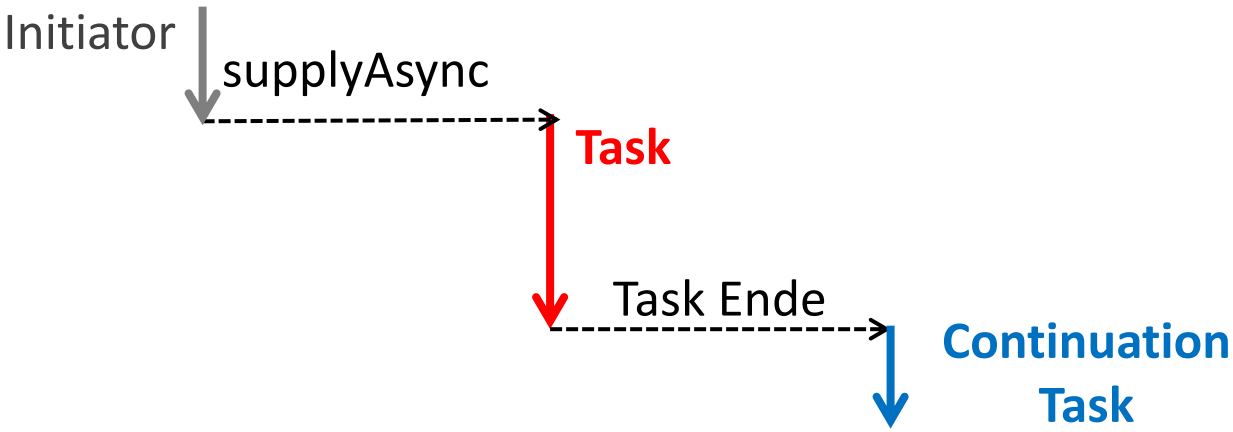
Parallele Programmierung
Task Parallel Library

Vorlesung 6

Prof. Dr. Luc Bläser

Quiz – Letzte Woche

```
public static void main(String[] args) {
  CompletableFuture.supplyAsync(() -> calculate()).
  thenAcceptAsync(System.out::println);
}
```



Was ist das Problem?

Technologiewechsel

- Von Java zu .NET
  - C# als Sprache
  - .NET Framework
  - Visual Studio oder VS Code als IDE
- .NET Task Parallel Library
  - Moderner Thread Pool

Inhalt Heute

- Threading in .NET
  - Unterschiede zu Java
- Task Parallel Library
  - Task Parallelität
  - Datenparallelität
  - Asynchrone Programmierung

Lernziele

- Unterschiede des parallelen Programmierens zwischen Java und .NET kennen
- Konzepte und Abstraktionsstufen der Task Parallel Library verstehen
- Task Parallel Library einsetzen können

.NET Threads

- Keine Vererbung: Delegate bei Konstruktor
- Exception in Thread => Abbruch des Programms

```
var myThread = new Thread(() => {
  for (int i = 0; i < 100; i++) {
     Console.WriteLine("MyThread step {0}", i);
  }
});
myThread.Start();
…
myThread.Join();
```

C# Lambdas

- Syntax fast gleich wie Java
  - (Parameterliste) => { Statement Sequence }
  - (Parameterliste) => Expression
- Lambda kann umgebende Variablen zugreifen
  - Auch schreibend

```
bool finished = false;
Thread myThread = new Thread(() => {
  …
  finished = true;
});
```

Prädestiniert für Data Races
sogar auf lokalen Variablen

Monitor in .NET

```
class BankAccount {
  private decimal balance;
  private object syncObject = new();

  public void Withdraw(decimal amount) {
    lock (syncObject) {
      while (amount > balance) {
        Monitor.Wait(syncObject);
      }
      balance -= amount;
    }
  }

  public void Deposit(decimal amount) {
    lock (syncObject) {
      balance += amount;
      Monitor.PulseAll(syncObject);
    }
  }
}
```

Monitor auf Hilfsobjekt als Best Practice

Analog zu synchronized Statement

Schlaufe auch notwendig

Analog zu notifyAll()

