Asynchronous Programing in .NET… from zero to hero

BY RICCARDO TERRELL

Asynchronous Programing in .NET… from zero to hero

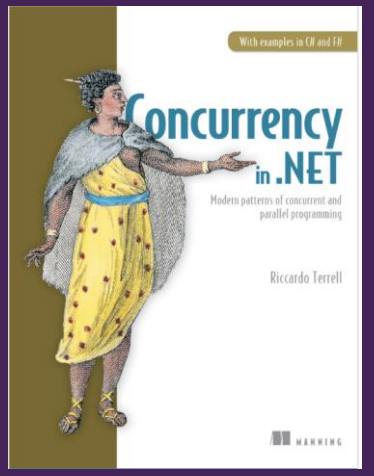
BY RICCARDO TERRELL

Objectives

- What / Why / Where Asynchronous Programming
- Increase scalability of code regardless the available cores
- Understand the implications of Asynchronous programming
- Combinators to compose Asynchronous operations deterministically
- Best Practices for exploiting Asynchronous programming
- Design your programs with **concurrency** in mind

Introduction - Riccardo Terrell

- Originally from Italy, currently - Living/working in Washington DC
- +/- 20 years in professional programming
  - C++/VB → Java → .Net C# → Scala → Haskell → C# & F# → ??
- Author of the book “Concurrency in .NET” – Manning Pub.
- *Polyglot programmer - believes in the art of finding the right tool for the job*
- *Organizer of the Pure Functional and DC F# User Group*

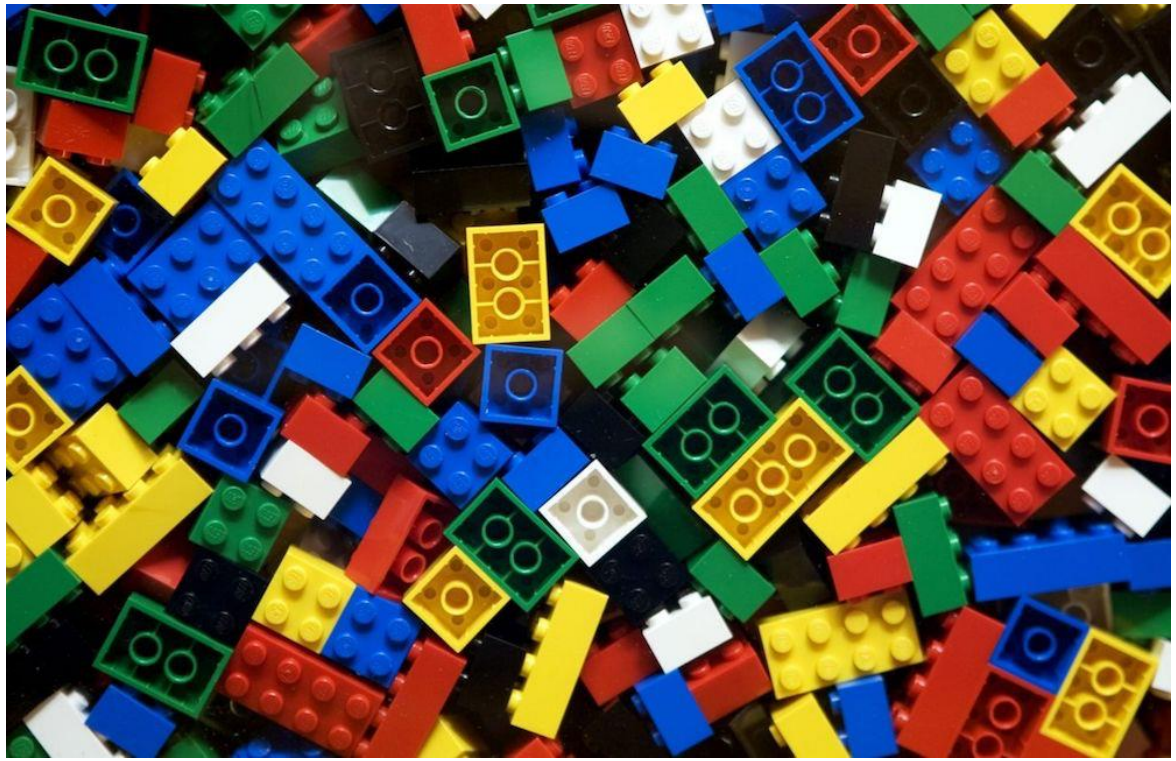


@trikace www.rickyterrell.com tericcardo@gmail.com

Strategies to parallelize the code

The first step in designing any parallelized system is **Decomposition**.

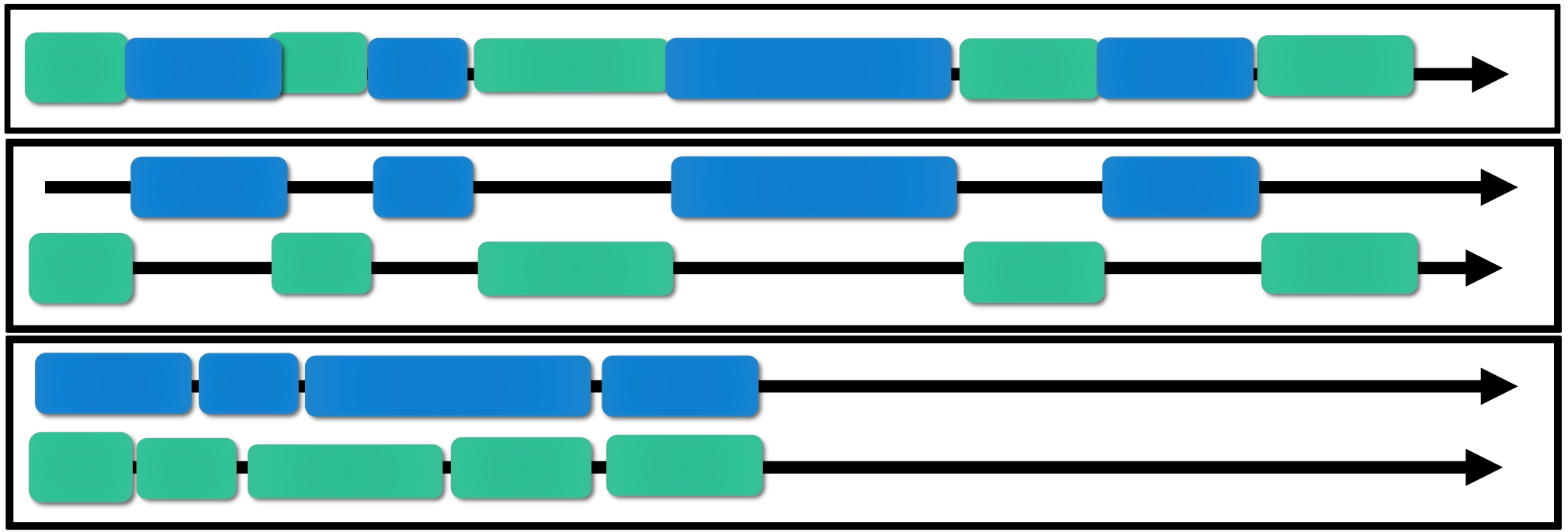
Decomposition is nothing more than taking a problem space and breaking it into discrete parts. When we want to work in parallel, we need to have at least two separate things that we are trying to run. We do this by taking our problem and decomposing it into parts.



Concurrency programming

Concurrency is the **composition** of independently executing computations.

