TECHNISCHE UNIVERSITÄT DRESDEN

ZENTRUM FÜR INFORMATIONSDIENSTE
UND HOCHLEISTUNGSRECHNEN
PROF. DR. WOLFGANG E. NAGEL

Komplexpraktikum "Paralleles Rechnen" B - Thread-parallele Ausführung von Conways Game-of-Life

Bengt Lennicke

Dresden, 24. Januar 2024

Inhaltsverzeichnis

1 Aufgabenstellung

Implementieren Sie eine thread-parallele Variante von Conways 'Game-of-Life' mit periodic boundary conditions. Nutzen Sie dazu OpenMP Compiler-Direktiven. Benutzen Sie double buffering um Abhängigkeiten aufzulösen.

- Beschreiben Sie Ihren Ansatz und gehen Sie sicher, dass die Arbeit thread-parallel ausgeführt wird.
- Messen und Vergleichen Sie die Ausführungszeiten für 1,2,4,8,16 und 32 Threads, für den GCC, als auch den Intel Compiler bei Feldgrößen von 128x128, 512x512, 2048x2048, 8192x8192 und 32768x32768.
- Nutzen Sie für die Berechnung eine geeignete Anzahl an Schleifendurchläufen (Zyklen des Spiels), sodass der genutzte Timer genau genug ist.
- Nutzen Sie dafür die "romeo"Partition von taurus.
- Achten Sie darauf, dass benachbarte Threads möglichst nah einander gescheduled sind.
- Testen Sie für die Feldgröße 128x128, welchen Einfluss die OpenMP Schleifenschedulingverfahren haben (OMP_SCHEDULE), indem Sie für die Ausführung mit 32-Threads des mit Intel kompilierten Benchmarks die Verfahren static, dynamic, guided, und auto bei default chunk size vergleichen
- Führen Sie jeweils 20 Messungen durch und analysieren Sie die Ergebnisse mit geeigneten statistischen Mitteln.

1.1 Conways 'Game of Life'

Conways 'Game of Life' ist ein Gedankenspiel bei dem auf einem zweidimensionalen Spielbrett(board) Felder(cell) 'lebendig' oder 'tot' sind. Im Spielverlauf können die Felder 'lebendig werden', 'sterben', 'am Leben' bzw. 'Tot bleiben'. Das Aktualisieren basiert auf 4 Regeln:[2]

- Jede lebende Zelle mit weniger als 2 lebendigen Nachbarn stirbt (Unterpopulation)
- Jede lebende Zelle mit 2 oder 3 lebendigen Nachbarn bleibt am Leben
- Jede lebende Zelle mit mehr als 3 lebendigen Nachbarn stirbt (Überpopulation)
- Jede tote Zelle mit genau 3 lebendigen Nachbarn wird lebendig (Reproduktion)

Mit diesen einfachen Regeln und wiederholtem Aktualisieren des Boards bildet es ein komplexes System, welches mehrere Interpretionsweisen erlaubt.[2]
In diesem Bericht ist das Spiel selbst nicht von besonderem Interesse, sondern die Programmiertechnische Umsetzung der Aktualisierung des Boards unter der Verwendung von openMP.

2 Umsetzung

2.1 Spielsetup

Das Spielbrett ist als Struktur in 'gol_board.h' definiert.

```
1       #include <stdbool.h>
2       typedef struct
3       {
4               int rows;
```

```
5                   int cols;
6                   bool grid[0];
7           } board;
```

Die Felder des Boards werden in einem Array von Bools 'grid' festgehalten. Dieses eindimensionale Array wird dann durch das Festlegen von der Anzahl von Reihen 'rows' und Spalten 'cols' als ein zweidimensionales Brett definiert.
Das 'grid' wird zunächst als leeres Arrays deklariert, relevant ist hier nur, dass es auf ein Array von bools verwiesen wird. Das Allokieren von Speicher für ein Array in gewünschter Boardgröße passiert in 'init_board()'.

```
1  board* init_board(int rows, int cols, int start_cells)
2  {
3           board *b = calloc(1, 2*sizeof(int) + (rows * cols) * sizeof(bool));
4           b->rows = rows;
5           b->cols = cols;
6
7           if (start_cells)
8           {
9                   #include <math.h>
10                  int i;
11                  for ( i = 0; i < start_cells; i++)
12                  {
13                          set_state(b, random_int(0,b->cols),
14                          random_int(0,b->rows),
15                          1);
16                  }
17          }
18
19          // Glider
20          // 001
21          // 101
22          // 011
23          else
24          {
25                  set_state(b, 2, 0, 1);
26                  set_state(b, 0, 1, 1);
27                  set_state(b, 2, 1, 1);
28                  set_state(b, 1, 2, 1);
29                  set_state(b, 2, 2, 1);
30          }
31          return b;
32 }
```

Für die Struktur des Boards wird Speicher für die 2 Integer 'rows' und 'cols' sowie für alle Zellen allokiert und mit 0 initalisiert. Dadurch ist für das Setzen einer Startbelegung nur notwendig, dass einige Felder auf 'lebendig' d.h. auf 1 gesetzt werden müssen.
Die Anzahl der gewünschten lebendigen Zellen in der Startbelegung werden an die Funktion übergeben ('start_cells'). Wenn die Zahl 0 ist, wird ein sogenannter 'Glider' gesetzt. Dieses Objekt bewegt sich Diagonal über das Spielbrett und bietet eine gute Möglichkeit zu testen, ob die Regeln des Spiels korrekt implementiert sind. Wenn 'start_cells' ungleich 0 ist, wird dementsprechend oft ein zufälliges Feld gewählt, welches auf 1 gesetzt wird. Hierbei ist natürlich möglich, dass eine Zelle zweimal gewählt wird und dementsprechend weniger als 'start_cells' Zellen lebendig sind. Da für diesen Versuch nur wichtig

ist, dass ein Board mit einer Startbelegung existiert, ist das nicht wichtig.
Zum setzen des Status' der Zellen wird in dieser Funktion 'set_state()' verwendet.

```
1  void set_state (board *b, int x, int y, bool state)
2  {
3          coords_on_board(b, &x, &y);
4          b->grid[ y * b->cols + x ] = state;
5  }
```

Zunächst werden die Koordinaten auf das Spielbrett 'zugeschnitten'. In der Aufgabenstellung ist ein Spielbrett mit periodischen Randbedingungen gefordert, d.h. 'coords_on_board()' verändert x und y, sollten diese nicht zwischen 0 und 'cols' bzw. 0 und 'rows' liegen, sodass die Koordinaten wirder auf der Brett liegen.
Dadurch ist das Setzen der Zelle darunter nie ein Zugriff außerhalb des belegten Speichers. Es wird in das Array an der Stelle $y *$ b->cols $+ x$ geschrieben. Diese Rechnung ergibt sich daraus, dass x für die Spalte und y für die Reihe steht. Im eindimensionalen Arrays muss für einen Zugriff auf z.B. die 3. Reihe wird das Feld nach allen Elementen der ersten beiden Reihen gebraucht. Dementsprechend $2 *$ 'Anzahl der Elemente pro Reihe' $= 2 *$ b->cols.
Für die periodischen Randbedingungen wird in 'coords_on_board()' modulo Rechnung verwendet.

```
1  void coords_on_board(board *b, int *x, int *y)
2  {
3          if ( *x < 0 || *x >= b->cols)
4          {
5                  *x = ((*x % b->cols) + b->cols) % b->cols;
6          }
7          if ( *y < 0 || *y >= b->rows)
8          {
9                  *y = ((*y % b->rows) + b->rows) % b->rows;
10         }
11 }
```

Ziel ist hier, dass z.B. x=-1 auf b->cols-1 abgebildet wird, d.h. die Nachbarzellen vom rechten Rand die Zellen ganz Links sind; analog mit Oben und Unten. Der modulo Operator '%' in C soll folgende Gleichung erfüllen: $a == (a/b) * b + a\%b$. Das bedeutet es sind auch negative Lösungen möglich. So ist z.B. mit x=-1 und b->cols=300 das Ergebnis
$a\%b = a - (a/b) * b = -1 - (-1/300) * 300 = -1$,
weil die '/' Division in hier ganzzahlig teilt. Diese negative Zahl liegt dann im Interval [-b, b]. Die Zahlen im positiven Bereich werden anschließend durch $+b- > cols\%b- > cols$ nicht verändert. Die Zahlen im negativen Bereich werden dadurch wie gewünscht auf ihr positiven Counterpart abgebildet.
Der Status in der nächsten Iteration wird über die Funktion 'get_new_state()' bestimmt.

```
1  bool get_new_state(board *b, int x, int y)
2  {
3          int neighbours = get_num_neighbours(b, x, y);
4          if (check_state(b, x, y))
5          {
6                  if (neighbours < 2) return 0;
7                  if (neighbours > 3) return 0;
8          }
9          else
10         {
11                 if ( neighbours == 3 ) return 1;
12         }
```

```
13 }
```

Mit 'get_num_neighbours()' wird die Anzahl der lebendigen Nachbarn der Zelle bestimmt. Anschließend wird über Vergleiche ermittelt, ob der Status der Zelle 1 oder 0 (lebendig oder tot) seien soll.

```
1 int get_num_neighbours(board *b , int x, int y)
2 {
3         return
4         check_state(b, x-1, y-1) + check_state(b, x, y-1) + check_state(b, x+1,
5         check_state(b, x-1, y)   +                          check_state(b, x+1,
6         check_state(b, x-1, y+1) + check_state(b, x, y+1) + check_state(b, x+1,
7 }
```

Dafür wird von jedem Nachbarn der Status abgefragt und die Ergebnisse aufaddiert. Die Summe entspricht den lebenden Nachbarn. Die Funktion 'check_state()' funktioniert genau wie 'set_state()'.

2.2 Threaded Brett-Updates

3 Ausführung

3.1 Hardware

Die Messung für die Bearbeitung der Aufgaben sind auf dem CPU Cluster Romeo der TU Dresden ausgeführt worden. Dieser Cluster bietet 192 nodes mit jeweils [1]:

- 2 x AMD EPYC CPU 7702 (64 cores) @ 2.0 GHz, Multithreading möglich
- 512 GB RAM
- 200 GB SSD Speicher
- Betriebssystem: Rocky Linux 8.7

3.2 Programm-Versionen

Relevant für die Reproduzierbarkeit sind die Versionen der verwendeten Bibliotheken und Programme.

- GNU Make 4.2.1
- gcc (GCC) 10.3.0
- Python 3.9.5
  - numpy 1.24.1
  - pandas 2.0.0
  - matplotlib 3.3.4

3.3 Messung

4 Auswertung

Literatur

[1] HPC Compendium, 'HPC Resources', 12.01.2024
https://doc.zih.tu-dresden.de/jobs_and_resources/hardware_overview/#romeo

[2] Wikipedia Seite, 'Conways Spiel des Lebens', 22.01.2024
https://de.wikipedia.org/wiki/Conways_Spiel_des_Lebens#Die_Spielregeln