.NET Task Parallel Library: Eine Rundtour

Luc Bläser

Hochschule für Technik Rapperswil

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

INSTITUT
FÜR
SOFTWARE

Parallel 2015 Konferenz
23. Apr. 2015, Karlsruhe

Die .NET Task Parallel Library (TPL)

- Stand der Technik für .NET Parallelprogrammierung
  - Ersetzt weitgehend explizites Multi-Threading
  - Eingeführt mit .NET 4, erweitert mit 4.5
- Bekannt für hohe Performance und Allgemeinheit
  - **Verschiedene Programmierabstraktionen**
  - Auf einem gemeinsamen Rückgrat

free icon from wikimedia.org

Die verschiedenen TPL-Abstraktionen

Ziel: Multi-Core

Datenparallelität

- Statement Unabhängigkeiten
- LINQ Unabhängigkeiten

Ziel: Nicht blockierend

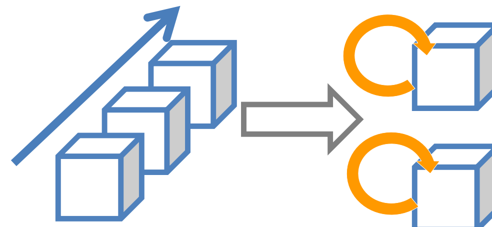
Asynchronität

- Asynchrone Methoden
- Reactive Data Flows

Task-Parallelisierung

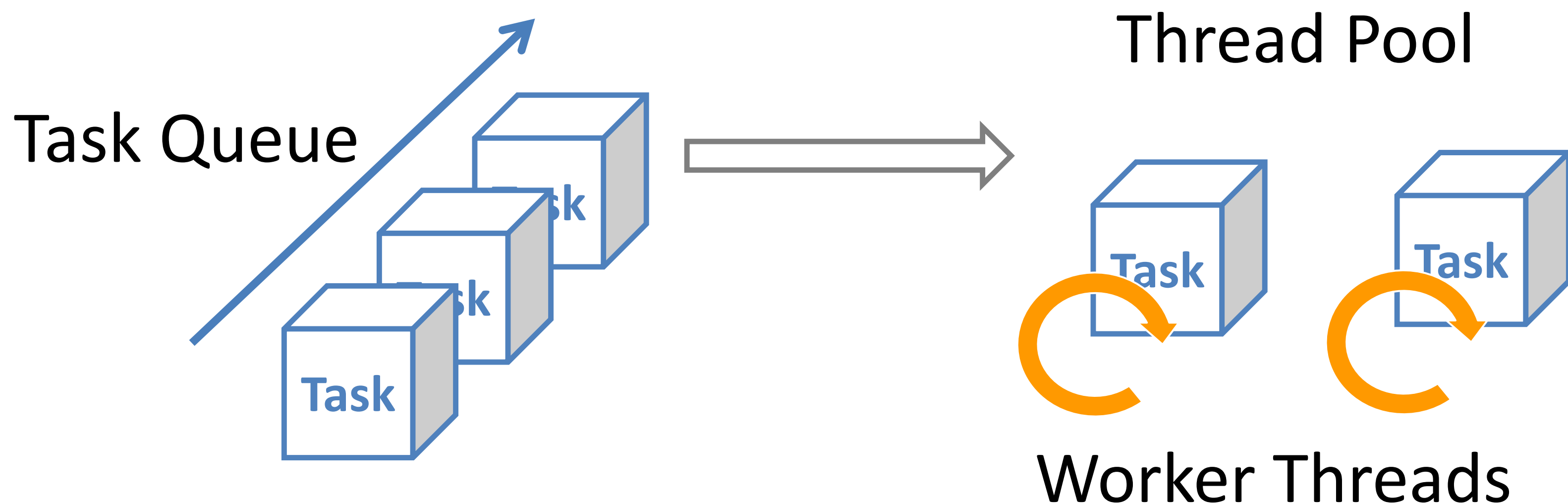
- Expliziter Einsatz eines Thread Pools

Thread Pool



Thread Pool: Prinzip

- Task implementiert potentiell parallele Arbeit
- Pool mit beschränkter Anzahl Worker Threads
- Tasks einreihen, Threads holen und führen sie aus