Concurrency provides a way to **structure a solution** to solve a problem that may (but not necessarily) be parallelizable.



Concurrency is not Parallelism… ask the Chef

8:00 am

=

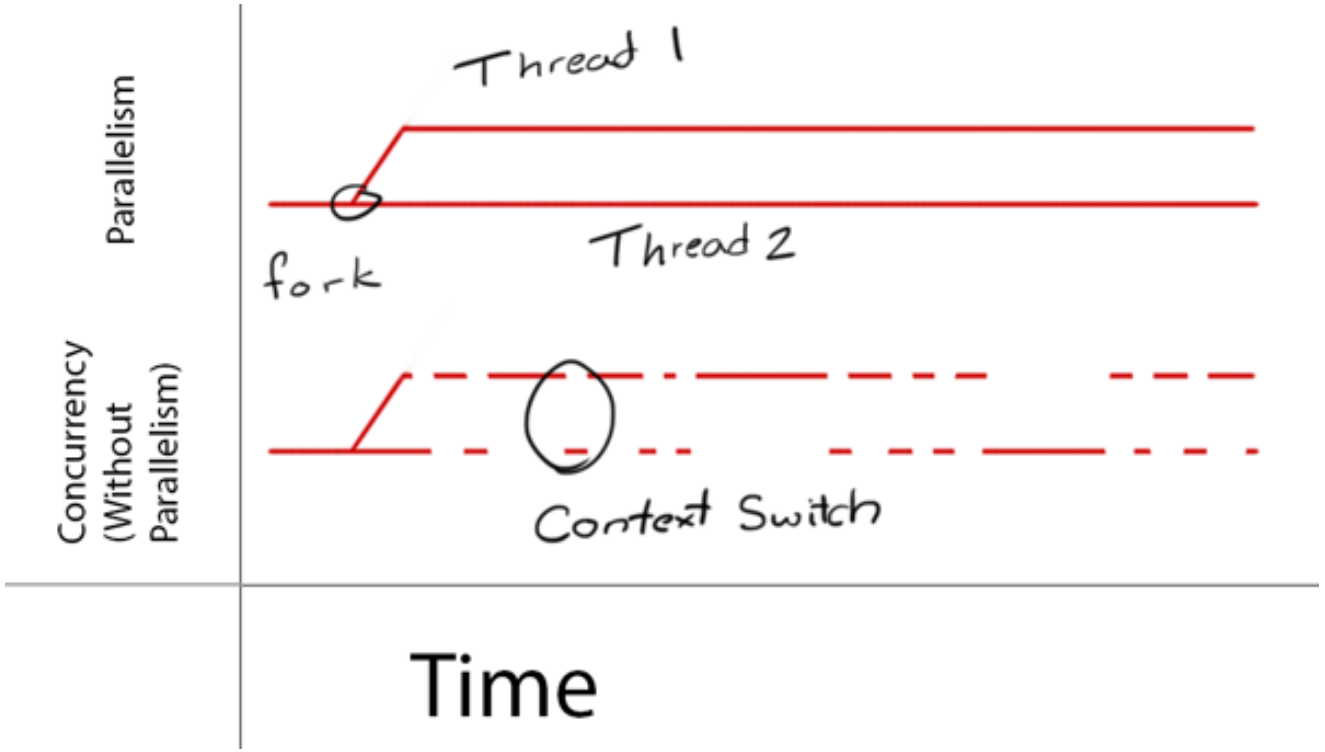
Concurrency is not **Parallelism**... ask the Chef

8:00 am

=

So tiring man!

Context Switches

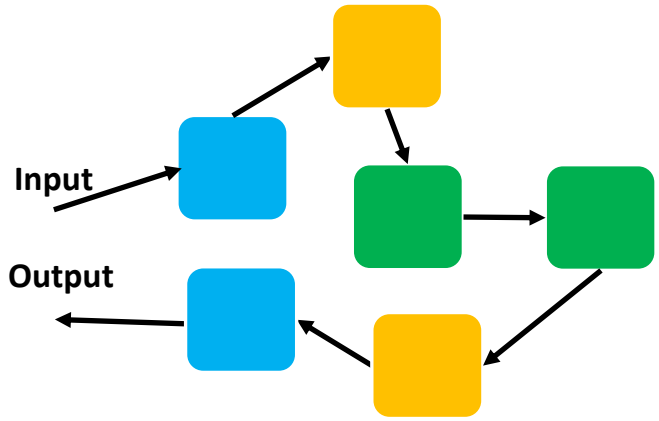


Concurrency is not **Parallelism**... ask the Chef

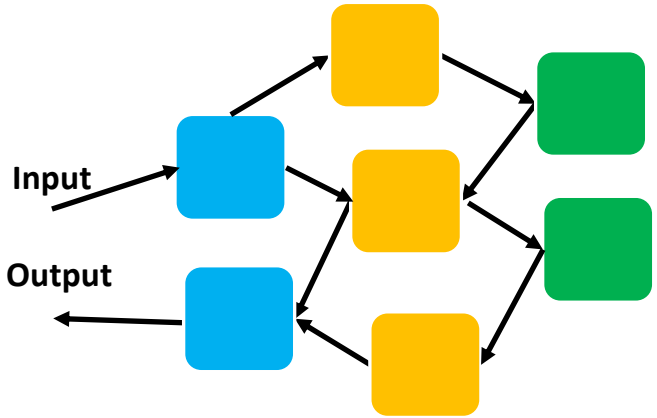
- **Concurrency is about dealing with lots of things at once.**
- **Parallelism is about doing lots of things at once.**

Different type of concurrency models lead to different challenges

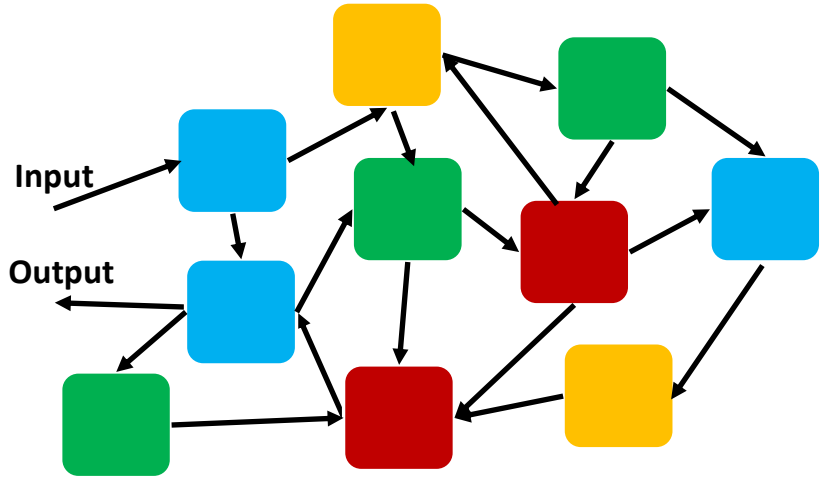
Sequential Programming



Task-Based Programming (Async)

