The Various Faces of the .NET Task Parallel Library

Luc Bläser

Hochschule für Technik Rapperswil

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

INSTITUT
FÜR
SOFTWARE

Multicore@Siemens 2015
5 Feb 2015, Nuremberg

The .NET Task Parallel Library (TPL)

- State of the art in .NET parallel programming
  - Replaces Most Explicit Multi-Threading
  - Introduced in .NET 4, extended 4.5
- Known for high performance & generality
  - **Different programming abstractions**
  - On a common backbone

free icon from wikimedia.org

TPL Appears in Various Faces

Goal: Multi-Core

Data Parallelism

- Statement independencies
- LINQ independencies

Goal: Non-Blocking

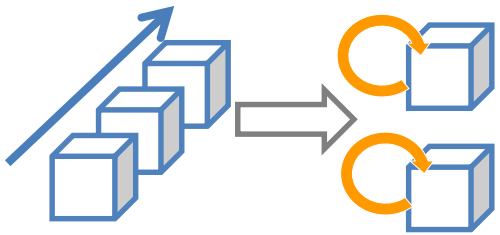
Asynchronous Programming

- Asynchronous methods
- Reactive data flows

Task Parallelization

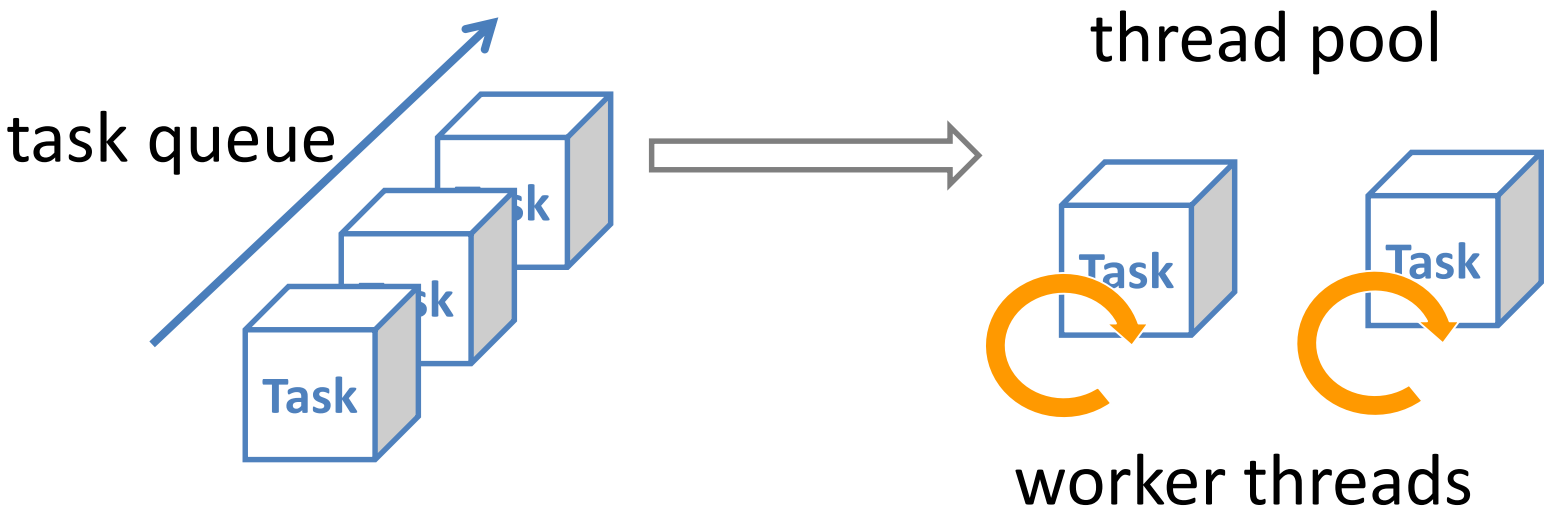
- Explicit thread pool usage

Thread Pool



Thread Pool Principle

- Task implement potentially parallel work
- Thread pool = limited number of worker threads
- Tasks are queued, threads fetch and execute tasks