.NET Monitor

- FIFO Warteschlange
  - Pulse informiert längst Wartenden
- Wait() in Schlaufe
  - Zwar kein Spurious Wakeup
  - Gleiche Probleme des Überholens (Signal and Continue)
- PulseAll() bei mehreren Bedingungen oder Erfüllungen mehrerer Threads
  - Analog zu Java Monitor
- Synchronisation mit Hilfsobjekt als Best Practice
  - Nicht zwingend: Gründe dafür und dagegen

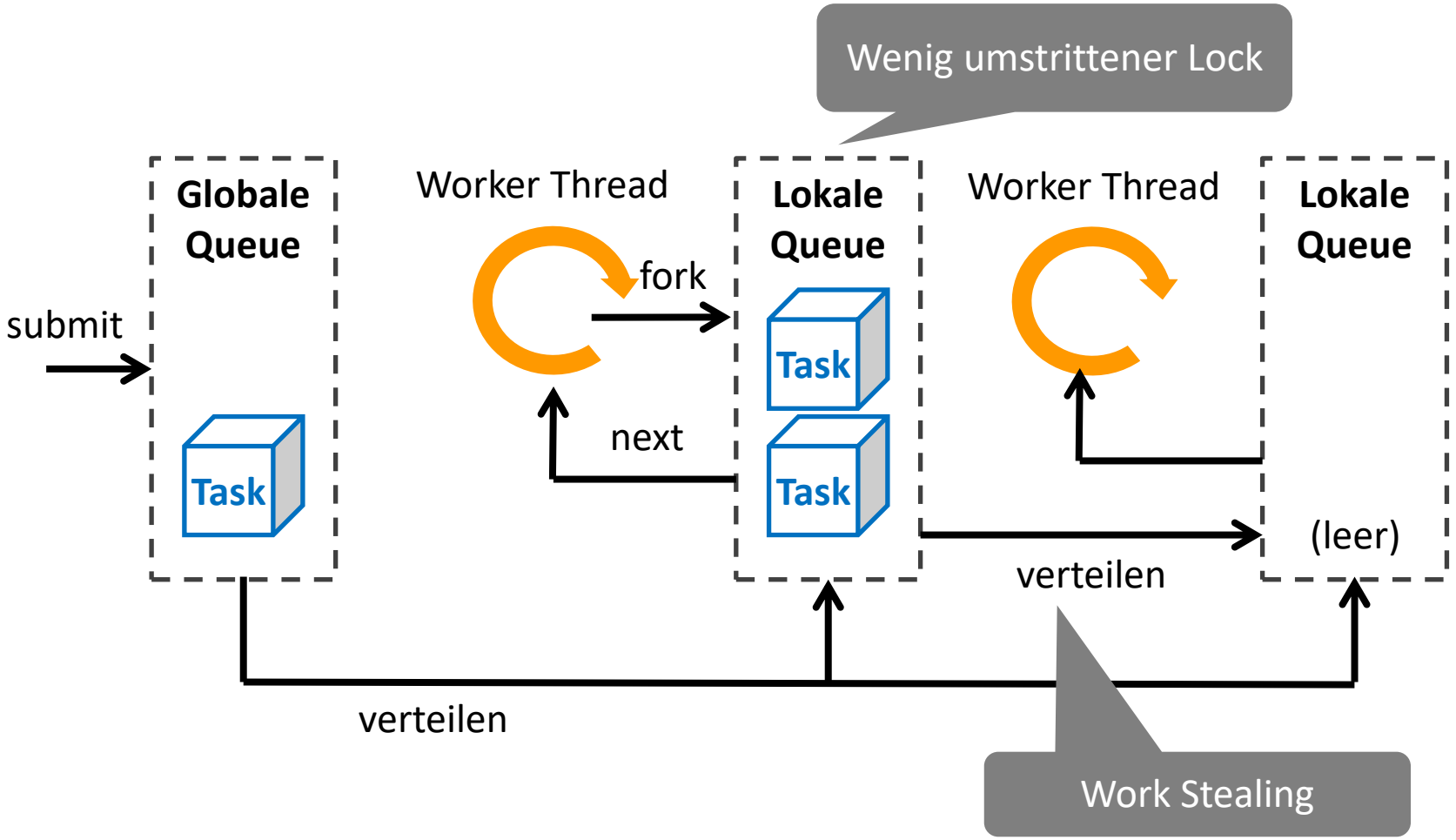
.NET Synchronisationsprimitiven

- Fehlen
  - Kein Fairness-Flag
  - Kein Lock & Condition
- Zusätzlich
  - ReadWriteLockSlim für Upgradeable Read/Write
  - Semaphoren auch auf OS-Stufe nutzbar
  - Mutex (binärer Semaphore auf OS-Stufe)
  - Spezielle Manual/AutoResetEvent
- Collections nicht Thread-safe
  - Ausser System.Collections.Concurrent

