Parallel ohne Locks: Was gilt in .NET und was in Java?

Luc Bläser

Hochschule für Technik Rapperswil

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

HSR Concurrency Lab
Prof. Dr. Luc Bläser

Parallel 2018 Konferenz
7. März 2018, Heidelberg

Ausgangslage

```
int count = 0;
foreach (var number in array) {
    if (IsPrime(number)) {
      count++;
    }
}
```

Parallelisieren!

Parallelisierung

```
int count = 0;
Parallel.ForEach(array, number => {
    if (IsPrime(number)) {
      count++;
    }
});
```

Korrekt?

Fehler

```
int count = 0;
Parallel.ForEach(array, number => {
    if (IsPrime(number)) {
      count++;
    }
});
```

Race Condition und Data Race

Falsche Resultate möglich

Race Condition

- Mehrere Threads greifen auf gemeinsame Ressource ohne genügende Synchronisation zu
- Mögliches Fehlverhalten oder falsche Resultate
  - Nicht-deterministisches Auftreten!

Ursache ist oft ein Data Race, aber nicht immer!

Data Race

- Unsynchronisierte nebenläufige Speicherzugriffe
  - Selbe Variable oder Array Element
  - Mindestens ein schreibender Zugriff
- Formaler Programmierfehler

Mehrere Threads

`count++`

Korrektur mit Locks

```
int count = 0;
Parallel.ForEach(array, number => {
   if (IsPrime(number)) {
     lock(sync) {
       count++;
     }
   }
});
```

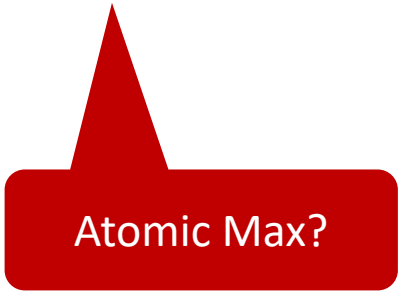
Lock ist relativ teuer

Korrektur mit Atomics

```
int count = 0;
Parallel.ForEach(array, number => {
    if (IsPrime(number)) {
      Interlocked.Increment(ref count);
    }
});
```

Komplizierterer Fall

```
int max = 0;
Parallel.ForEach(array, number => {
    if (IsPrime(number)) {
      lock(sync) {
        max = Math.Max(number, max);
      }
    }
});
```



Lock-freie Programmierung

- Ziel: Nebenläufig korrekt ohne Locks
  - Locks sind teuer
  - Risiko von Deadlocks
- Zwei Varianten
  - Gar keine Synchronisation
  - Synchronisation ohne Locks

Wo ist keine Synchronisation nötig?

- Read-Only Verwendung
- Separate Speicherzugriffe pro Thread

- Oder indirekt: Einbettung in thread-sichere Struktur

Korrekt ohne Synchronisation

```
Parallel.For(0, input.Length - 1, index => {
    result[index] = IsPrime(input[index]);
});
```

Indirekt: Ohne explizite Synchronisation

- Z.B. Mit Functional Style / LINQ

```
array.AsParallel().Where(IsPrime).Max()
```

- Oder Concurrent Collection

```
var bag = new ConcurrentBag<int>();
Parallel.ForEach(array, number => {
    if (IsPrime(number)) {
      bag.Add(number);
    }
});
```

Synchronisation ohne Locks

- Atomare Operationen
  - Z.B. Interlocked in .NET, Atomic-Objekte in Java
- Memory Barriers
  - Z.B. mit volatile

Kenntnis des Speichermodells unabdingbar

Memory Model

- Nebenläufige Zugriffe auf gemeinsamen Speicher
  - Selbe Variablen oder Array-Elemente
- Welche Garantien bzw. Nicht-Garantien gelten?
  - Weak Consistency
- Sprachspezifikation ist auschlaggebend!
  - Java: Language Specification
  - .NET: CLI Ecma Standard

Garantien bzw. Nicht-Garantien

- Atomarität
  - Welcher Code wird atomar ausgeführt?
- Sichtbarkeit
  - Sehen Threads Änderungen untereinander?
- Ordnung
  - Wird die Reihenfolge von Code eingehalten?