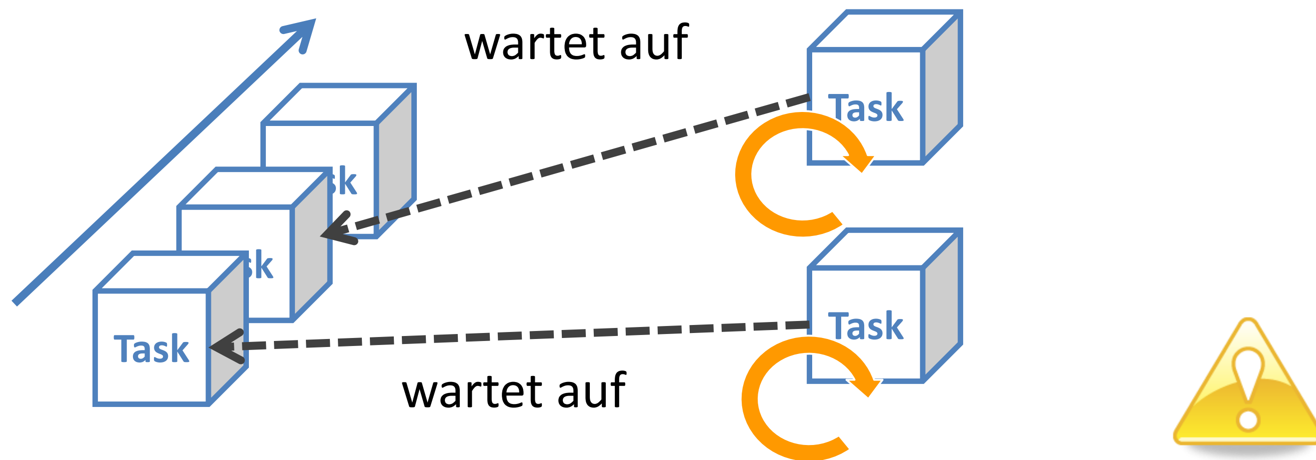
#Worker Threads = #Prozessoren + #I/O-Aufrufe

Der Nutzen

- Hoher Grad an Parallelität modellierbar
  - Viele Tasks = billige passive Objekte
- Tiefe Thread-Kosten
  - Nur wenige rezyklierte Threads
- „Free lunch“ bei Task-basierten Programmen
  - Automatisch schneller mit mehr Cores

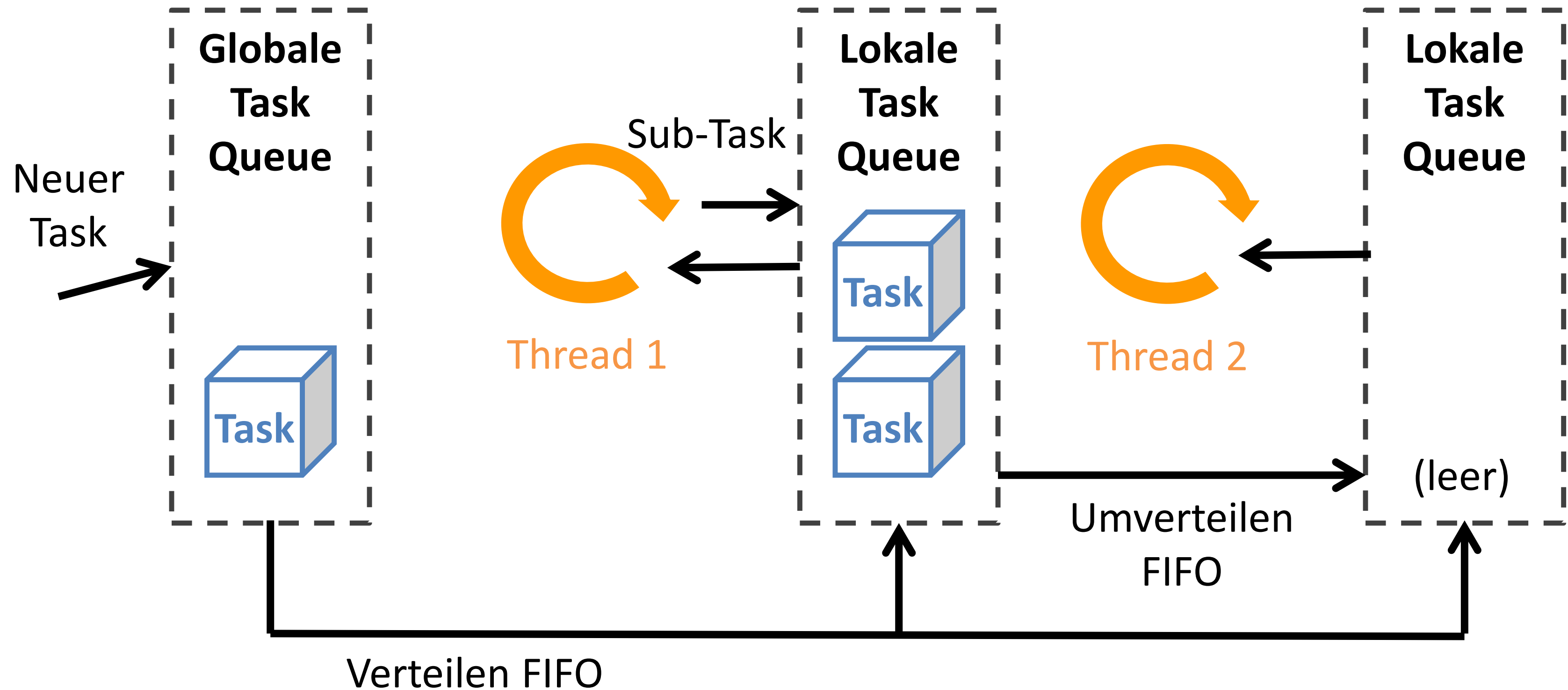
Und die Schattenseite?

- Keine Warteabhängigkeiten zwischen Tasks
  - Sonst Deadlock (bei fixer Anzahl Worker Threads)
  - Thread Injection (TPL fügt langsam neue Threads hinzu)
  - Ausnahme: Geschachtelte Tasks



Der Work Stealing Mechanismus

- Verringerte Contention mittels lokalen Queues
- Anzahl Threads gemäß Task-Durchsatz angepasst



© Bild von Jürg Gutknecht, ETH Zürich

Vortragsstruktur

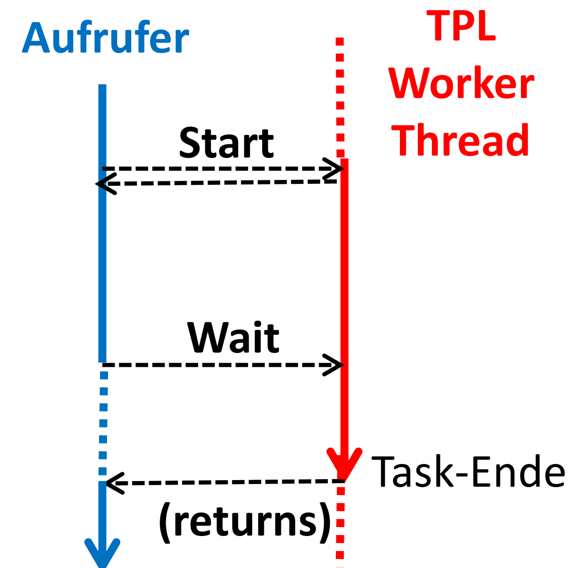
1) Task-Parallelisierung
2) Datenparallelität
3) Asynchronität

1) Task-Parallelisierung

- Explizite Verwendung von Thread Pool Tasks
  - Basis für höhere Abstraktionsstufen
  - Ideal für komplexere Muster

```
var task = Task.Factory.StartNew(() => {
  // task implementation
  return …;
});
…
…
var result = task.Result;
```

Implizites **`task.Wait()`**



Vorsicht vor Nebenläufigkeitsfehlern

- Gleich gefährlich wie bei explizitem Multi-Threading
  - Tasks laufen evtl. nebenläufig (in verschiedenen Threads)

```
for (int i = 0; i < 100; i++) {
  Task.Factory.StartNew(() => {
    Console.WriteLine(i);
  });
}
```