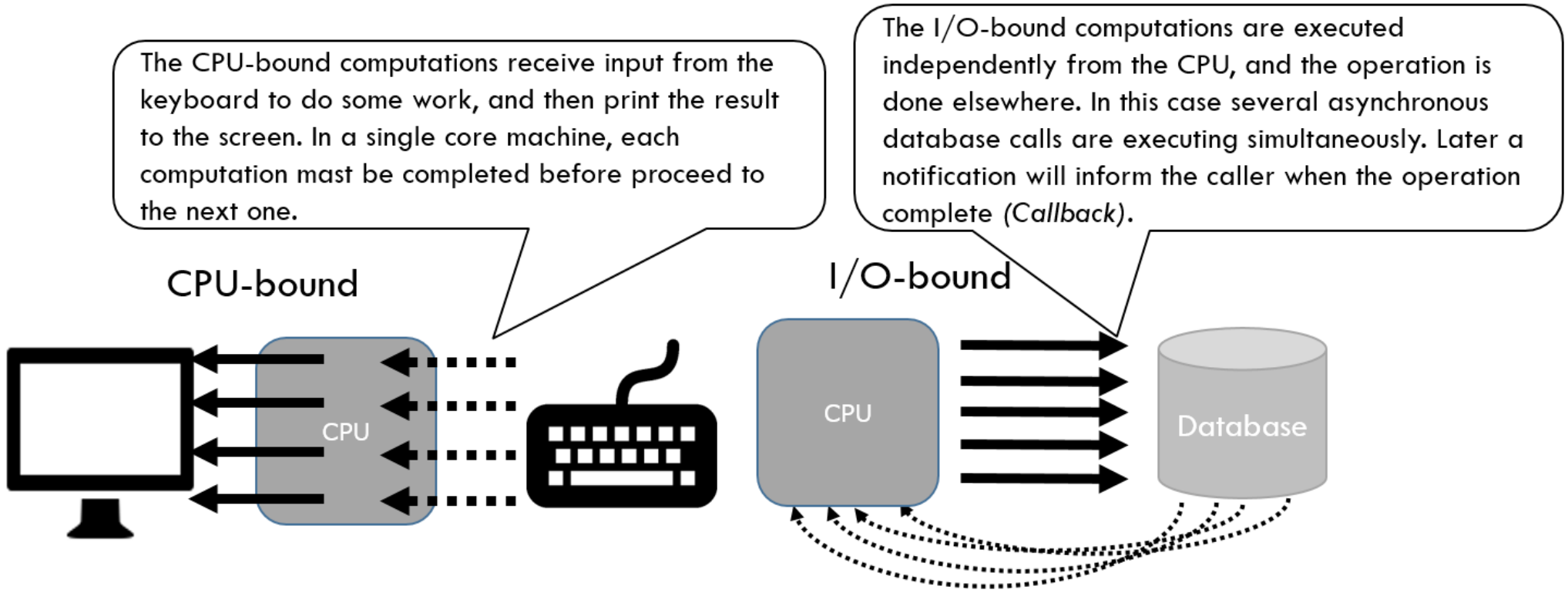


Message-Passing Programming



Is it CPU-bound, or I/O-bound?

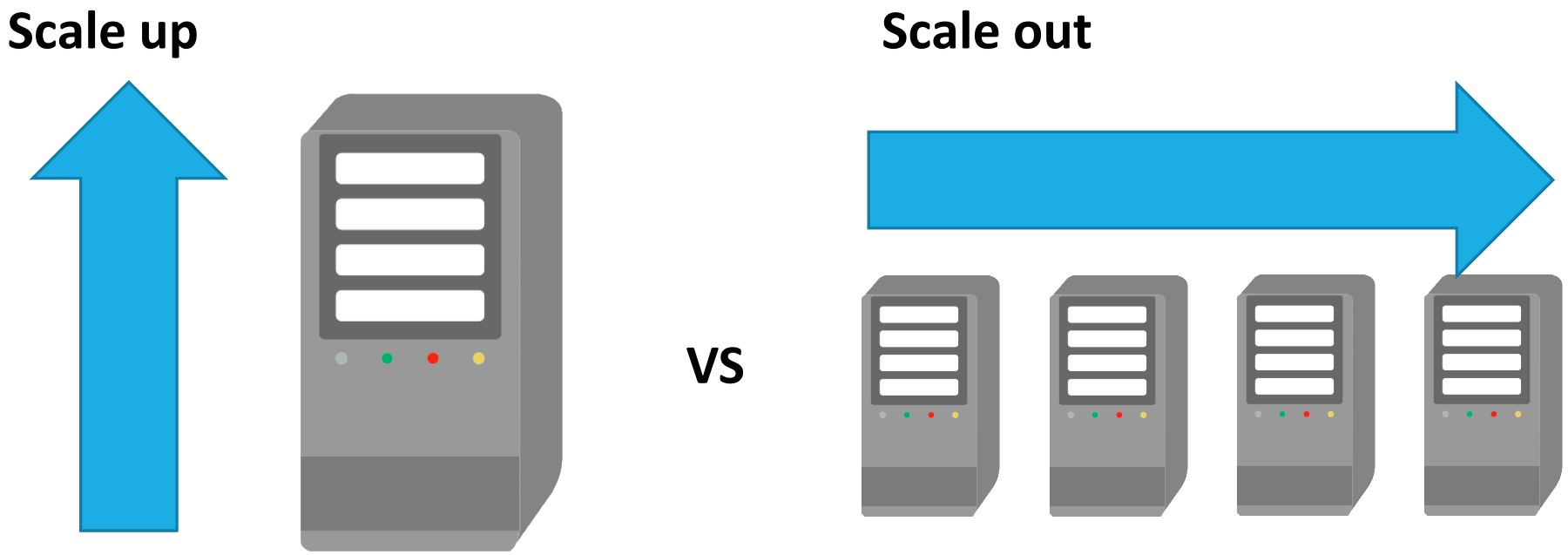
Asynchronous for I/O Bound



Asynchronous Workflows

- Software is often I/O-bound, it provides notable performance benefits
  - Connecting to the Database, Request web services, Read/Write on disks
- Not easy to predict when the operation will complete (non-deterministic)
- **IO bound operations can scale regardless of threads**
  - **This can even work a for huge numbers of computations**
  - **Unbounded** parallelism – no hardware constraints
  - **IO bound computations can often “overlap”**

Its about maximizing resource use



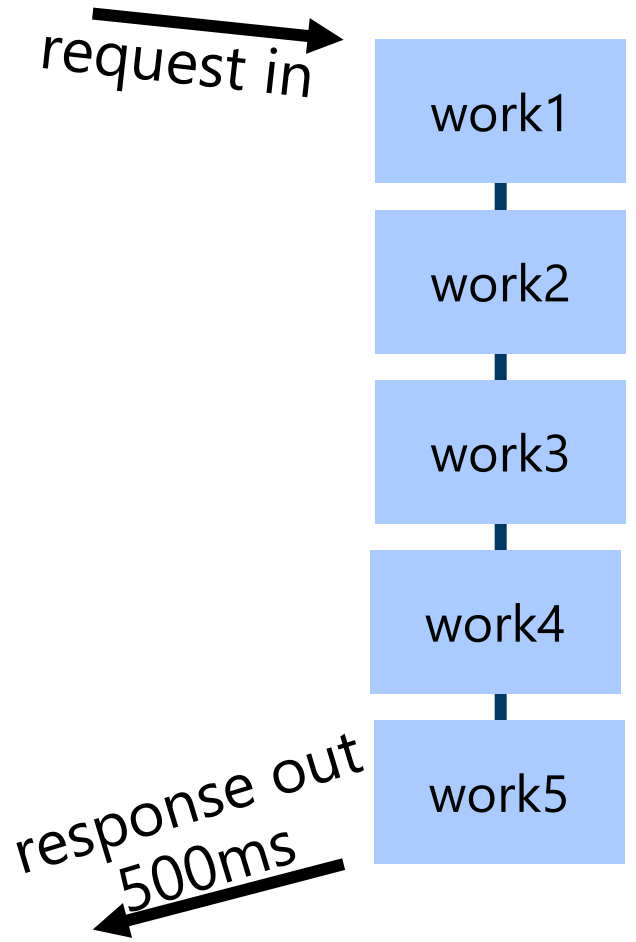
Scalability is the ability to cope and perform under an increasing workload