.NET Task Parallel Library (TPL)

- Work Stealing Thread Pool
  - Seit .NET 4: Effizienter Thread Pools
  - Auch bei Java Fork-Join Pool benutzt
- Verschiedene Abstraktionsstufen
  - Task Parallelization: Explizite Tasks starten und warten
  - Data Parallelization: Parallele Statements und Queries
  - Asynchrone Programmierung: mit Continuation Style

Work Stealing Thread Pool

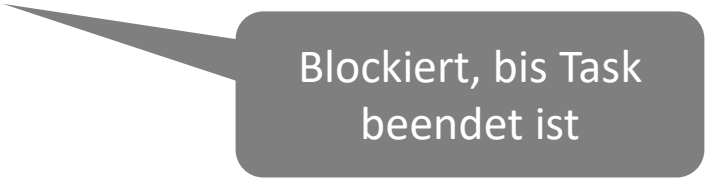


Thread Injection

- TPL fügt zur Laufzeit neue Worker Threads hinzu
- Hill Climbing Algorithmus
  - Misst Durchsatz & variiert Anzahl Worker Threads
  - Plus/minus ein Thread => Durchsatz besser?
- Kein Deadlock bei Task-Abhängigkeiten
  - Aber ineffizient, nicht dafür gemacht
  - Deadlock mit `ThreadPool.SetMaxThreads()` möglich

Task Parallelisierung: Start and Wait

```
Task task = Task.Run(() => {
    // task implementation
});
// perform other activity
task.Wait();
```



Task mit Rückgabe

Rückgabetyp

```
Task<int> task = Task.Run(() => {
    int total = … // some calculation
    return total;
});
…
Console.Write(task.Result);
```

Blockiert bis Task Ende und liefert dann Resultat

Typischer Fehler

```
for (int i = 0; i < 100; i++) {
  Task.Run(() => {
     // use i as input
     Console.WriteLine("Working with " + i);
  });
}
```

Was ist der Fehler?

Exception in Tasks

- Propagierung an Aufrufer von `Wait()` oder `Result`
  - `AggregateException`
- Bei Fire & Forget => Unobserved Exception vom GC
  - Seit .NET 4.5: Exception wird stillschweigend ignoriert
  - Abonnierung dieser Exception über Event `TaskScheduler.UnobservedTaskException`

Geschachtelte Tasks

- Tasks können Sub-Tasks starten und/oder abwarten
  - Kein spezieller ForkJoinTask nötig

```
Task.Run(() => {
  …
  var left = Task.Run(() => Count(leftPart));
  var right = Task.Run(() => Count(rightPart));
  int result = left.Result + right.Result;
  …
});
```

Datenparallelität in TPL

- Parallel Statements
  - Unabhängige Statements

```
void MergeSort(l, r) {
  long m = (l + r)/2;
  MergeSort(l, m);
  MergeSort(m, r);
  Merge(l, m, r);
}
```

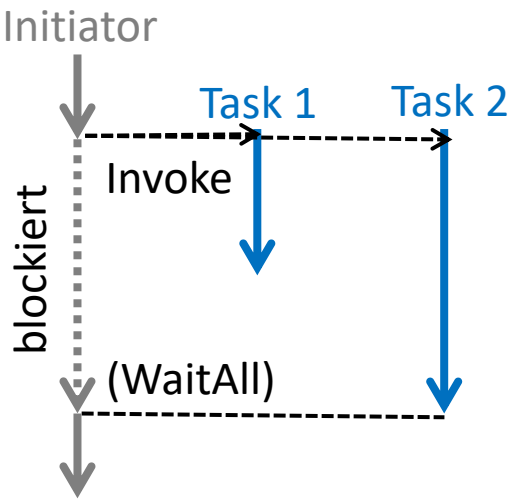
- Parallel Loop
  - Unabhängige Schlaufen-Bodies

```
void Convert(IList list) {
  foreach (File file in list) {
    Convert(file);
  }
}
```