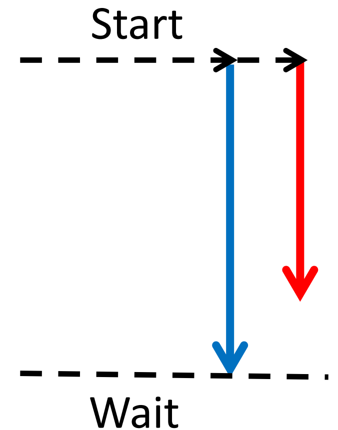
Data Race = formaler Fehler

- Vermeiden Sie
  - Race Conditions (Low-Level und High-Level)
  - Deadlocks (inkl. Livelocks)
  - Starvation (Fairness-Probleme)

Verschiedene Muster (1)

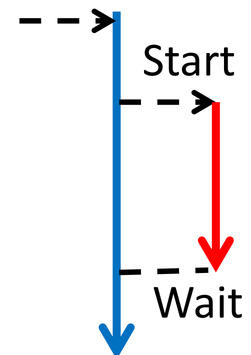
- Start & Join

```
var task1 = Task.Factory.StartNew(CountLeft);
var task2 = Task.Factory.StartNew(CountRight);
…
Task.WaitAll(task1, task2);
// äquivalent zu task1.Wait(); task2.Wait();
```



- Geschachtelte Tasks

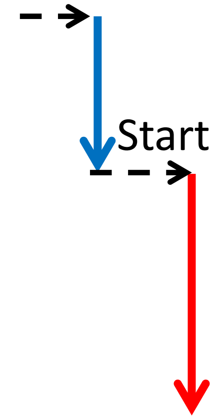
```
var outerTask = Task.Factory.StartNew(() => {
  var innerTask = Task.Factory.StartNew(() => { … });
  …
  innerTask.Wait();
}
```



Verschiedene Muster (2)

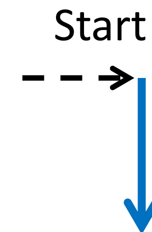
- Verkettung

```
var firstTask = Task.Factory.StartNew(…);
var secondTask = new Task(…);
firstTask.ContinueWith(secondTask);
```



- Fire & Forget

```
Task.Factory.StartNew(() => {
   …
}
```



Fehlerbehandlung

- Unbehandelte Exception in Task => Task ist „faulted“
  - Wird an Wait() oder Result propagiert
  - Sonst ignoriert (Default seit .NET 4.5)
    - `TaskScheduler.UnobservedTaskException`

Vorsicht auch bei Fire & Forget

- Exceptions in Tasks werden ignoriert (Default)

- Anwendung kann vor Task Ende terminieren
  - Thread Pool verwendet Background Threads

2) Datenparallelität

- Deklarativ: Nutze Unabhängigkeiten im Programm
  - Kann parallelisiert werden, muss aber nicht unbedingt
  - Ziel: Beschleunigung durch Multi-Cores

2a) Statement-Level

2b) LINQ Ausdrücke

Statement-Level Parallelism

- Parallele Statements
  - Unabhängige Statements

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

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

- Parallel Schlaufen
  - Unabhängige Loop Steps

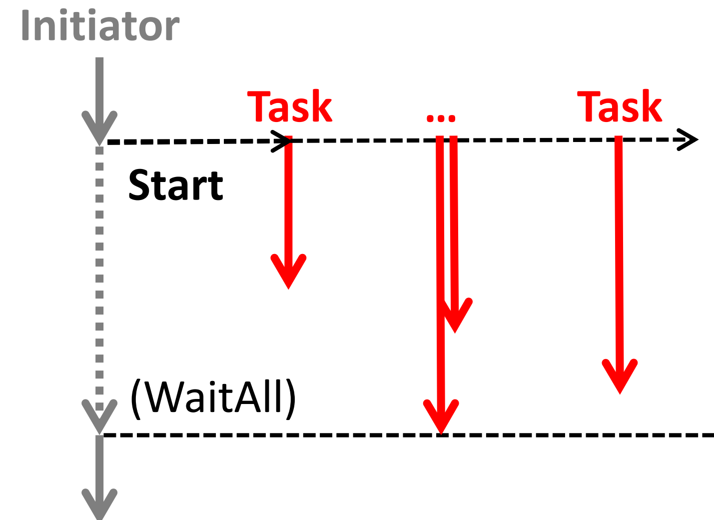
```
void Convert(IList<File> files) {
  foreach (File f in files) {
    Convert(f);
  }
}
```

```
Parallel.ForEach(files,
  f => Convert(f)
);
```