Generell nicht atomar

```
count++
```

Lesen in Register
Inkrementieren in Reg.
Schreiben in Speicher

```
if (x > max) {
   max = x;
}
```

Vergleich und Zuweisung nicht atomar

Garantierte Atomarität

- Einzelnes Lesen bzw. einzelnes Schreiben von Referenzen oder Datentypen mit höchstens 32 Bit
  - byte, short, int, float
- Nur in Java: Mit volatile auch 64 Bit Typen
  - long und double
- Spezielle atomare Operationen

Atomare Operationen

- Addieren, Inkrement, Exchange (Java getAndSet)
- CompareExchange (Java testAndSet)

```
Interlocked.CompareExchange(ref T x, T y, T z)
```

⇔

atomic {
  T temp = x;
  if (x == z) x = y;
  return temp;
}

Nicht garantiert sichtbar

```
bool stop = false;
```

Thread 1

```
stop = true;
```

Thread 2

```
while (!stop) {}
```

```
load stop in reg1;
while (!reg1) {}
```

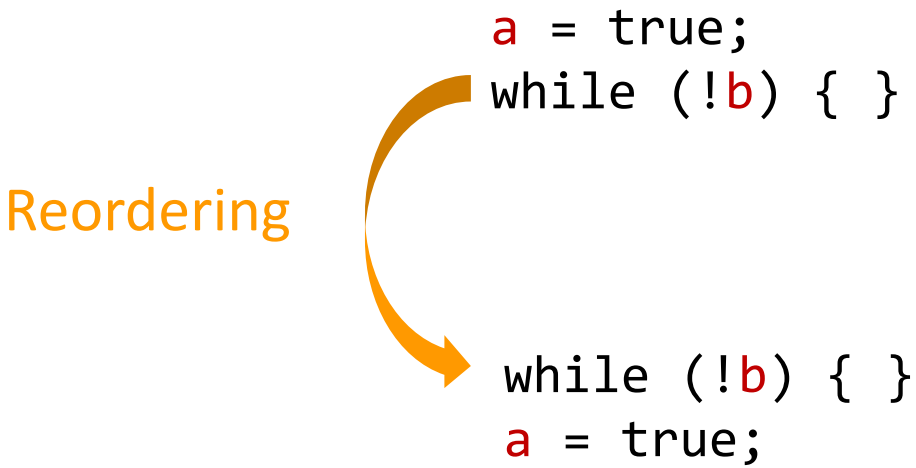
Endlosschlaufe möglich

Garantierte Sichtbarkeit

- In Java spezifiziert
  - Locks: Änderungen vor Release sind bei Acquire deselben Locks sichtbar
  - Volatile-Write macht Updates Lese-Zugreifer derselben Variable sichtbar
  - Thread-/Task-Start und Join
  - final Variablen nach Konstruktor
- In .NET nicht spezifiziert
  - Jedoch implizit über Ordnung definiert

Mögliche Umordnung

- Code darf prinzipiell umgeordnet werden, solange Semantik aus Sicht einzelner Threads gilt (as-if-serial)
- Durch Compiler, Laufzeitsystem und Prozessor



Garantierte Ordnung

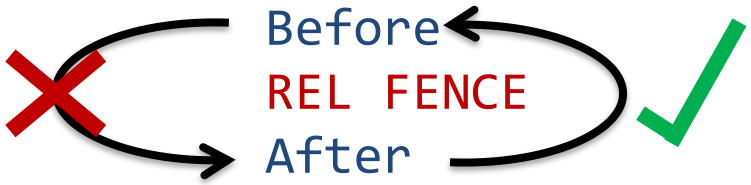
- Bestimmte einseitige Ordnung (Half Fences) bei
  - Lock Acquire/Release
  - Atomare Operationen
  - Volatile Zugriffe
  - Thread/Task Start & Join
- Explizite .NET MemoryBarrier als Full Fence
- Nur Java: Synchronisationen werden untereinander nicht umgeordnet

Half Fences

- Unterscheidung in Acquire und Release Fence
- Acquire Fence (load fence): Bleibt davor
- Release Fence (store fence): Bleibt danach

Acquire Fence

Release Fence



Acquire Fences

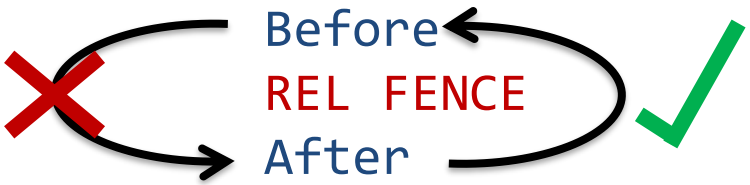
- Bleibt davor
  - Lock Acquire
  - Atomare Operationen
  - Volatile Read
  - Thread/Task Start

Acquire Fence

Release Fence

- Bleibt danach
  - Lock Release
  - Atomare Operationen
  - Volatile Write
  - Thread/Task Join