Parallele Statements

- Menge an Statements potentiell parallel ausführen
- Als Tasks starten
- Barriere der Tasks am Ende

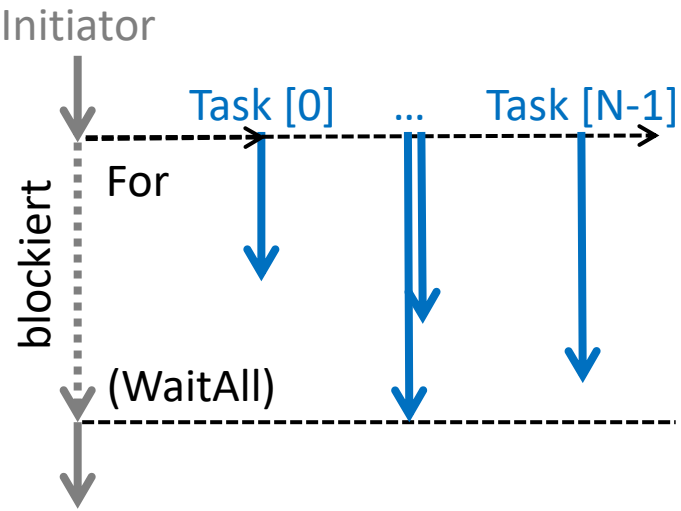
```
Parallel.Invoke(
   () => MergeSort(l, m),
   () => MergeSort(m, r)
);
```



Parallele Loop

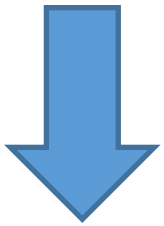
- Schlaufen-Bodies potentiell parallel ausführen
- Gruppierung der Bodies in Tasks
- Barriere dieser Tasks am Ende

```
Parallel.ForEach(list,
  file => Convert(file)
);
```



Parallel For

```
for (i = 0; i < array.Length; i++) {
  DoComputation(array[i]);
}
```

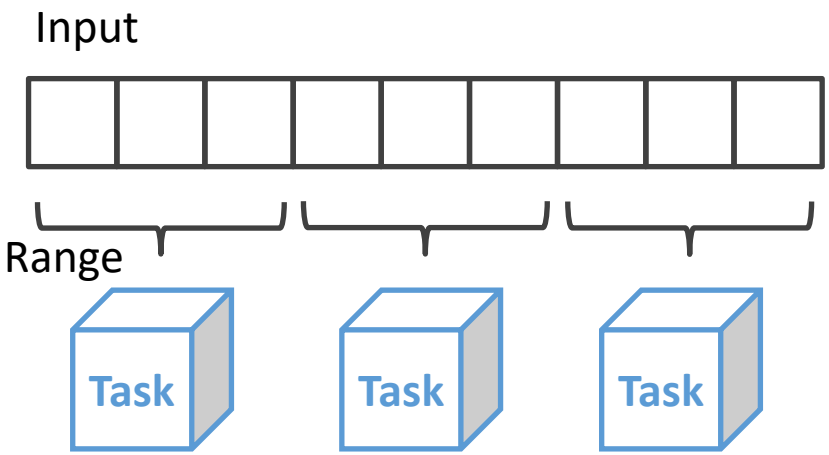


Falls Iterationen unabhängig sind

```
Parallel.For(0, array.Length,
  i => DoComputation(array[i])
);
```

Parallele Loop Partitionierung

- Schlaufe mit vielen sehr kurzen Bodies
  - Ineffizient: Jede Iteration als parallelen Tasks ausführen
- TPL gruppiert automatisch mehrere Bodies zu Task



Partitionierung bei Parallel Loops/Invoke

- Aufteilen gemäss verfügbarer Worker Threads

Range Partitioner

Bei Indexierung
(Parallel.For & Invoke)

