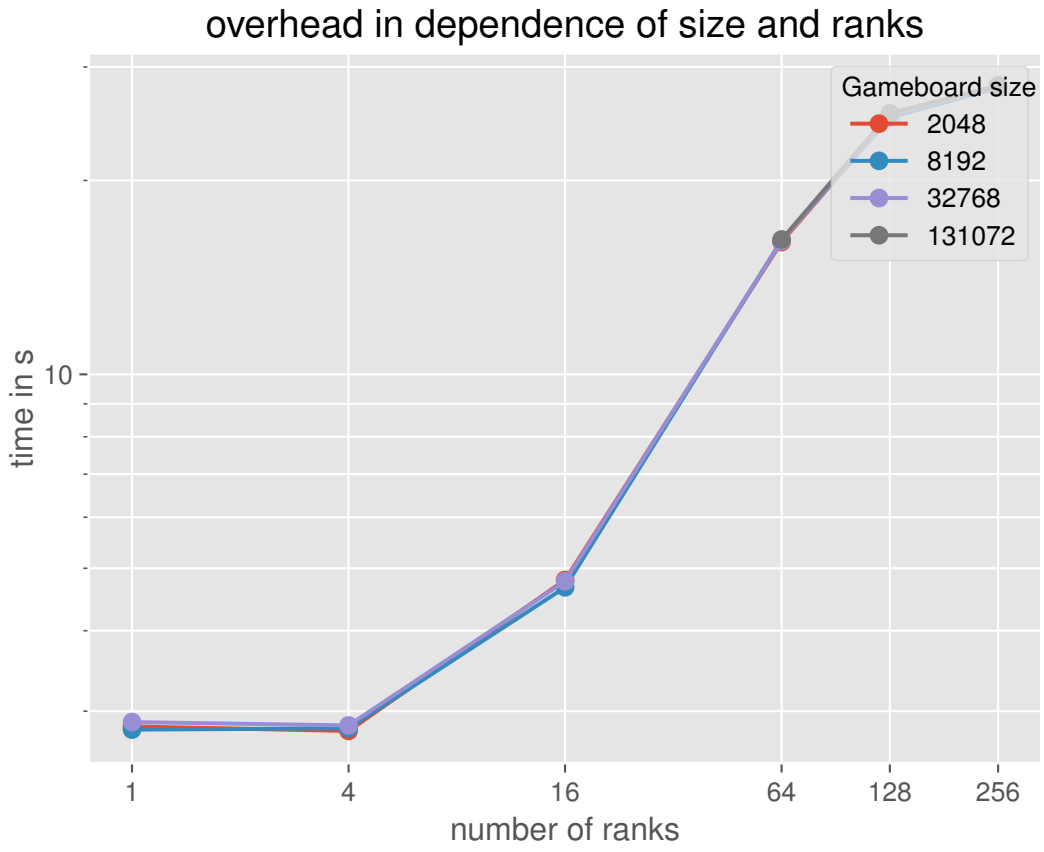

Abbildung 7: Logarithmische Darstellung der Ausführungszeit des Programms.

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

[AMD20] AMD. *HPC Tuning Guide for AMD EPYC 7002 Series Processors*. https://www.amd.com/system/files/documents/amd-epyc-7002-tg-hpc-56827.pdf. 2020

[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