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, 26. 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 folglich 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. Das zufällige Wählen der Zellen basiert auf der 'rand()' Funktion von 'math.h'.
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 $+b\%b$ 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                  return 1;
9          }
10         else
11         {
```

```
12                  if ( neighbours == 3 ) return 1;
13                  return 0;
14          }
15  }
```

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. Für 'tote' Zellen wird nur überprüft, ob es 3 lebendige Nachbarn gibt, ansonsten ändert sich nichts. 'Lebendige' Zellen 'sterben' wenn sie weniger als 2 oder mehr als 3 Nachbarn haben. Hier wird zuerst mit 2 verglichen, da es wahrscheinlicher ist und in dem Fall der Vergleich '>3' gespart wird.

```
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  }
```

Für die Bestimmung der lebendingen Nachbarn wird von jeder anliegenden Zelle (inkl. diagonal) 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

Die Funktion 'update_board()' nimmt ein Board entgegen und führt eine Iteration des Spiels dafür aus.

```
1  void update_board(board *b)
2  {
3          int i, j;
4          board *buf;
5          buf = create_board_copy(b);
6          #pragma omp parallel for num_threads(THREADS) collapse(2)
7          for ( i = 0; i < b->rows; i++)
8          {
9                  for ( j = 0; j < b->cols; j++)
10                 {
11                         set_state(b, j, i, get_new_state(buf, j, i));
12                 }
13         }
14         free(buf);
15 }
```

Die Veränderung der einzelnen Zellen soll keinen Einfluss auf die Statusupdates anderer Zellen haben. Dementsprechend wird eine Kopie des Boards erstellt mit 'create_board_copy()'. In dieser Funktion wird mit 'malloc()' der notwendige Speicher allokiert und mit 'memcpy' dann darin die Kopie erschaffen. Anschließend werden die einzelnen Zellen iteriert. Es wird der Zustand einer Zelle für die nächste Iteration auf der Kopie ermittelt und die tatsächliche Änderung wird auf dem original Board gemacht.
Das Iterieren über die Zellen wird mit 'openMP' bzw. 'omp.h' gethreaded. Die Compiler Anweisung dafür ist in Zeile 6. Darin wird festgelegt, dass die folgenden 2 ('collapse(2)') for-loops ('for') in threads ausgeführt werden sollen. Die Anzahl der Threads ist mit 'num_threads(THREADS)' festgelegt auf den Wert vom Macro 'THREADS'. Das bietet die Möglichkeit beim Kompilieren die Anzahl der Threads festzulegen ohne den Quelltext anzupassen. Mehr dazu unter 2.4 und 2.5.

2.3 Implementation der Zeitmessung

Die Verwendung bzw. Messung der 'update_board()' Funktion ist in 'gol_main.c' definiert. Außerdem ist darin die 'main()' Funktion, also der Eintrittspunkt in die Messanwendung', definiert. In dieser Mainfunktion wird unterschieden ob das Programm mit einem Kommandozeilen Argument aufgerufen wurde oder nicht. Wenn nicht werden die Messungen für die unterschiedlichen Boardgrößen durchgeführt. Wenn mit Argument, dann ist es ein Messung für die unterschiedlichen 'OMP_SCHEDULE' Einstellungen und es wird nur eine analoge Messung am 128x128 Board gemacht.