Release Fence



Volatile für Memory Barrieren

- Verschiedene Semantiken je nach Sprache
- Java: Half Fence + Synchronization Order
- .NET: Nur Half Fence
- C/C++: Keine Fence

```
volatile int x;
```

Volatile Read

```
… = x;
```

Volatile Write

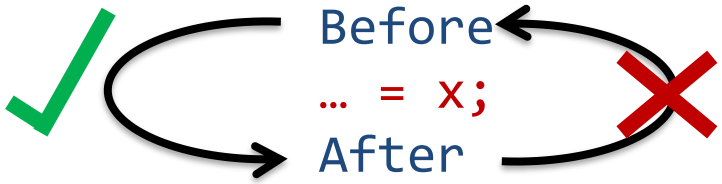
```
x = …;
```

Volatile = Half Fence

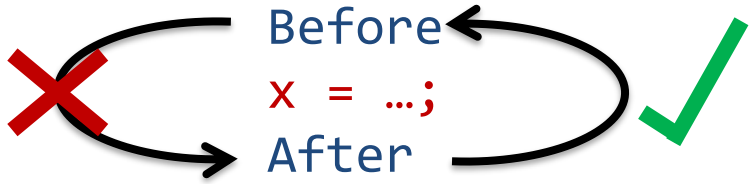
- Volatiles Lesen («Acquire»): Bleibt davor
- Volatiles Schreiben («Release»): Bleibt danach
- Kein Data Race mehr

```
volatile int x;
```

Volatile Read



Volatile Write



Java: Synchronization Order

- Folgende Operationen werden untereinander nie umgeordnet
  - Lock, Unlock
  - Volatile Zugriffe
  - Atomare Operationen
  - Thread/Task Start/Joins

Zweier-Barriere

```
volatile bool a = false, b = false;
```

Thread 1

```
a = true;
while (!b) { }
```

Thread 2

```
b = true;
while (!a) { }
```

✓ **Korrekt in Java**
(wegen Synchronization Order)

(jedoch Spin Waits)

Zweier-Barriere

```
volatile bool a = false, b = false;
```

Thread 1

```
a = true;
while (!b) { }
```

Thread 2

```
b = true;
while (!a) { }
```

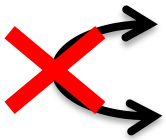
Umordnungen der Half-Fences möglich
Keine Synchronization Order in .NET

Falsch in .NET

Korrektur speziell in .NET

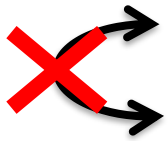
```
volatile bool a = false, b = false;
```

Thread 1



```
a = true;
Thread.MemoryBarrier();
while (!b) { }
```

Thread 2



```
b = true;
Thread.MemoryBarrier();
while (!a) { }
```

Thread.MemoryBarrier als Full Fence

Maximum ohne Locks?

```
lock(sync) {
  max = Math.Max(number, max);
}
```

Fehlversuch zum ersten

```
if (number > max) {
  max = number;
}
```

Unter anderem Data Races!

Falsch

Fehlversuch zum zweiten

```
if (number > Volatile.Read(ref max)) {
  Volatile.Write(ref max, number);
}
```

If und Setzen sind nicht atomar => Race Condition!

Immer noch falsch

Korrekte Lösung

```
bool success = false;
do  {
  int old = Volatile.Read(ref max);
  int value = Math.Max(number, old);
  success = CompareExchange(ref max, value, old) == old;
} while (!success);
```

Lesen mit Fence

Schreibe, nur falls noch gleich ist

Wiederhole bei Konflikt

Ineffizient bei Contention
Starvation ist möglich

Leichte Optimierung

```
while (true) {
  int old = Volatile.Read(ref max);
  if (number <= old ||
      CompareExchange(ref max, number, old) == old) {
    break;
  }
}
```

Aufhören, falls zu klein

Wiederholte Fehlversuche bleiben möglich

Schlussfolgerungen

- Lock-freie Programmierung ist diffizil
  - Schnell komplex, fehlerhaft oder ineffizient
  - Meist unnötig für Anwendungssoftware
- Explizite Synchronisation kann man aber reduzieren
  - Immutability / Read-Only
  - Disjunkte Thread-Zugriffe
  - Functional Style, Parallel LINQ, Reactive
  - Vorfabrizierte Concurrent Collections
- Java und .NET Memory Modell
  - Sind zwar ähnlich, aber z.T. wenig anders

Danke für Ihre Aufmerksamkeit

- Kontakt
  - **Prof. Dr. Luc Bläser**
    HSR Hochschule für Technik Rapperswil
    lblaeser@hsr.ch
  - **HSR Concurrency Lab**
    http://concurrency.ch