Striped Partitioner

Bei Iterator
(Parallel.ForEach)

Explizite Partitionierung

Range Partitioner

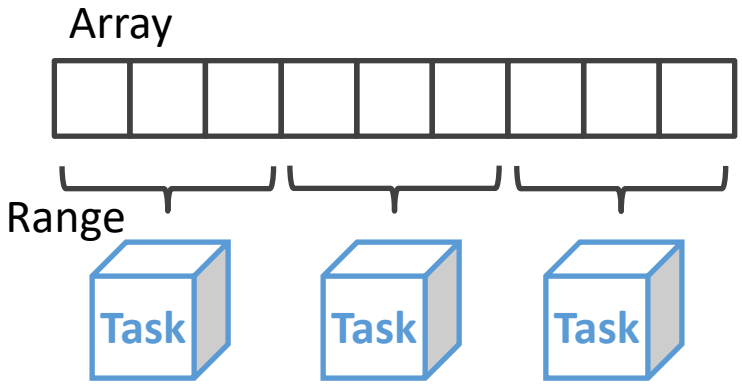
```
Parallel.ForEach(Partitioner.Create(0, array.Length),
        (range, _) => {
  for (int i = range.Item1; i < range.Item2; i++) {
    DoCalculation(array[i]);
  }
});
```

Inklusive untere Grenze

Exklusive obere Grenze

Vorteil: Weniger viele Body-Delegates

Nachteil: Künstliche Unterschlaufe



Fehlerhafte parallele Aggregation

```
long totalWords = 0;
Parallel.ForEach(list, file => {
  totalWords += CountWords(file);
);
```

Race Condition

Wie lässt es sich korrigieren?

Korrekte parallele Aggregation

```
long totalWords = 0;
Parallel.ForEach(list, file => {
  int subTotal = CountWords(file);
  lock (someLockObj) {
    totalWords += subTotal;
  }
);
```

Atomare Instruktion statt Lock wäre noch effizienter (später behandelt)

Parallel LINQ (PLINQ)

- Parallelisierung von Language-Integrated Query
  - LINQ: SQL-angelehnte Abfragesprache auf Objektmodell
  - Parallele Verarbeitung von Collection Operationen
- Pendant zu Java Stream API

LINQ Beispiele mit Methoden

ISBN von allen Bücher zum Titel Concurrency

```
bookCollection.
  Where(book => book.Title.Contains("Concurrency")).
  Select(book => book.ISBN)
```

Jede Zahl einer Liste auf bool abbilden (true ⇔ Primzahl)

```
inputList.
  Select(number => IsPrime(number))
```

Sequentiell

Eingebettete C# LINQ Syntax

ISBN von allen Bücher zum Titel Concurrency

```
from book in bookCollection
  where book.Title.Contains("Concurrency")
  select book.ISBN
```

Jede Zahl einer Liste auf bool abbilden (true ⇔ Primzahl)

```
from number in inputList
select IsPrime(number)
```

Sequentiell

Parallele LINQ Beispiele

ISBN von allen Bücher zum Titel Concurrency

```
from book in bookCollection.AsParallel()
  where book.Title.Contains("Concurrency")
  select book.ISBN
```

Beliebige Reihenfolge als Resultat

Jede Zahl einer Liste auf bool abbilden (true ⇔ Primzahl)

```
from number in inputList.AsParallel().AsOrdered()
select IsPrime(number)
```

Reihenfolge erhalten (Performance-Nachteil)

Vergleich: Java Stream API

ISBN von allen Bücher zum Titel Concurrency

Explizit unordered erlauben

```
bookCollection.parallelStream().unordered().
filter(book -> book.getTitle().contains("Concurrency")).
map(book -> book.getISBN());
```

Jede Zahl einer Liste auf bool abbilden (true ⇔ Primzahl)

```
inputList.parallelStream().
map(number -> isPrime(number));
```