```
1  // no cmd arg
2  if (argc==1)
3  {
4          struct times meas_times[5];
5          FILE *file;
6          int meas_num = 20;
7          int iterations[5] = {100,50,20,1,1};
8
9          char *path;
10         path = concat(concat("../evaluation/data/", COMPILER_STR),
11         concat("/", THREADS_STR));
12
13         //128x128
14         measure(128, meas_num, iterations[0], &meas_times[0]);
15         ...
16
17         // write results
18         file = fopen(concat(path,
19         concat("/","times128.csv")), "w");
20         write_times(file, meas_num, &meas_times[0], iterations[0]);
21         fclose(file);
22         ...
23
24         return 0;
25 }
```

Für jede Boardgröße werden 'meas_num=20' Messungen durchgeführt. Die Start- und Endzeiten der Messungen werden pro Boardgröße in einer 'times' Struktur gespeichert. Diese Beininhaltet dafür unteranderem einen Pointer auf ein Array an 'timespec' Objekten von 'time.h'. Die Anzahl der Schritte, welche für ein Board durchgeführt werden unterscheidet sich für die einzelnen Boardgrößen. Für 128x128 werden 100 Schritte, für 512x512 50 Schritte, für 2048x2048 20 Schritte und für die restlichen Größen lediglich ein Schritt. Für die geringeren Boardgrößen werden mehr Schritte berechnet, damit die Messzeit nicht in zu kleinen Zeitbereichen liegt, in denen die Messfunktionen möglicherweise unpräzise werden. Die Laufzeiten werden beim Schreiben in die Dateien durch die Anzahl der Schritte dividiert, um vergleichbare 'Laufzeit pro Boarditeration' zu bekommen.
Die Berechnung des ersten Schrittes braucht etwas länger als die restlichen, da noch mehr Zellen 'leben' und 'get_new_state()'2.1 benötigt länger für 'lebendige' Zellen. Daher könnten Messungen mit mehr Schritten bei allen Boardgrößen das Ergebnis noch verbessern. Zu Beginn werden 12,5% der Zellen auf 'lebend' gesetzt. Da dies zufällig passiert sind auch viele Zellen einfach und sterben direkt nach dem 1. Schritt, weshalb das Board direkt danach deutlich leerer ist.
Die Messzeiten werden mit 'clock_gettime(CLOCK_MONOTONIC, time);' aus 'time.h' bestimmt.
Das Schreiben der Ergebnisse wird erst im Anschluss an die Messungen durchgeführt, um die Messungen nicht durch etwaige Schreibprozesse zu beeinflussen.

2.4 run.sh

Das Shell-Skript 'run.sh' ermöglicht verschiedene Setups für eine Messung. So ist für 2.2 z.B. die Umgebungsvariable 'THREADS' notwendig für das Kompilieren und für den gesamten Versuch Messungen mit unterschiedlichen Vorraussetzungen notwendig.

```
1 #!/bin/sh
2
3 make clean
4 THREADS=$1 make $2
5 if [ -z $3 ]; then
6         OMP_PLACES=cores OMP_PROC_BIND=close ./gol_main
7 else
8         OMP_SCHEDULE=$3 OMP_PLACES=cores OMP_PROC_BIND=close ./gol_main $3
9 fi
```

Daher ist es über 3 Kommandozeilen Argumente möglich mit unterschiedlicher Anzahl an Threads und mit gcc oder icc zu Kompilieren. Desweiteren kann die Messung zusätzlich mit oder ohne 'OMP_SCHEDULE' Einstellungen gestartet werden. Das Skript ermöglich desweiteren ein einfaches Starten mit den Variablen 'OMP_PLACES=cores' und 'OMP_PROC_BIND=close', wodurch die Threads möglichst 'nah' beieinander laufen. Dadurch werden Zugriffszeiten auf gemeinsamen Speicher minimiert.
Durch dieses Implementation ist es möglich alle in der Aufgabenstellung geforderten Messungen in einem sbatch-Skript zu starten ohne den Quelltext oder ähnliches anpassen zu müssen. Genaueres dazu in 2.6.

2.5 Makefile

Die Messanwendung wird mit 'make' gebaut und der GNU und Intel Compiler (gcc/icc) wird verwendet. Dafür ist im folgenden Makefile der Ablauf definiert.

```
1  gcc: gcc_gol_main
2  icc: icc_gol_main
3
4  threads:
5          if [ -z "${THREADS}" ]; then \
6                  echo "THREADS is not defined."; \
7                  exit 1; \
8          fi
9
10 gcc_gol_board.o: threads gol_board.c
11         gcc -fopenmp -D THREADS=${THREADS} -c gol_board.c
12
13 gcc_visualize.o: visualize.c
14         gcc -c visualize.c
15
16 gcc_gol_main.o: gol_main.c
17         gcc -D THREADS=${THREADS} -D COMPILER=gcc -c gol_main.c
18
19 gcc_gol_main: gcc_gol_main.o gcc_visualize.o gcc_gol_board.o
20         gcc -fopenmp -o gol_main gol_main.o visualize.o gol_board.o
21
22
23 icc_gol_board.o: threads gol_board.c
24         icc -fopenmp -D THREADS=$(THREADS) -c gol_board.c
```

```
25
26 icc_visualize.o: visualize.c
27         icc -c visualize.c
28
29 icc_gol_main.o: gol_main.c
30         icc -D THREADS=${THREADS} -D COMPILER=icc -c gol_main.c
31
32 icc_gol_main: icc_gol_main.o icc_visualize.o icc_gol_board.o
33         icc -fopenmp -o gol_main gol_main.o visualize.o gol_board.o
34
35 clean:
36         rm gol_board.o visualize.o gol_main.o gol_main pbms/test* pbms/*.gif pbm
```

Das Makefile unterteilt sich in 2 Teile: einen für den GNU Compiler und einen für den Intel Compiler. Durch jeweils 'make gcc' oder 'make icc' wird der dementsprechende Compiler verwendet.
Abgesehen vom Kompiler selbst und dem gesetzen 'COMPILER' Macro sind die beiden Bauprozesse analog. Für die Verwendung von openMP bzw. 'omp.h' wird der Tag '-fopenmp' beim Kompilieren von 'gol_board.c' gesetzt. Außerdem werden wie bereits erwähnt die Macros 'THREADS' und 'COMPILER' mit '-D' gesetzt. Dadurch kann das Programm für verschiedene Anzahlen an Threads Kompiliert werden ohne, dass der Quelltext angepasst werden muss. Desweitern werden beide Macros im Programm verwendet für den Pfad zu den Dateien für die Messergebnisse.

2.6 slurm.sh

Die Messungen werden auf dem 'romeo' Cluster der TU Dresden durchgeführt. Die Prozesse dafür werden von 'slurm' mit dem Starten eines 'sbatch' Skripts gestartet. Das folgende Skript lässt sequentiell die Messungen für die verschiedenen Threadzahlen, Compiler und 'OMP_SCHEDULE'-Einstellungen laufen.

```
1  #!/bin/bash
2
3  #SBATCH --ntasks=16
4  #SBATCH --time=04:30:00
5  #SBATCH --account=p_lv_kp_wise2324
6  #SBATCH --job-name=kp_pr_aufgabe_b
7  #SBATCH --output=kp_pr_aufgabe_a_%j.out
8  #SBATCH --error=kp_pr_aufgabe_a_%j.err
9
10 # make sure run.sh is executable
11 chmod u+x run.sh
12
13 module load GCCcore/10.3.0
14 make clean
15 # Thread-Measurements
16 srun --exclusive -n 1 --cpu-freq=2000000 ./run.sh 1 gcc
17 srun --exclusive -n 1 --cpu-freq=2000000 ./run.sh 2 gcc
18 ...
19
20 module load intel-compilers
21 make clean
22 srun --exclusive -n 1 --cpu-freq=2000000 ./run.sh 1 icc
23 srun --exclusive -n 1 --cpu-freq=2000000 ./run.sh 2 icc
24 ...
```

```
25
26 # OMP_SCHEDULE measurements
27 srun --exclusive -n 1 --cpu-freq=2000000 ./run.sh 32 icc static
28 srun --exclusive -n 1 --cpu-freq=2000000 ./run.sh 32 icc guided
29 srun --exclusive -n 1 --cpu-freq=2000000 ./run.sh 32 icc dynamic
30 srun --exclusive -n 1 --cpu-freq=2000000 ./run.sh 32 icc auto
```

Dafür wird jeweils mit srun ein Prozess auf einer CPU mit 2GHz gestartet. Die 2GHz sind über '–cpu-freq' festgelegt. Für den Prozess wird das in 2.4 beschriebene Shell Skript gestartet mit unterschiedlichen Argumenten. Der Tag '–exclusive' wurde lediglich für die finale Messung verwendet, um sicherzustellen, dass andere Programme nicht die Ausführungszeiten beeinflussen.

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
- icc (ICC) 2021.7.1 20221019
- 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