Running I/O operation using a Task

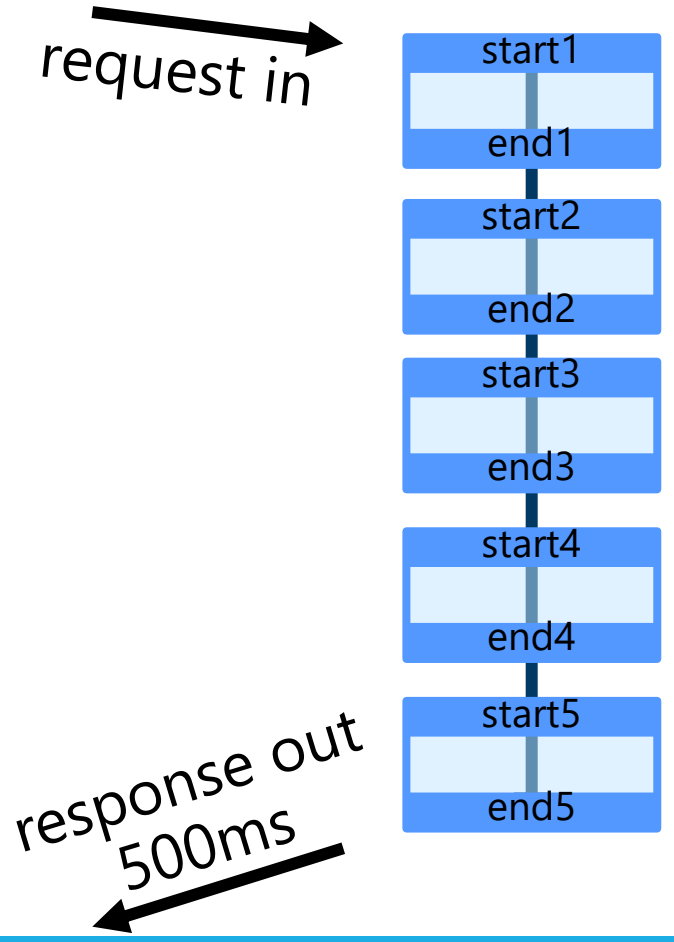
```
Task.Run(()=>
{
        using(var stream = File.Open(filePath))
        {
            byte[] buffer = new byte[16384];
            var bytesRead = stream.Read(buffer, 0, buffer.Length);
        }
});
```

ThreadPool

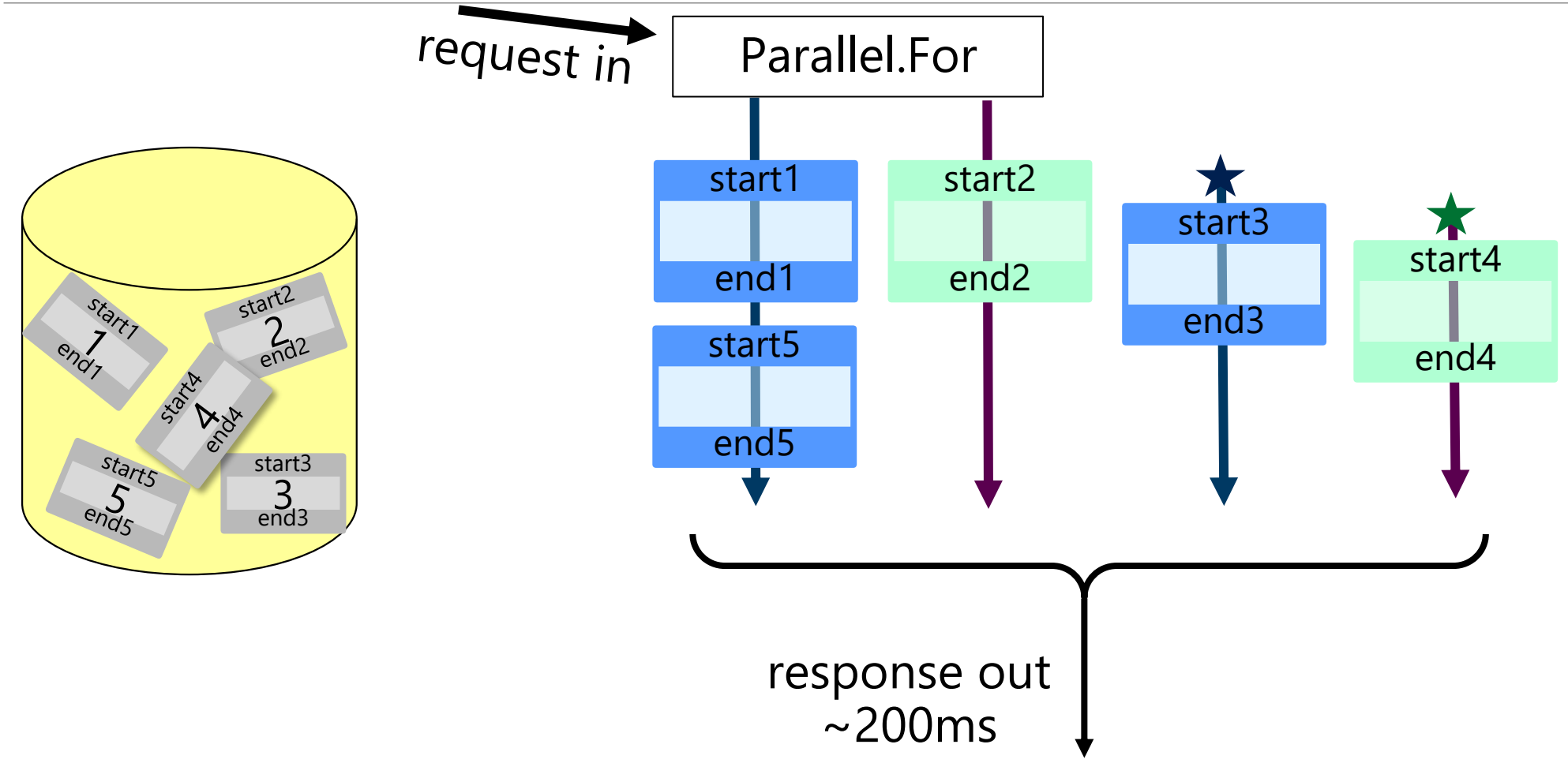
```
// table1.DataSource = LoadDataSequentially(1,5);
// table1.DataBind();

public List<Data> LoadDataSequentially(int first, int last)
{
    var loadedData = new List<Data>();

    for (int i = first; i <= last; i++) {
        Data data = Data.Deserialize(i);
        loadedData.Add(data);
    }

    return loadedData;
}
```



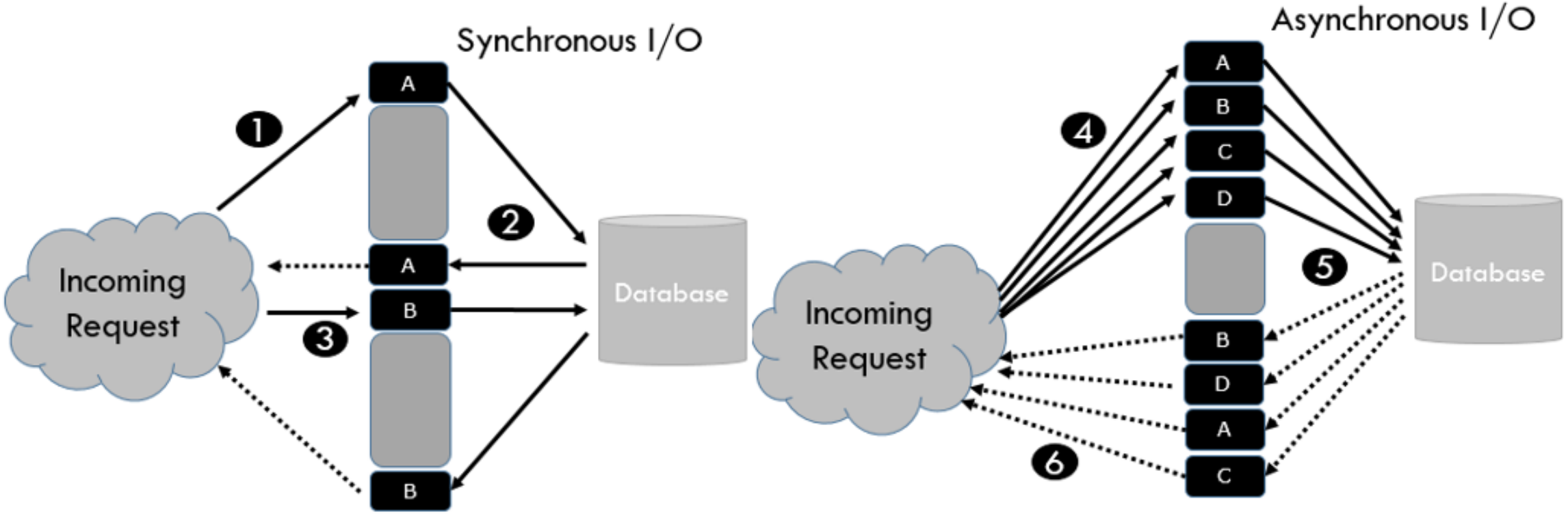
ThreadPool


request in
start1
end1
start2
end2
start3
end3
start4
end4
start5
end5
response out
500ms

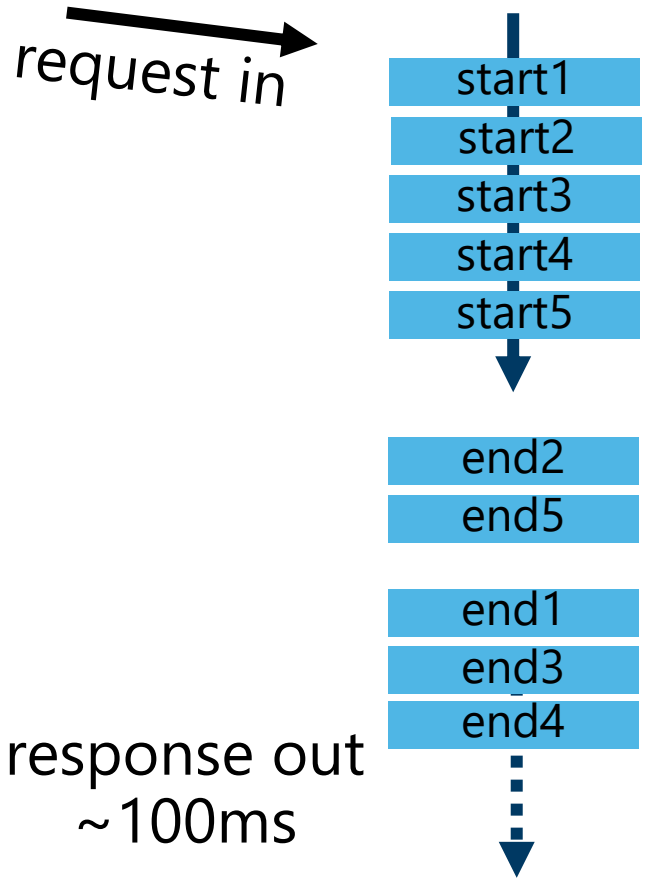
ThreadPool


request in
Parallel.For
start1
end1
start5
end5
start2
end2
start3
end3
start4
end4
start1
1
end1
start2
2
end2
start4
4
end4
start5
5
end5
start3
3
end3
response out
~200ms

Synchronous vs Asynchronous



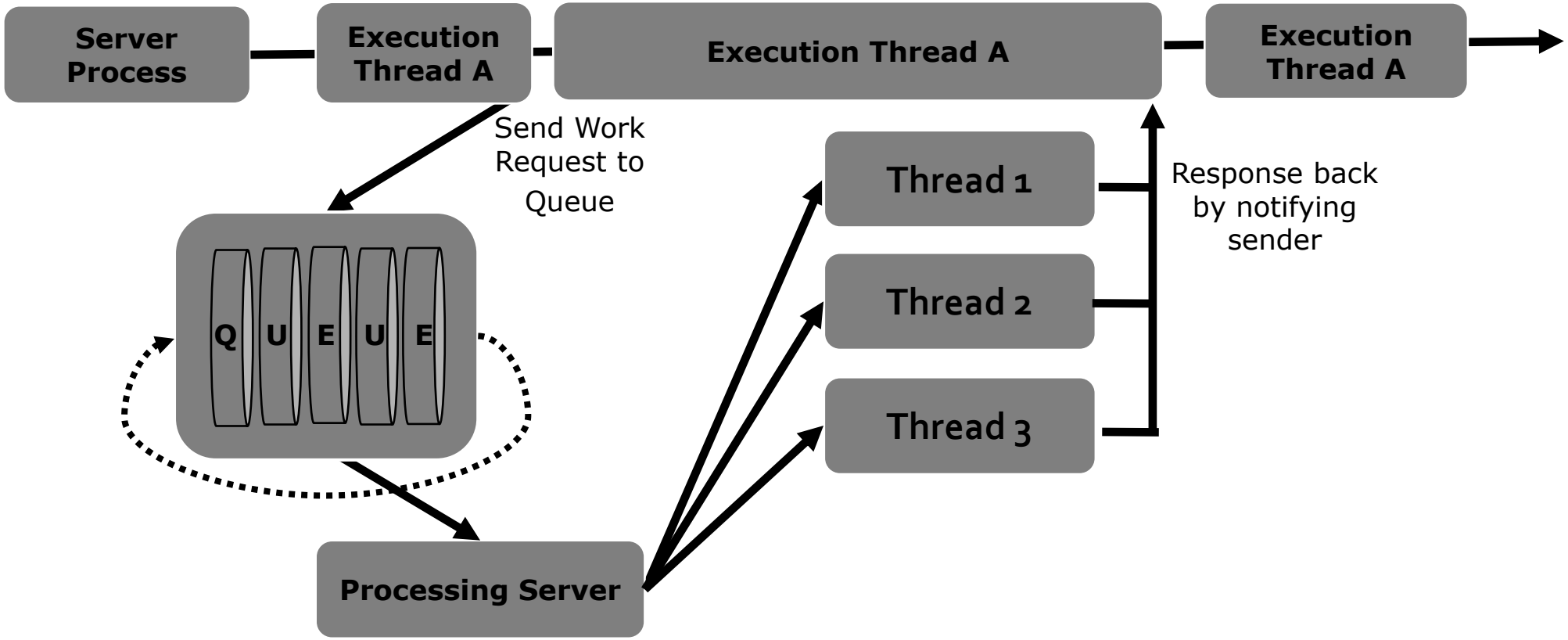
ThreadPool


request in
start1
start2
start3
start4
start5
end2
end5
end1
end3
end4
response out
~100ms

ThreadPool

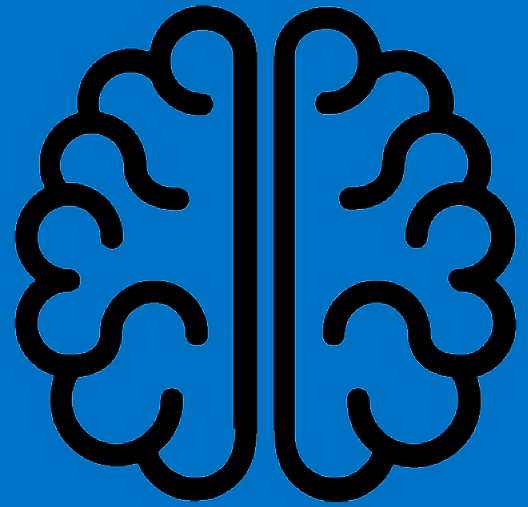
```
// table1.DataSource = await LoadDataAsync(1,5);
// table1.DataBind();

public async Task<List<Data>> LoadDataAsync(int first, int last)
{
    var tasks = new List<Task<Data>>();

    for (int i = first; i <= last; i++)
    {
        Task<Data> t = Data.LoadFromDatabaseAsync(i);
        tasks.Add(t);
    }

    Data[] loadedData = await Task.WhenAll(tasks);
    return loadedData.ToList();
}
```

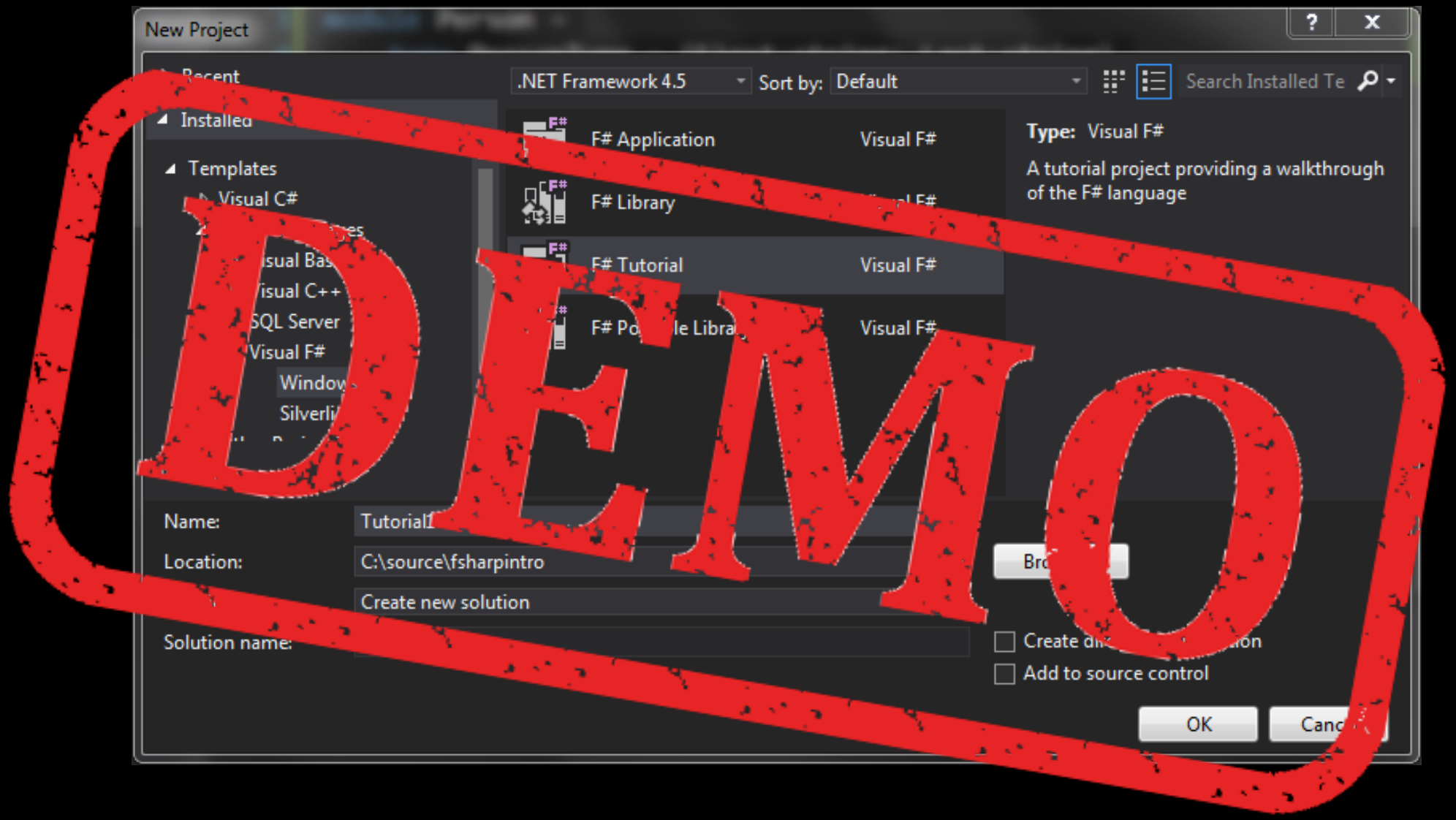
A different asynchronous pattern
Queuing work for later execution



Brain Teasers

?????




New Project
.NET Framework 4.5
Sort by:
Default
Search Installed Te
Installed
Templates
Visual C#
Visual C++
SQL Server
Visual F#
Silverli
F# Application
Visual F#
F# Library
F# Tutorial
Visual F#
Visual F#
Type: Visual F#
A tutorial project providing a walkthrough of the F# language
Name:
Tutorial1
Location:
C:\source\fsharpintro
Create new solution
Solution name:
Add to source control
OK
DEMO

Task (async/await) combinators

Task.WhenAll – rendezvous point

```
IEnumerable<MailAddress> addrs = new ...

IEnumerable<Task> asyncOps = from addr in addrs
                             select SendMailAsync(addr);


await Task.WhenAll(asyncOps);
```

Task.WhenAny – fine control

Redundancy: Doing an operation multiple times and selecting the one that completes first

Interleaving: Launching multiple operations and needing them all to complete, but processing them as they complete

Throttling: Allowing additional operations to begin as others complete

Early bailout: An operation represented by t1 can be grouped in a WhenAny with another task t2, and we can wait on the WhenAny task. t2 could represent a timeout, or cancellation, or some other signal that will cause the WhenAny task to complete prior to t1 completing

Task.WhenAny - Redundancy

```
var cts = new CancellationTokenSource();

var recommendations = new List<Task<bool>>() {
            GetBuyRecommendation1Async(symbol, cts.Token),
            GetBuyRecommendation2Async(symbol, cts.Token),
            GetBuyRecommendation3Async(symbol, cts.Token),
};

Task<bool> recommendation = await Task.WhenAny(recommendations);

cts.Cancel();

if (await recommendation) BuyStock(symbol);
```

Task.WhenAny - Interleaving

```
List<Task<Bitmap>> imageTasks =
        (from imageUrl in urls select GetBitmapAsync(imageUrl));

foreach(var imageTask in imageTasks)
{
        Bitmap image = await imageTask;
        imageTasks.Remove(imageTask);
        ProcessImage(image);
}
```