Reihenfolge bleibt bei Liste ohne Weiteres erhalten

Asynchrone Programmierung mit TPL

- Task in TPL lancieren

Task repräsentiert asynchronen Aufruf

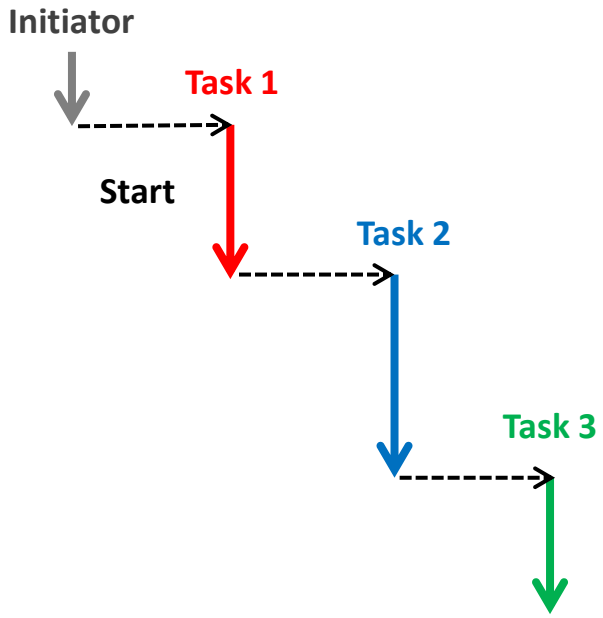
```
…
var task = Task.Run(
  LongOperation
);
// perform other work
int result = task.Result;
```

Zugriff ähnlich wie mit Future

Task Continuations

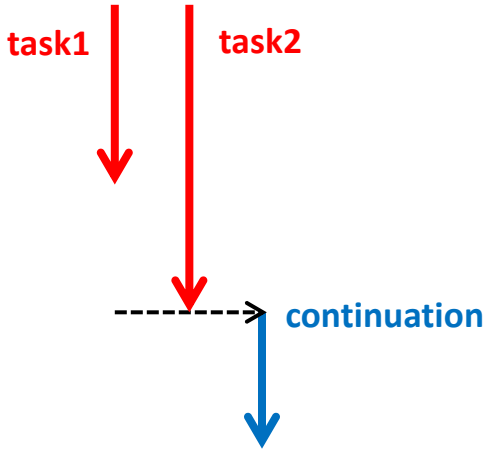
- Definiere Task, der automatisch nach dem Ende eines anderen startet
- Wie Java CompletableFutures

```
task1.
ContinueWith(task2).
ContinueWith(task3);
```

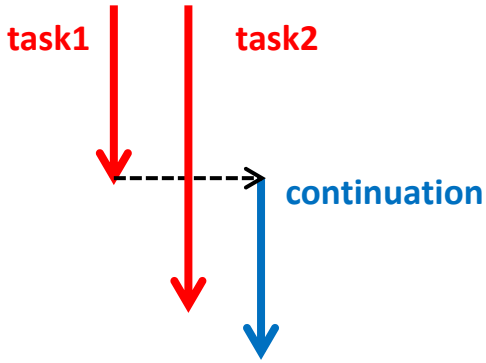


Multi-Continuation

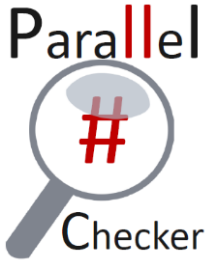
```
Task.WhenAll(task1, task2).
ContinueWith(continuation);
```




```
Task.WhenAny(task1, task2).
ContinueWith(continuation);
```

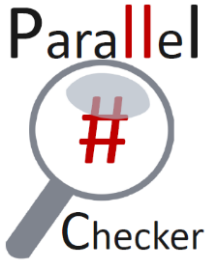


Parallel Checker



- Erkennung von Nebenläufigkeitsfehler in C#
  - Data Races
  - Deadlocks
- Statische Analyse in Visual Studio während Coding
  - Schnelle und präzise Warnungen
  - Dafür eventuell nicht alle Fehler
- https://github.com/blaeser/parallelchecker