Parallele Statement Ausführung

- Task pro Gruppe von Statements und Loop-Schritten
- Automatische Partitionierung in Tasks
- Warte-Barriere am Ende des parallelen Blocks

```
Parallel.For(0, N,
  i => DoComputation(i)
);
```



Parallel LINQ

- Erlaube parallele Abfrageverarbeitung

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

Beliebige Reihenfolge beim Resultat

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

Erhalte Input-Reihenfolge

Abfrage soll frei von Seiteneffekten sein (keine Race Conditions)

3) Asynchronität

- Ziel: Nicht-blockierende Logiken/User Interfaces
- In C# eingebaute Syntax

Potentiell asynchrone Methode

```
public async Task<int> LongOperationAsync() { … }

…
var task = LongOperationAsync();
OtherWork();
int result = await task;
…
```

Warte auf Ende der async Methode

Async ≠ Asynchrone Methode

- async Methode
  - blockiert nicht unbedingt während ganzer Ausführung
  - **Teils synchron, teils asynchron**
- async Rückgabetypen
  - Task<T>: Rückgabetyp T
  - Task: keine Rückgabe, Aufrufer kann aber darauf warten
  - void: nur Fire & Forget
- await Ausdruck
  - Macht erst weiter, wenn Methode beendet ist
  - Liefert Rückgabewert (falls definiert)

Beispiel: Asynchrone Downloads

Rückgabetyp string

Suffix „async“ als Namenskonvention

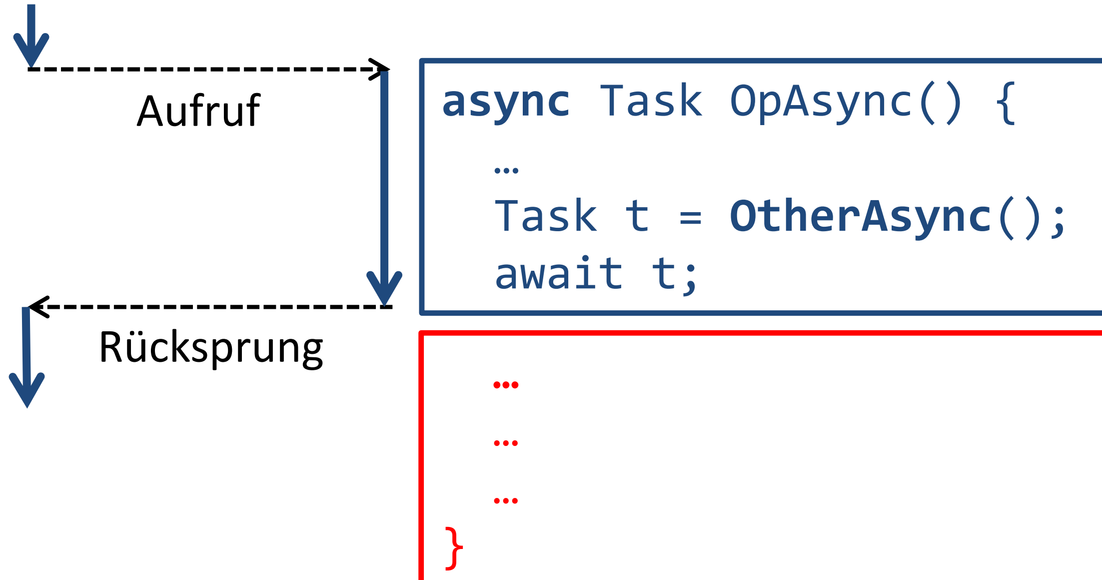
```
async Task<string> ConcatWebSitesAsync(string url1, string url2)
{
  var client = new HttpClient();
  var download1 = client.GetStringAsync(url1);
  var download2 = client.GetStringAsync(url2);
  string site1 = await download1;
  string site2 = await download2;
  return site1 + site2;
}
```

Direkte String-Rückgabe möglich

```
async Task<string>
GetStringAsync(string url)
```

Async Methodenaufruf

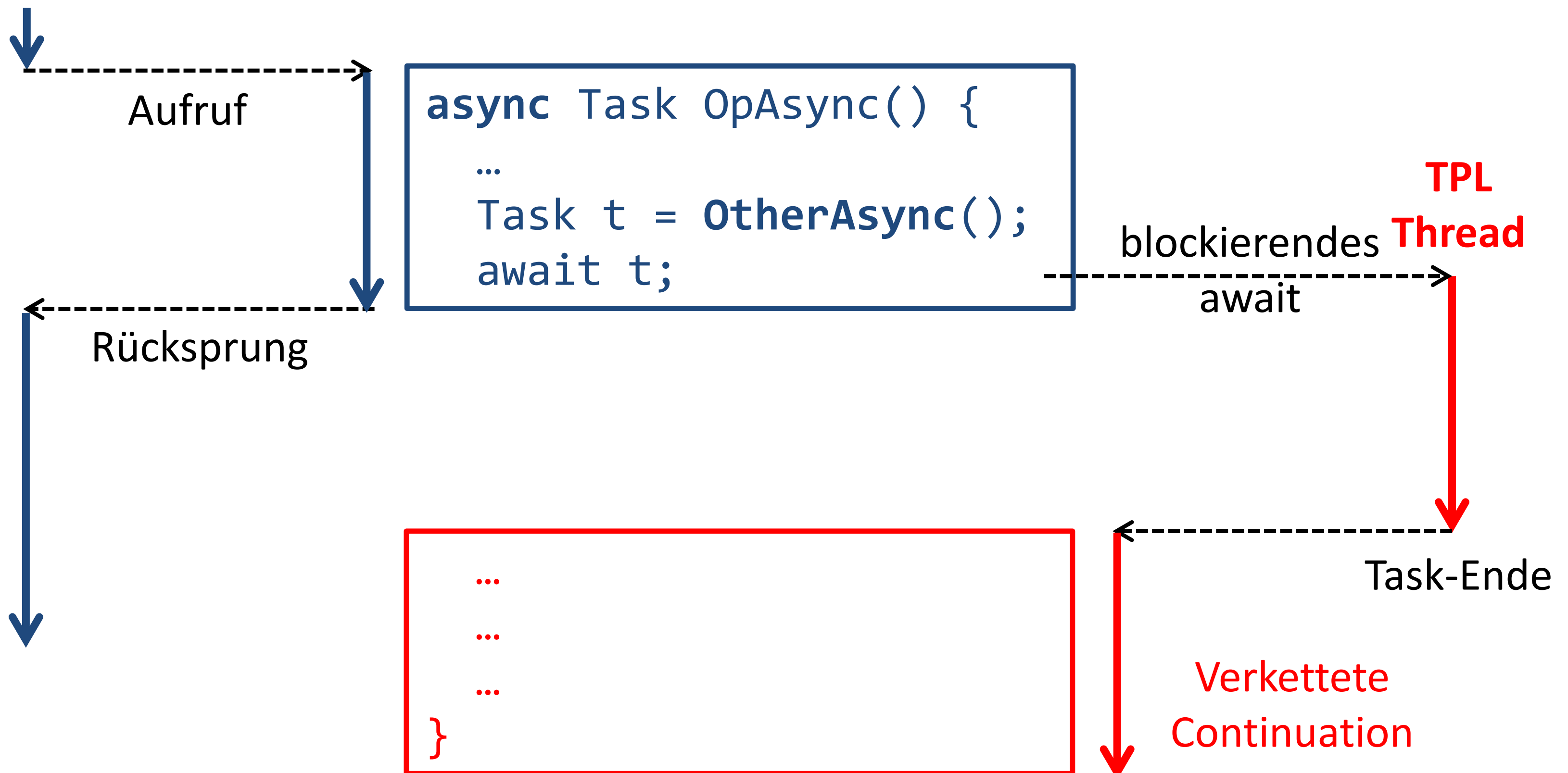
- Methode läuft synchron, bis ein await blockiert
  - Wartet auf anderen Thread oder IO
- Rücksprung zum Aufrufer beim blockierenden await



Läuft nach Ende von Task t

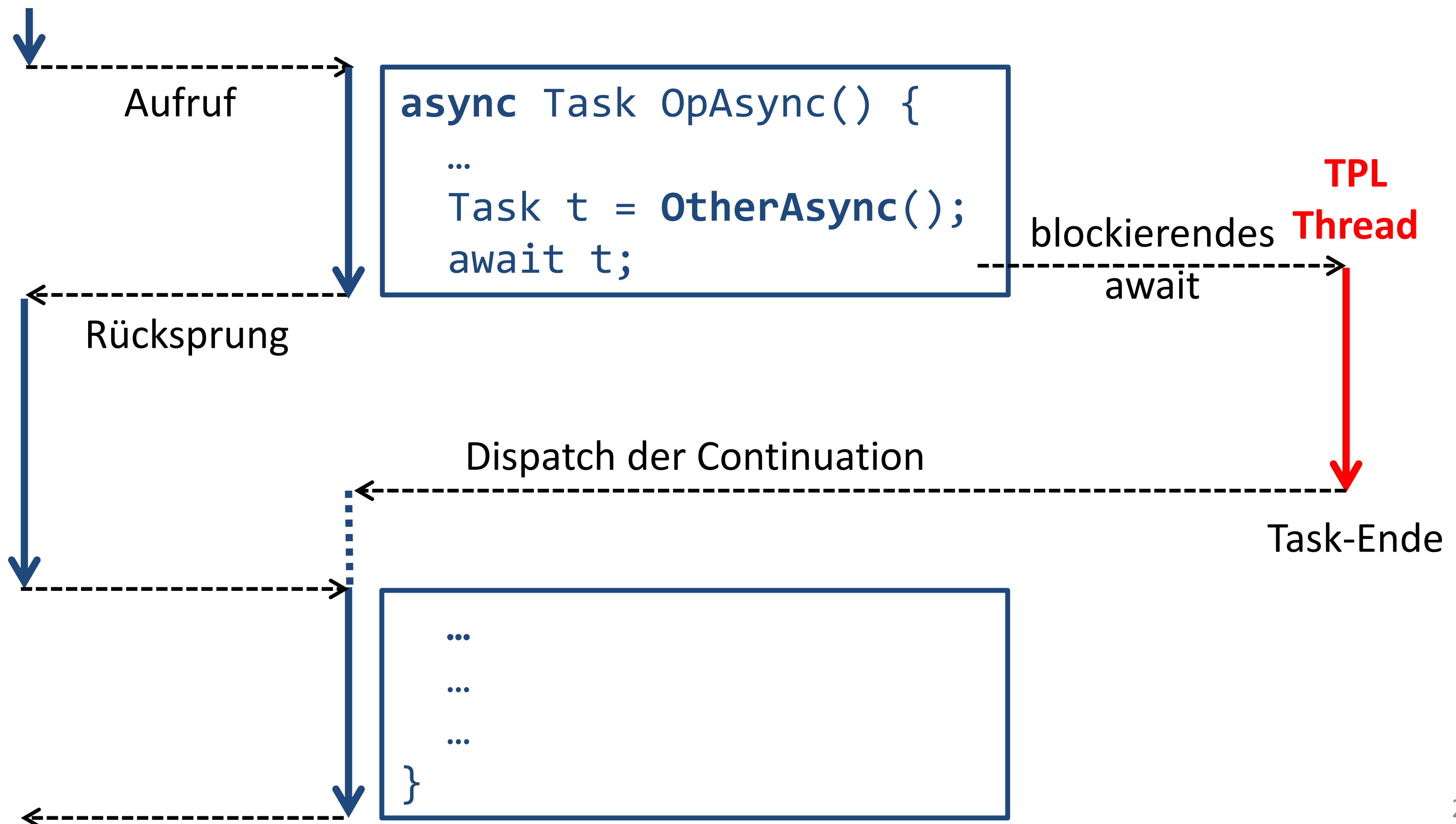
Fall 1: Aufrufer ist kein UI-Thread

- Anderer Thread führt Ausführung nach await durch



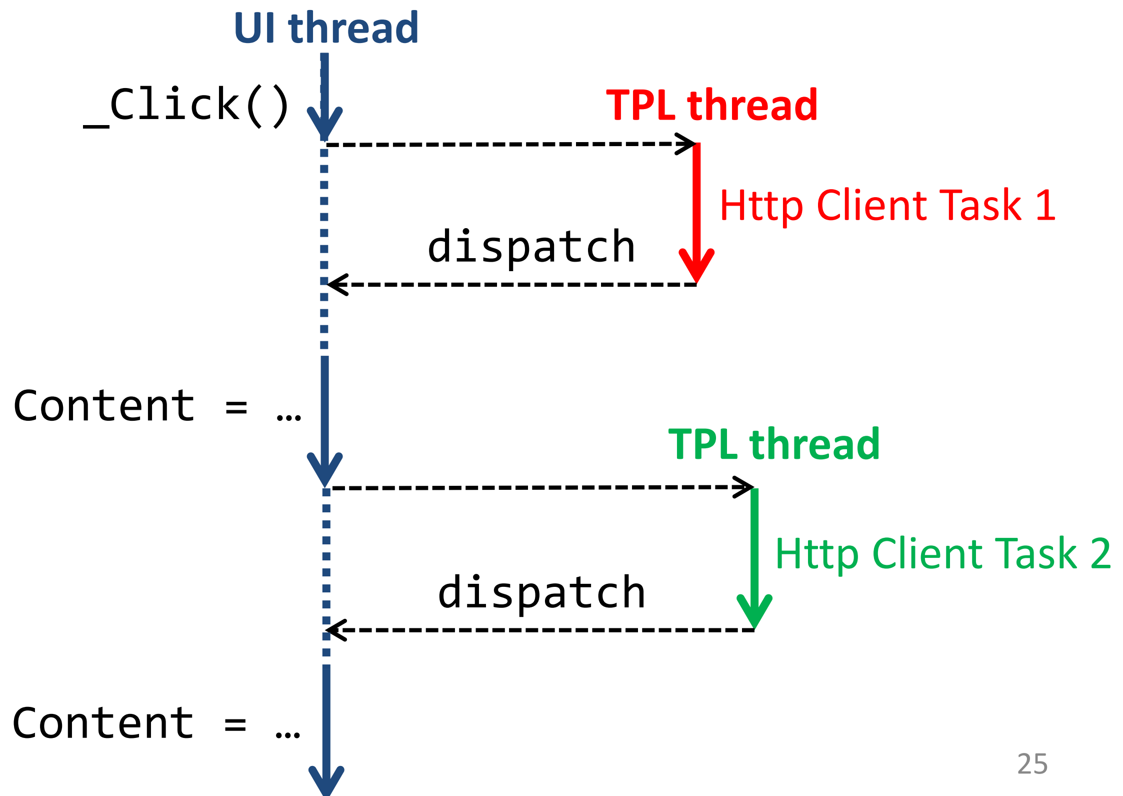
Fall 2: Aufrufer ist UI-Thread

- Rest wird später zum UI-Thread versandt (Dispatch)



Leserlicher nicht-blockierender UI-Code

```
async void startDownload_Click(…) {
   HttpClient client = new HttpClient();
   foreach (var url in collection) {
      var data = await client.GetStringAsync(url);
      textArea.Content += data;
   }
}
```



Achtung: Notorische Fehler

1. Async Methoden sind nicht per se asynchron
   - `Task.Run()` für lang-laufende synchrone Logik
2. Thread-Wechsel innerhalb Methodenaufruf
   - Thread-lokaler Zustand ist dann nicht mehr gültig
3. Quasi-Parallelität der UI Event-Behandlung
   - await is gleich problematisch wie das alte `DoEvents()`
4. Race Conditions bleiben möglich
   - Z.B. bei Fall 1 => Beide Fälle testen
5. UI Deadlock Risiko
   - Kein `task.Wait()`, `task.Result` im UI-Thread Code

Schlussfolgerungen

- TPL ist ein mächtiges effizientes Werkzeug
  - Verschiedene Abstraktionen auf Basis des Thread Pools

Abstraktion	Bestandteil	Anwendungsfall
Task-Parallelisierung	Explizites Starten, Warten, Verketten von Tasks	Komplexe Task-Strukturen
Datenparallelität	Parallel Invoke / Loops PLINQ	Deklarative Multi-Core Beschleunigung
Asynchronität	async/await	Nicht-blockierende Logiken/UIs

- Beachten Sie die Fallstricke
  - Nebenläufigkeitsfehler, Fire & Forget, async/await, …

Danke für Ihre Aufmerksamkeit

- Concurrency Research, Consulting, und Training
  - http://concurrency.ch
- Kontakt
  - **Prof. Dr. Luc Bläser**
    HSR Hochschule für Technik Rapperswil
    IFS Institut für Software
    Rapperswil, Schweiz
  - lblaeser@hsr.ch