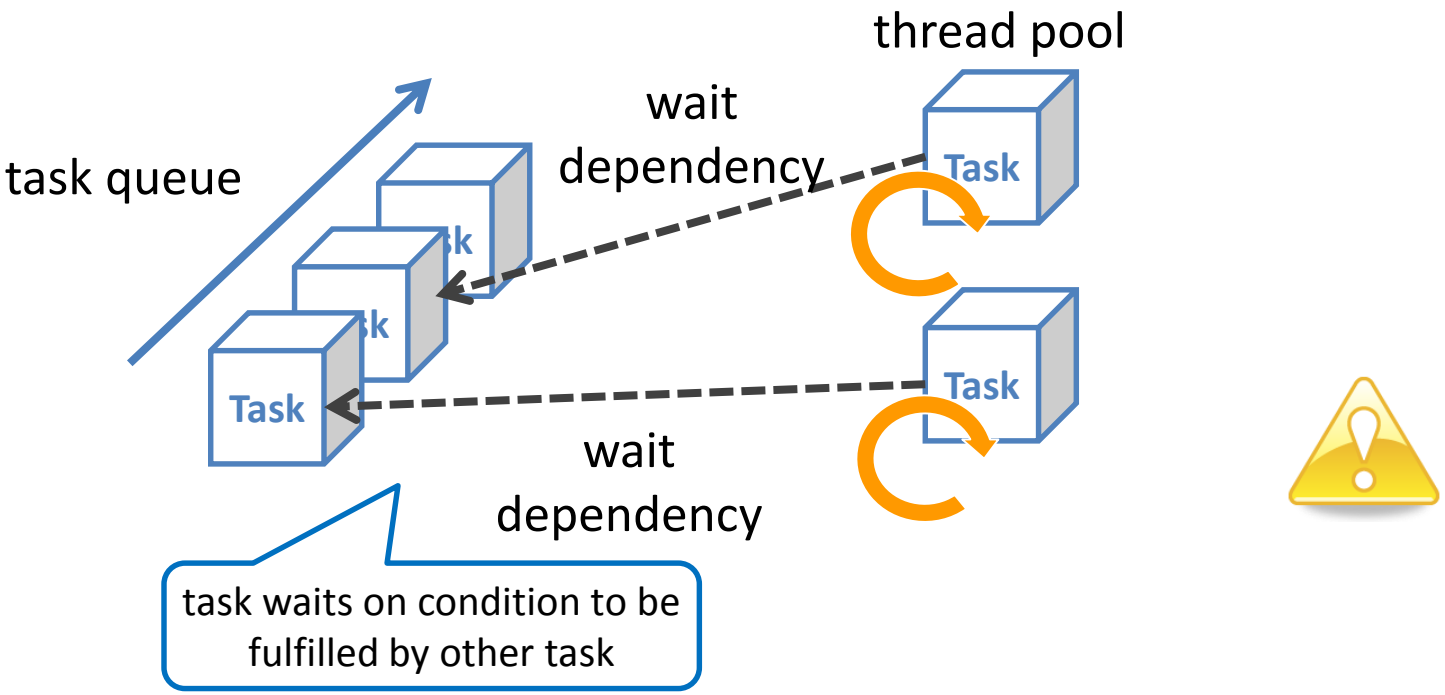
#worker threads = #processors + #I/O calls

The Thread Pool Advantage

- Modell large degree of parallelization
  - Many tasks = cheap passive objects
- Pay low threading costs
  - Only few recycled threads
- Free lunch with task-modelled programs
  - Automatically faster with more cores