Installation Parallel Checker



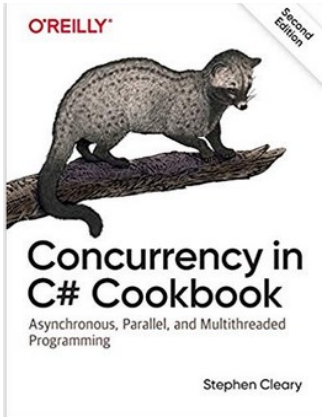
- Visual Studio
  - Menü „Tools -> Extensions and Updates“
  - „Parallel Checker“ suchen und installieren
- Visual Studio Code:
  - Nuget Package “ConcurrencyChecker.ParallelChecker”
  - Package zu den C#-Projekten installieren

Rückblick: Lernziele

- Unterschiede des parallelen Programmierens zwischen Java und .NET kennen
- Konzepte und Abstraktionsstufen der Task Parallel Library verstehen
- Task Parallel Library einsetzen können

Weitergehende Lektüre (1)

S. Clearly. Concurrency in C# Cookbook, 2nd edition, O'Reilly, 2019

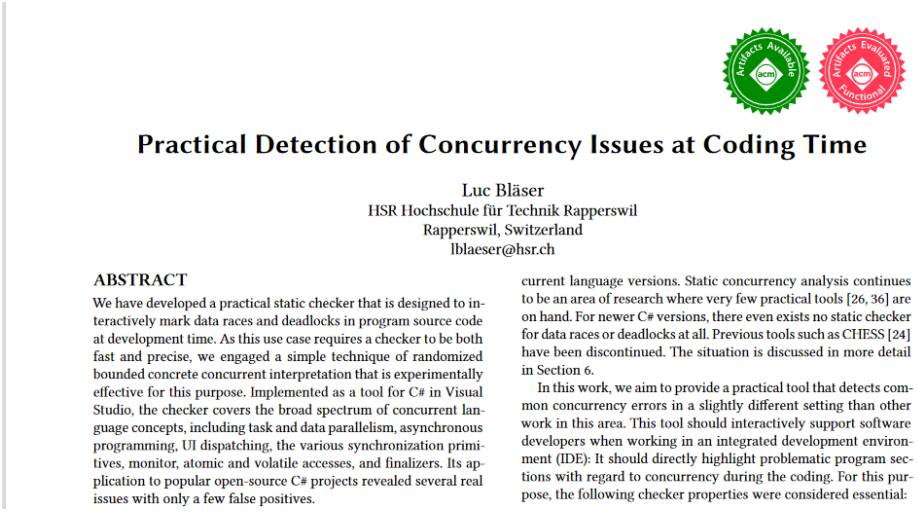


M. A. Zhou, Parallel Computing in .NET, Entwickler-Press, 2009

J. Albahari, Threading in C#, http://www.albahari.com/threading

Weitergehende Lektüre (2)

- Paper zur Task Parallel Library
  - D. Leijen, W. Schulte, and S. Burckhardt. The Design of a Task Parallel Library. OOPSLA '09.
- Paper zum Parallel Checker
  - L. Bläser. Practical Detection of Concurrency Issues at Coding Time. ISSTA 2018, July 2018.



Practical Detection of Concurrency Issues at Coding Time

Luc Bläser
HSR Hochschule für Technik Rapperswil
Rapperswil, Switzerland
lblaeser@hsr.ch

ABSTRACT

We have developed a practical static checker that is designed to interactively mark data races and deadlocks in program source code at development time. As this use case requires a checker to be both fast and precise, we engaged a simple technique of randomized bounded concrete concurrent interpretation that is experimentally effective for this purpose. Implemented as a tool for C# in Visual Studio, the checker covers the broad spectrum of concurrent language concepts, including task and data parallelism, asynchronous programming, UI dispatching, the various synchronization primitives, monitor, atomic and volatile accesses, and finalizers. Its application to popular open-source C# projects revealed several real issues with only a few false positives.

current language versions. Static concurrency analysis continues to be an area of research where very few practical tools [26, 36] are on hand. For newer C# versions, there even exists no static checker for data races or deadlocks at all. Previous tools such as CHESS [24] have been discontinued. The situation is discussed in more detail in Section 6.

In this work, we aim to provide a practical tool that detects common concurrency errors in a slightly different setting than other work in this area. This tool should interactively support software developers when working in an integrated development environment (IDE): It should directly highlight problematic program sections with regard to concurrency during the coding. For this purpose, the following checker properties were considered essential: