Habib Mubarak, Eraycan Gökcan

Aufgabenblatt 6

Aufgabe 6.1

i)
Nein, es würde wahrscheinlich keinen Sinn machen, für diese Schleife Multi-Threading zu verwenden. Die Anzahl der Iterationen ist sehr gering (nur zwei), und jede Ausführung ist sehr schnell. In einem solchen Fall kann der Overhead, der durch das Erstellen von Threads und die Synchronisation entsteht, größer sein als der potenzielle Geschwindigkeitsvorteil durch parallele Ausführung.

ii)
Concurrency bedeutet, dass mehrere Aufgaben “gleichzeitig” verwaltet werden, aber nicht wortwörtlich gleichzeitig ausgeführt werden. Es wird zwischen den Aufgaben gewechselt. Beispiel: Ein Koch bereitet mehrere Gerichte “gleichzeitig” zu.

Parallelismus bedeutet, dass mehrere Aufgaben wortwörtlich gleichzeitig ausgeführt werden. Beispiel: Mehrere Köche bereiten jeweils ein Gericht / mehrere Gerichte zu.

Beispiel aus der realen Welt:

Es soll auf jeden Frame einer Videodatei ein Filter angewendet werden. Die Frames werden in Chunks für alle CPU-Kerne aufgeteilt. Die Aufgabe ist jedoch nicht perfekt parallelisiert, weil:

- Nach der Bearbeitung müssen alle Videosegmente wieder korrekt zusammengesetzt werden, was sequentiell abläuft.
- Manche Videosegmente benötigen eventuell mehr Rechenzeit als andere, wodurch Kerne unterschiedlich lange beschäftigt sind.

iii)

In C++ kann es zu unvorhersehbarem Verhalten kommen, wenn voneinander abhängige Teilaufgaben parallel von mehreren Threads ausgeführt werden, da diese Threads möglicherweise gleichzeitig auf gemeinsame Daten zugreifen und diese verändern, wodurch Logikfehler entstehen können.

Wenn Threads gleichzeitig auf gemeinsame Daten zugreifen, treten Probleme auf, wenn Schreibzugriffe beteiligt sind. Diese Situation wird als Race Condition bezeichnet.

Zum Beispiel könnte ein Thread versuchen, Daten zu lesen, bevor ein anderer Thread diese vollständig geschrieben oder aktualisiert hat, was zu falschen oder inkonsistenten Ergebnissen führt.

iv)

- Jedem der n Elemente wird ein eigener Thread zugewiesen
- In der ersten Iteration addieren je zwei Threads ihre Elemente paarweise. Es entstehen n/2 Teilsummen. (Zeit O(1), denn parallel).
- Allgemein addieren in der k-ten Iteration je zwei Threads ihre Teilsummen paarweise. Es entstehen n/(2^k) Teilsummen.
- In der letzten Iteration ist k = log_2(n), denn am Ende entsteht n/(2^log_2(n)) = n / n = 1 (Teil-)Summe.
- Jede Iteration benötigt O(1) Zeit und es gibt log_2(n) Iterationen

=> Gesamtlaufzeit O(log_2(n))

v)

CPU Model: Intel(R) Core(TM) i5-9400 CPU @ 2.90GHz

Total Supported Threads: 6 threads

Caches (sum of all):

L1d:	192 KiB (6 instances)
L1i:	192 KiB (6 instances)
L2:	1.5 MiB (6 instances)
L3:	9 MiB (1 instance)

Aufgabe 6.2

i)

```
 Hello, world!  Hello, world! 42
 Hello, world! 1
 Hello, world! 6
 Hello, world!  Hello, world! 8
5
 Hello, world! 0
 Hello, world! 3

 Hello, world!  Hello, world! 7
9
```

HelloWorld.out

Die Ausgabe zeigt, dass die Zeilen nicht sauber getrennt sind, und teilweise ineinander übergehen oder in ungewöhnlicher Reihenfolge erscheinen. Das liegt daran, dass alle Threads gleichzeitig auf die Konsole schreiben. Es kommt dadurch Race Conditions beim Schreiben in die Konsole.

Beim erneuten Ausführen ohne erneutes Kompilieren kommen unterschiedliche Ergebnisse:

```
 Hello, world! 1
 Hello, world! 2
 Hello, world!  Hello, world! 9
4
 Hello, world! 5
 Hello, world! 8
 Hello, world! 6
 Hello, world!  Hello, world!  Hello, world! 7
0
3
```

HelloWorld.out

ii)

```
 Hello, world! 3
 Hello, world! 1
 Hello, world! 6
 Hello, world! 0
 Hello, world! 9
 Hello, world! 2
 Hello, world! 4
 Hello, world! 5
 Hello, world! 7
 Hello, world! 8
```

HelloWorld.out

Hier erscheinen die Zeilen nur in ungewöhnlicher Reihenfolge, aber sie sind jetzt sauber getrennt. Durch “critical” wird die Ausgabe in die Konsole nur von jeweils einem Thread gleichzeitig ausgeführt.

Aufgabe 6.3

bug1.cpp:

Der Bug hier ist, dass man private verwendet hat und nicht firstprivate. Mit private kreiert jeder Thread eine Kopie von n, aber nicht initialisiert! Man möchte aber, dass für jeden Thread n = 1000 ist. Also kann man stattdessen firstprivate verwenden. firstprivate initialisiert n von jedem Thread mit dem Wert vom Master-Thread, also 1000.

bug2.cpp:

Die Variable tmp wird von mehreren Threads gleichzeitig verwendet.

Der Wert von tmp hängt in jedem Schleifendurchlauf vom vorherigen ab(tmp += i). Das ist entweder nicht unabhängig parallelisierbar oder ist sehr komplex zu parallelisieren. Jedoch ist `output_scalar[i] = static_cast<float>(tmp) / i` parallelisierbar, solang man den Wert für tmp schon weiß.

Wir können die tmp-Werte für alle Schleifendurchläufe erstmal sequentiell berechnen und nur den zweiten Teil parallelisieren. So bleibt die Korrektheit enthalten.

bug3.cpp:

Die nowait-Anweisung führt dazu, dass die Threads nach der (ersten) for-Schleife nicht auf die anderen Threads wartet (kein Synchronisation). Sodurch passiert höchstwahrscheinlich, dass mindestens ein Thread in der zweiten for-Schleife mit einem alten sum-Wert arbeitet und sodurch falsche Ergebnisse rauskommen.

Wenn man nowait entfernt, findet Synchronisation statt nach der ersten for-Schleife. Somit arbeiten dann alle Threads mit dem richtigen sum-Wert danach.

bug4.cpp:

Damit jeder Thread am Ende den endgültigen sum-Wert hat in der letzten for-Schleife, muss davor eine Synchronisation stattfinden. Jedoch gibt es nach atomic keinen impliziten Barrier, deshalb muss man explizit den Barrier hinzufügen.

Aufgabe 6.4

```
Number of steps:         100000000
Number of threads:       4

--- Results ---
Sequential:
  Value:     3.14159
  Runtime:   0.11856 seconds

Thread-local sums method:
  Value:     3.14159
  Runtime:   0.02796 seconds
  Speedup:   4.24001

OpenMP reduction method:
  Value:     3.14159
  Runtime:   0.02776 seconds
  Speedup:   4.27005

[PASSED] Thread-local sums method matches sequential result.
[PASSED] OpenMP reduction method matches sequential result.
```

Pi.out

Der erwartete Speedup für die Thread-local sums Methode und OpenMP reduction Methode ist jeweils 4x, da wir mit 4 Threads die Berechnung parallelisieren. Tatsächlich sind die Ergebnisse auch so. Zwischen den 2 Methoden werden keine großen Unterschiede erwartet, denn im Wesentlichen passiert die Parallelisierung in gleicher Art und Weise in beiden Methoden.

Aufgabe 6.5

```
Runtime Scalar:  79.68807 ms
Runtime Vc:      20.42699 ms (Speedup: 3.90112)
Runtime OpenMP:  8.01897 ms (Speedup: 9.93744)

SIMD and scalar results are the same.
OpenMP+SIMD and scalar results are the same.
```

Matrix.out

Der erwartete Speedup für die OpenMP+SIMD Methode ist 16x, da wir mit 4 Threads die Berechnung parallelisieren und 4 Floats innerhalb eines SIMD-Vektors haben. Jedoch haben wir hier nur einen Speedup von ungefähr 10x. Das kann an einigen Gründen liegen:

- OpenMP verursacht Overhead beim Verwalten der Threads. Auch die Vc-Library verursacht eventuell zusätzllichen Overhead.
- Sowohl SIMD als auch Multithreading führen zu einem höheren Bedarf an Speicherbandbreite. Wenn der Speicher die Daten nicht zügig genug bereitstellt, warten die Recheneinheiten der CPU auf den Speicher.
- Mehrere Threads können auf Speicheradressen zugreifen, die im selben Cache-Line liegen. Selbst wenn sie unterschiedliche und unabhängige Daten bearbeiten, führt das zu unnötigem Datenaustausch zwischen den CPU-Caches, was die Performance beeinträchtigen kann.

Aufgabe 6.6

Für alle Resultate war der Code im Release-Modus und mit Optimierungslevel O3 kompiliert. Getestet wurde mit der p67_snark_loop.txt-Datei im save-Ordner.

```
// Ergebnisse für Implementierung mit evolve():
Simulation für 100000 Generationen.
Gesamtdauer: 12901 Millisekunden.
```

GameOfLife.exe

```
// Ergebnisse für Implementierung mit evolve_parallel():
Simulation für 100000 Generationen.
Gesamtdauer: 4128 Millisekunden.
```

GameOfLife.exe

Der erwartete Speedup für die Implementierung mit evolve_parallel() ist 4x, da wir mit 4 Threads die Berechnung parallelisieren. Wir sehen hier einen Speedup von 3x. Das kann wieder an dem Overhead beim Verwalten der Threads und/oder an dem Cache-Line-Grund von davor liegen.