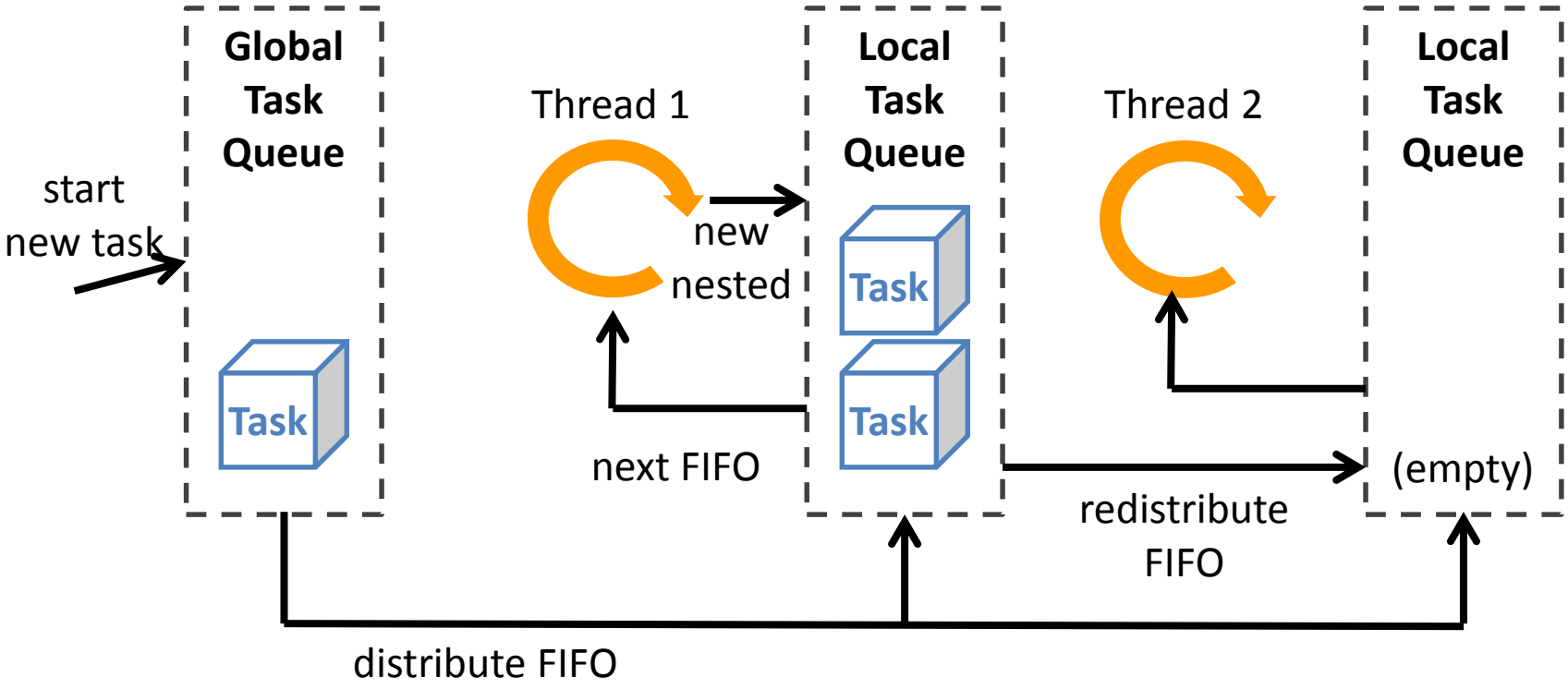
And the Downside?

- Tasks must not have mutual wait dependencies
  - Otherwise deadlock (if thread pool amount is fixed)
  - Thread injection (TPL increases number of threads slowly)
  - Exception: Nested tasks



TPL's Work Stealing Thread Pool

- Reducing contention with local queues
- Number of threads adjusted to task throughput



Structure of Talk

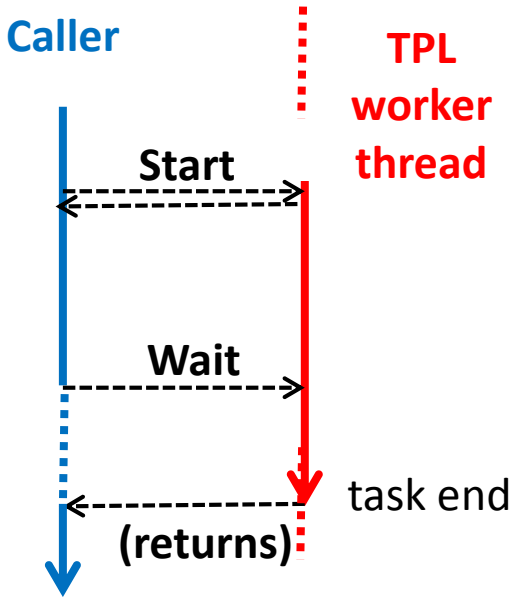
1) Task Parallelization
2) Data Parallelism
3) Asynchronous Programming

1) Task Parallelization

- Explicit handling of thread pool tasks
  - Elementary support for higher abstraction level
  - Ideal for more complex pattern than higher levels

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

Implicit **task.Wait()**



Watch Out for Concurrency Errors

- As immanent as in multi-threading
  - Tasks may run concurrently (by different threads)

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

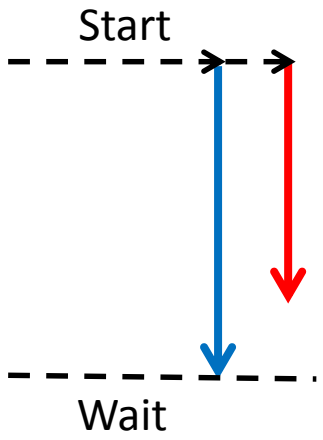
Data race = formal error

- Avoid
  - Race conditions (low-level and high-level)
  - Deadlocks (incl. livelocks)
  - Starvation (fairness issues)

Different Styles (1)

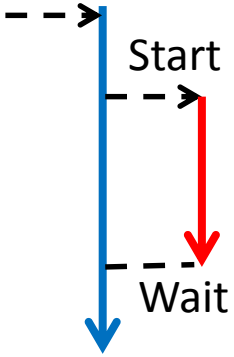
- Start & Join

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



- Nested tasks

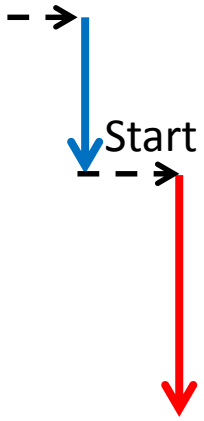
```
var outerTask = Task.Factory.StartNew(() => {
  var innerTask = Task.Factory.StartNew(() => { … });
  …
  innerTask.Wait();
}
```



Different Styles (2)

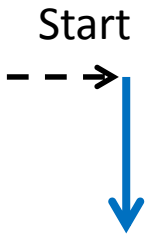
- Chaining

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



- Fire & Forget

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



Exception Handling

- Unhandled exception in task => task is faulted
  - Propagated to caller of Wait() or Result
  - Otherwise ignored (default since .NET 4.5)
    - `TaskScheduler.UnobservedTaskException`
  - Not automatically propagated along task chain
    - Successor task should call Wait()/Result of predecessor

Caution with Fire & Forget

- Exceptions in task are ignored (by default)

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

ignored

- Application may stop before task is completed
  - Thread pool uses background threads

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

sudden termination

```
  …
}
```

2) Data Parallelism

- Declarative: Exploit independencies in program code
  - Can be parallelized, but do not necessarily have to
  - Goal: Acceleration by multi-cores

2a) Statement-Level

2b) LINQ Expressions

Statement-Level Parallelism

- Parallel Statement Blocks
  - Independent 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 Loop Blocks
  - Independent loop steps

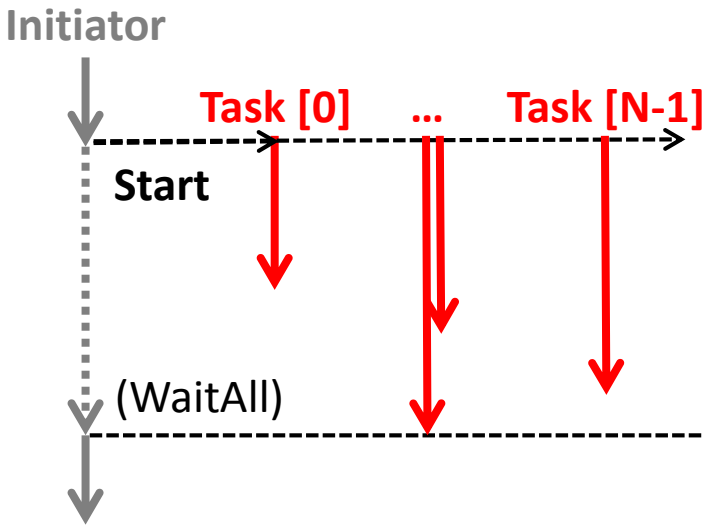
```
void Convert(IList<File> files) {
  foreach (File f in files) {
    Convert(f);
  }
}
```

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

Parallel Statement Execution

- Task started per invoked statement or loop step
- Wait-barrier at the end of parallel block

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



Parallel LINQ

- Permit parallel query processing

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

arbitrary result order

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

Retain input order

Query should be side-effect-free (avoid race conditions)

3) Asynchronous Programming

- Goal: Non-blocking Logics/User Interfaces
- Institutionalized language keywords async/await

Potentially asynchronous method

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

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

Wait for termination of async method

Async ≠ Asynchronous Method

- async method
  - Caller is not necessarily blocked during entire execution
  - **Partially synchronous, partially asynchronous**
- async method return types
  - Task<T>: return value T
  - Task: no return value, but caller can await it
  - void: only fire & forget
- await expression
  - continues only when method is completed
  - evaluates to return value (if defined)

Example: Asynchronous Downloads

Suffix „Async“ as naming convention

Return type string

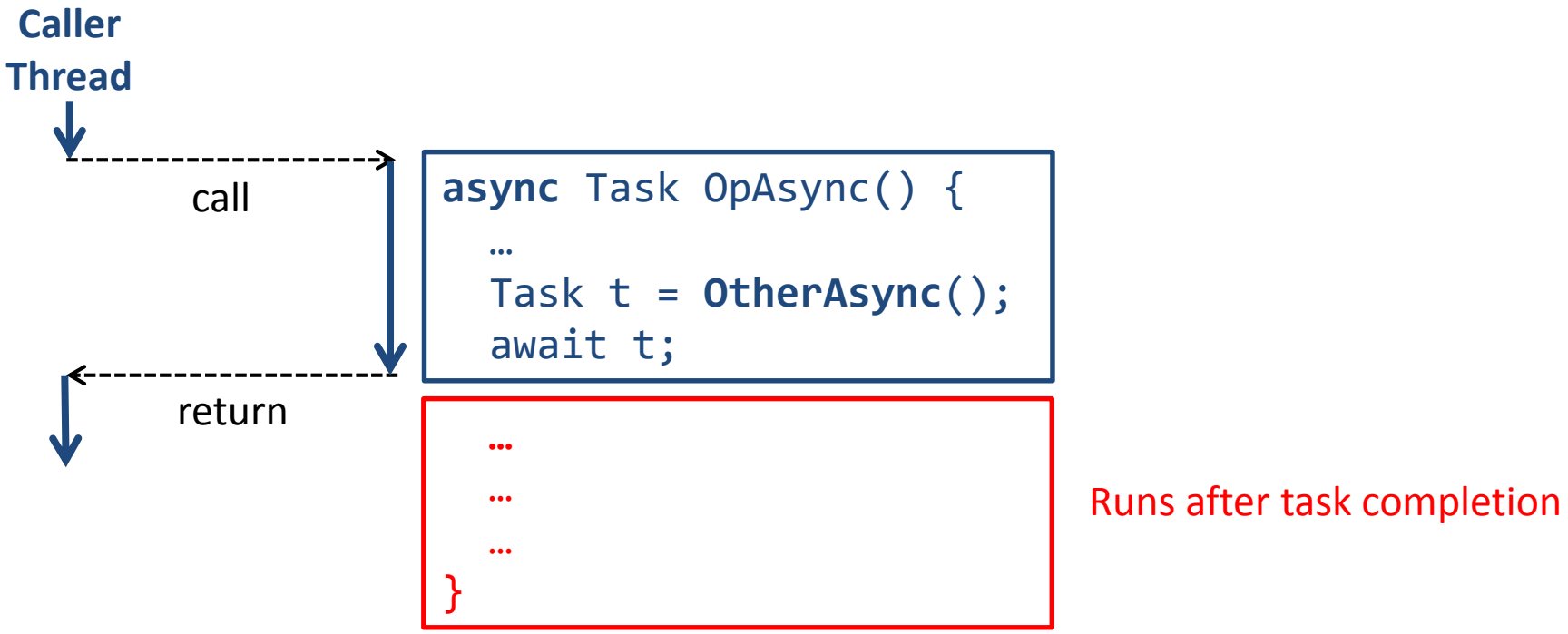
```
async Task<string> ConcatWebSitesAsync(string url1, string url2)
{
  var client = new HttpClient();
  var download1 = client.GetStringAsync(url1);
  var download2 = client.GetStringAsync(url2);
  string site1 = await download1;
  string site2 = await download2;
  return site1 + site2;
}
```

async Task<string> GetStringAsync(string url)

Immediate string return possible

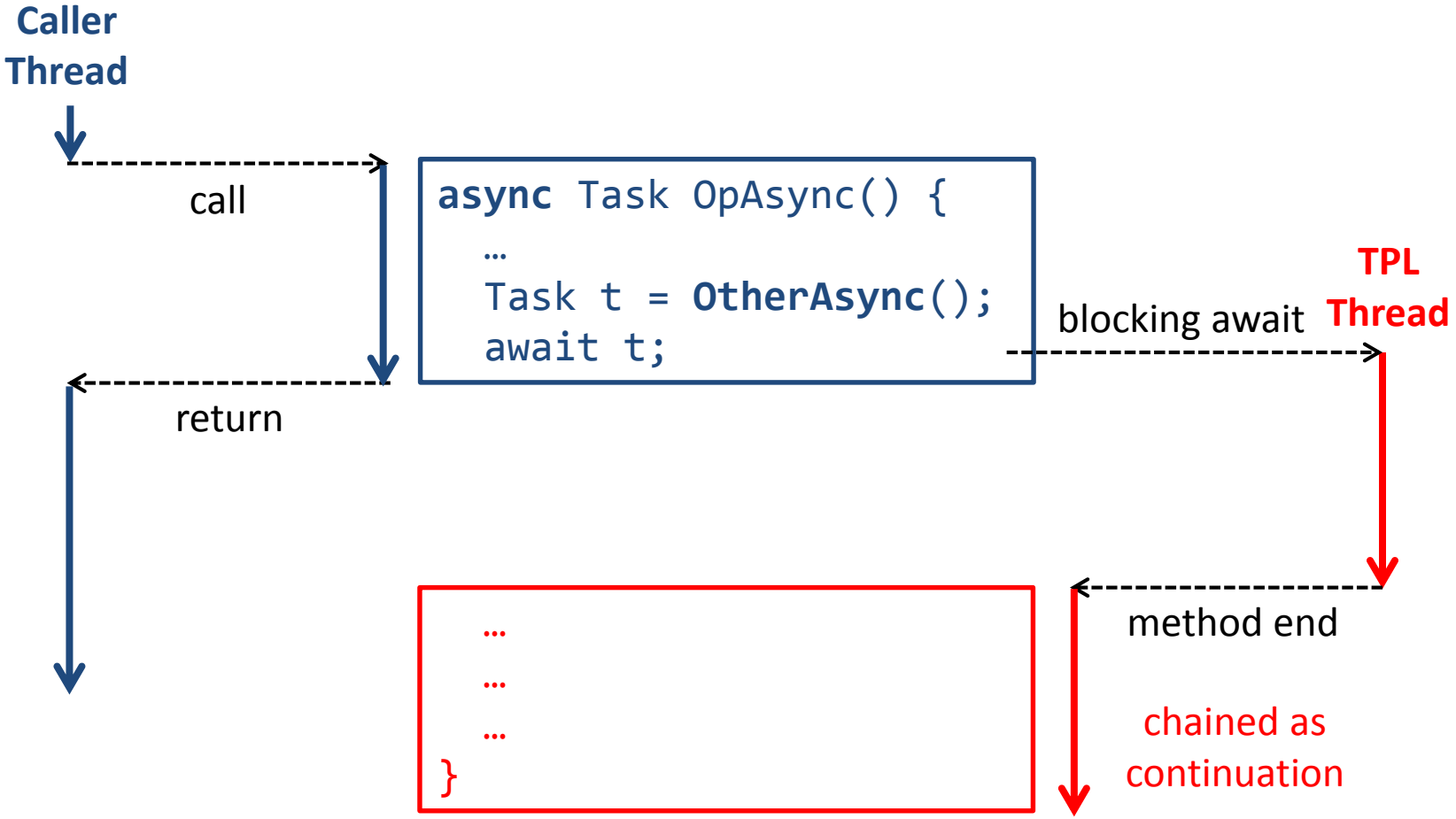
Async Method Call

- Method runs synchronously until a blocking await
  - Wait on other thread or IO
- Returns to caller upon blocking await



Case 1: Caller is a Non-GUI Thread

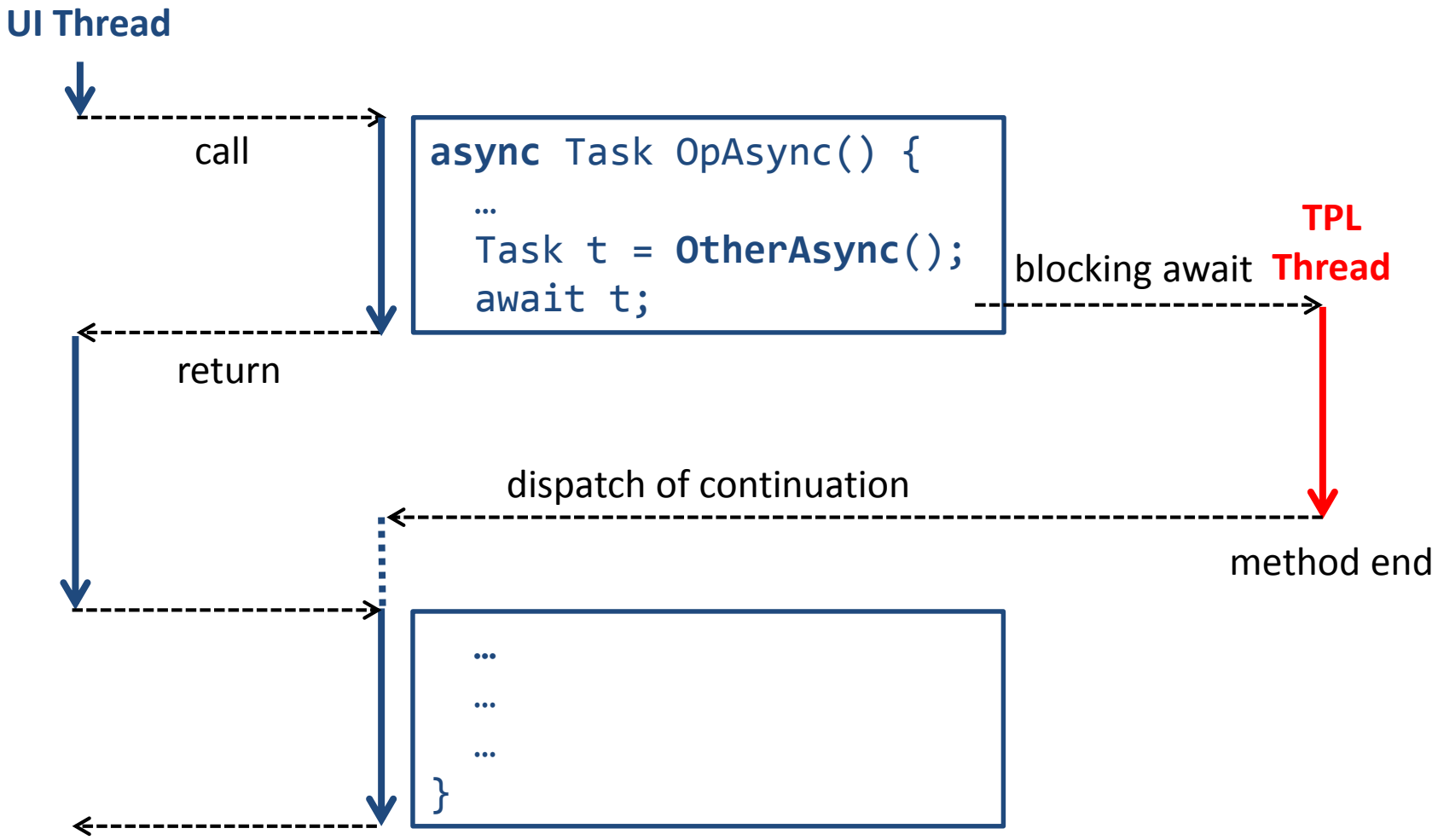
- Other thread continues execution after await



More precisely: Caller has no dispatching synchronization context

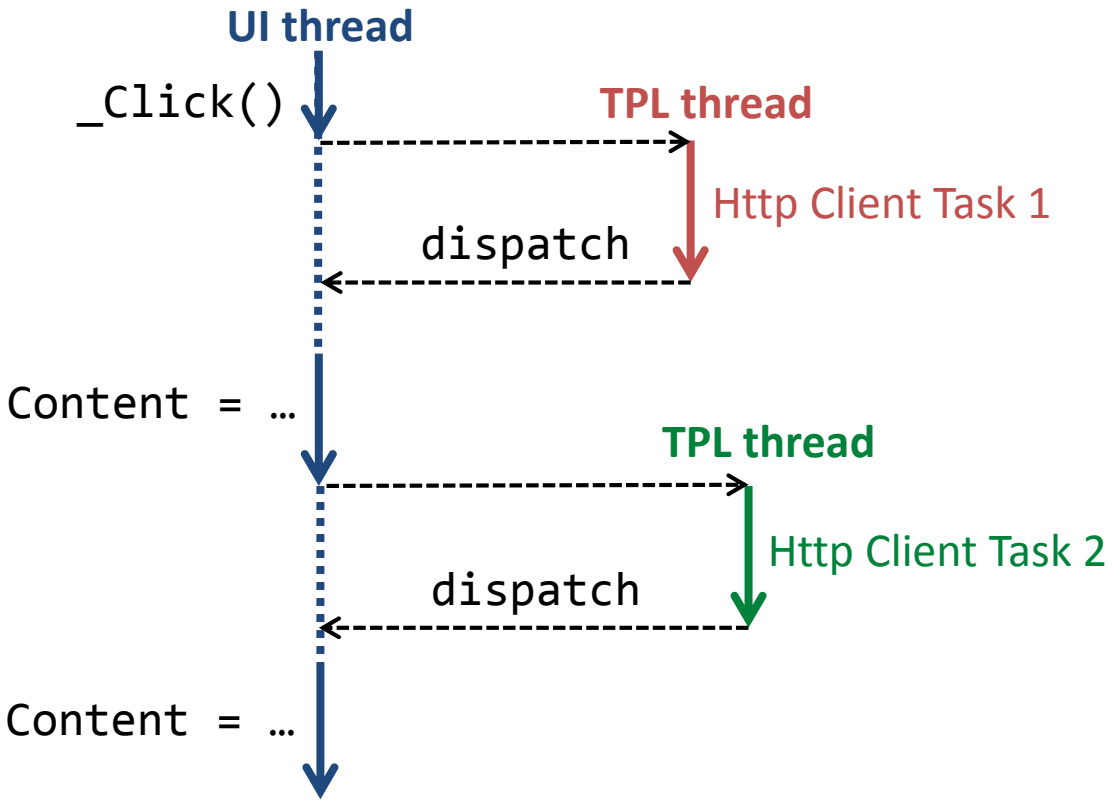
Case 2: Caller is GUI Thread

- Remainder is later dispatched on UI Thread



Non-Blocking & Coherent UI Logic

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



Caution: Notorious Pitfalls

1. Async method are not per se asynchronous
   - Use `Task.Run()` for long-running synchronous code
2. Thread switches within same method incarnation
   - Thread-local state no longer valid
3. Quasi-parallelism of UI event handling
   - await is equally tricky as old `DoEvents()`
4. Race conditions remain possible
   - Remember case 1 => test both cases
5. UI deadlocks immanent
   - No `task.Wait()`, `task.Result` in UI thread code

Conclusions

- TPL is particularly powerful because of its different abstractions on top of the thread pool

Abstraction	Ingredient	Focus
Task Parallelization	Explicit task start, wait, chain etc.	Complex task structures
Data parallelism	Parallel Invoke / Loops PLINQ	Declarative multi-core acceleration
Asynchronous programming	Async/await	Non-blocking logics/UI

- But beware of the pitfalls!
  - Concurrency errors, fire & forget, async/await, …

Thank You for Your Attention

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