Catch the bug(s)

Task.WhenAny - Interleaving

```
List<Task<Bitmap>> imageTasks =
        (from imageUrl in urls select GetBitmapAsync(imageUrl)).ToList();

while(imageTasks.Count > 0)
{
        Task<Bitmap> imageTask = await Task.WhenAny(imageTasks);
        imageTasks.Remove(imageTask);
        Bitmap image = await imageTask;
        ProcessImage(image);
}
```

Task.WhenAny – Early Bailout

```
var fooCts = new CancellationTokenSource();
var delayCts = new CancellationTokenSource();

var foo = FooAsync(fooCts.Token);
var delay = Task.Delay(2500, delayCts.Token);

if(await Task.WhenAny(foo, delay) != foo)
{
        fooCts.Cancel();
        foo.ContinuationWith(t => /* observer t.Exception */ );
}

else

        delayCts.Cancel();
```

Reliable Tasks with Retry

```
async Task<T> Retry<T>(Func<Task<T>> task, int retries,
                       TimeSpan delay, CancellationToken? cts = null) =>

    await task().ContinueWith(async innerTask =>
    {
        cts?.ThrowIfCancellationRequested();
        if (innerTask.Status != TaskStatus.Faulted)
            return innerTask.Result;
        if (retries == 0)
            throw innerTask.Exception;
        await Task.Delay(delay, cts.Value);

        return await Retry(task, retries - 1, delay, cts.Value);
    }).Unwrap();
```

Retry in action

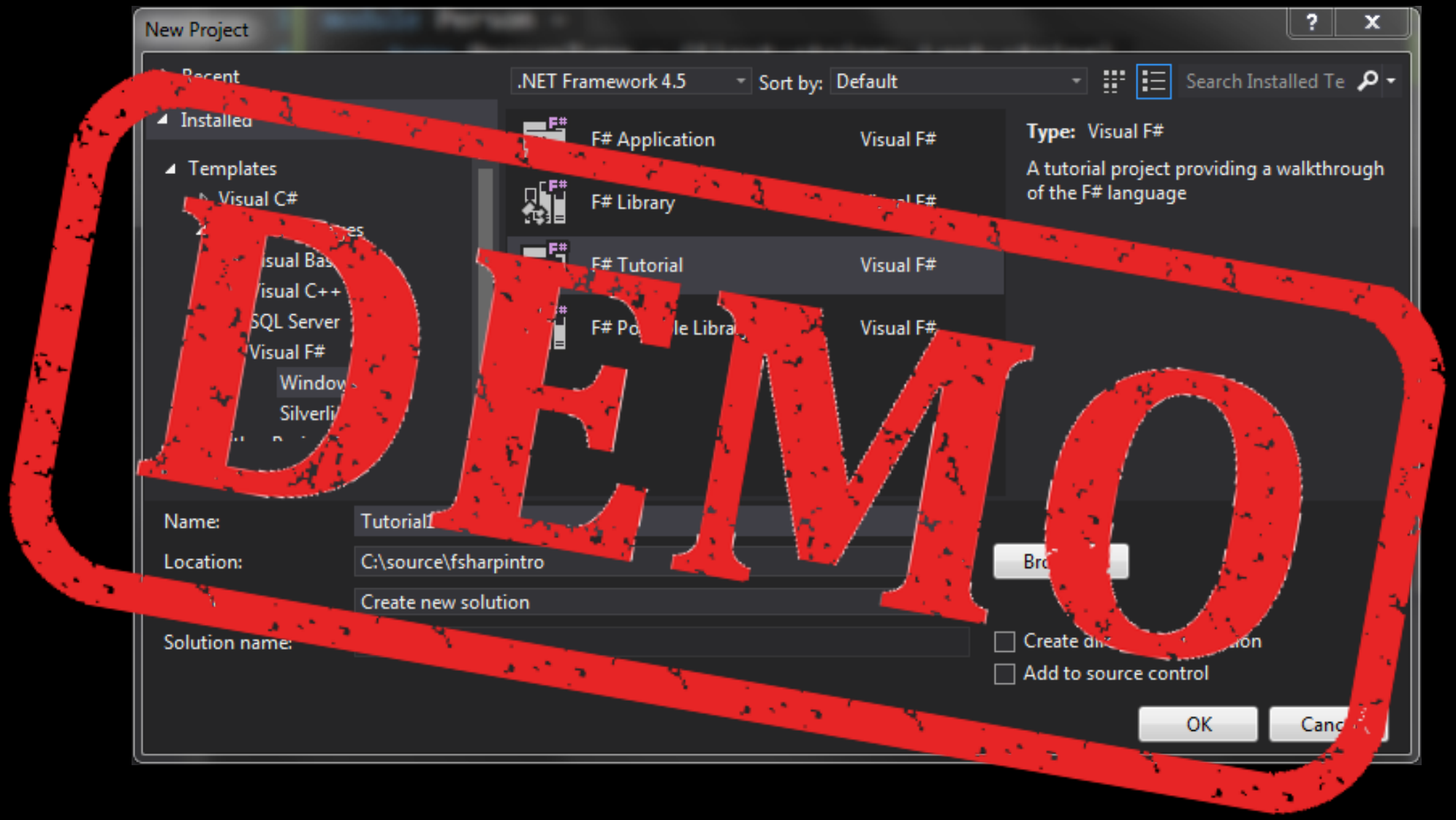
```
Image image = await Retry(async () =>
                    await DownloadImageAsync("Bugghina001.jpg")
                  , 5, TimeSpan.FromSeconds(2));
```

Reliable Tasks with Otherwise

```
async Task<T> Otherwise<T>(this Task<T> task, Func<Task<T>> orTask) =>
    await task.ContinueWith(async innerTask =>
    {
        if (innerTask.Status == TaskStatus.Faulted) return await orTask();
        return await Task.FromResult<T>(innerTask.Result);
    }).Unwrap();
```

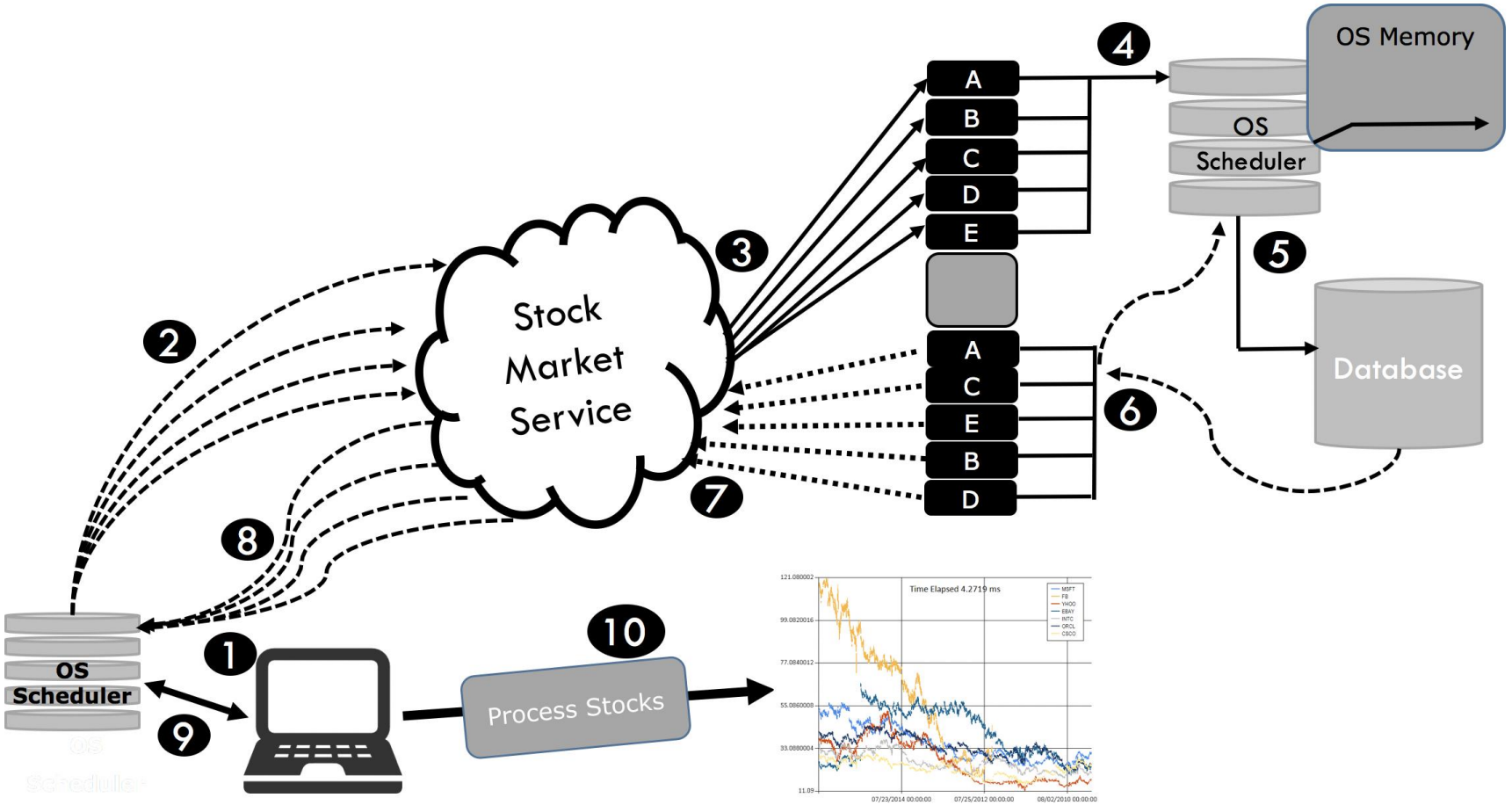
Otherwise in action

```
var image = await Retry(async () =>
                    await DownloadImageAsync("Bugghina001.jpg")
                .Otherwise(async () =>
                    await DownloadImageAsync("Bugghina002.jpg")),
            5, TimeSpan.FromSeconds(2));
```


New Project
.NET Framework 4.5
Sort by:
Default
Search Installed Te
Installed
Templates
Visual C#
Visual C++
SQL Server
Visual F#
Silverli
F# Application
Visual F#
F# Library
F# Tutorial
Visual F#
Visual F#
Type: Visual F#
A tutorial project providing a walkthrough of the F# language
Name:
Tutorial1
Location:
C:\source\fsharpintro
Create new solution
Solution name:
Add to source control
OK
DEMO

Process Tasks as they complete

Asynchronous Parallel Operations
(Stock-Tickers)

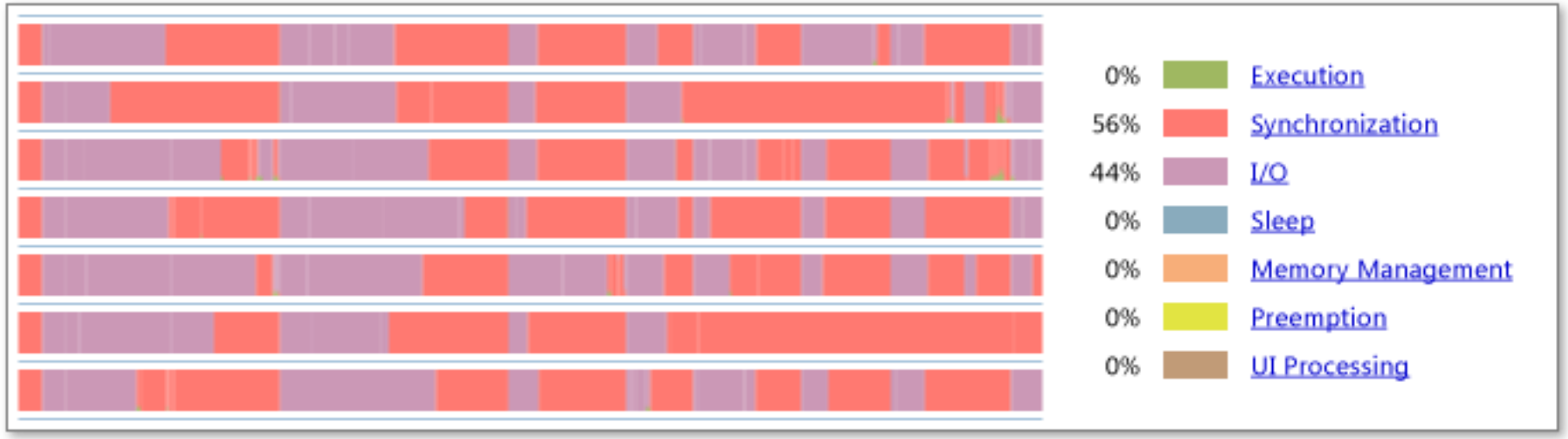


Handle Tasks as they complete

```
string[] stocks = new[] { "MSFT", "FB", "AAPL", "GOOG", "ORCL" };

List<Task<Tuple<string, StockData[]>>> stockHistoryTasks =
                                                stocks.Select(ProcessStockHistory).ToList();

while (stockHistoryTasks.Count > 0)
{
    Task<Tuple<string, StockData[]>> stockHistoryTask = await
                                                        Task.WhenAny(stockHistoryTasks);

    stockHistoryTasks.Remove(stockHistoryTask);

    Tuple<string, StockData[]> stockHistory = await stockHistoryTask;
    await ProcessStock(stockHistory);

}
```

Throttling Async

Inefficient I/O



Performance overheads with Task.WhenAny & Task.WhenAll

```
static async Task ExecuteInParallel(this IEnumerable<T> collection,
                                    Func<T, Task> processor,
                                          int degreeOfParallelism)
{
    var queue = new ConcurrentQueue<T>(collection);
    var tasks = Enumerable.Range(0, degreeOfParallelism).Select(async _ =>
    {
        T item;
        while (queue.TryDequeue(out item))
        {
            await processor(item);
        }
    });


    await Task.WhenAll(tasks);
}
```

The RequestGate

```
public class RequestGate
{
    SemaphoreSlim semaphore;
    public RequestGate(int count) =>
        semaphore = new SemaphoreSlim(initialCount: count, maxCount: count);

    public async Task<IDisposable> AsyncAcquire(TimeSpan timeout,
                                   CancellationToken cancellationToken = new CancellationToken())
    {
        var ok = await semaphore.WaitAsync(timeout, cancellationToken);
        if (ok)
             Thread.BeginCrtiticalRegion();
            return new SemaphoreSlimRelease(semaphore);
        throw new Exception("couldn't acquire a semaphore");
    }
    private class SemaphoreSlimRelease : IDisposable
    {
        SemaphoreSlim semaphore;
        public SemaphoreSlimRelease(SemaphoreSlim semaphore) => this.semaphore = semaphore;
        public void Dispose() {
                Thread.EndCrtiticalRegion();
                semaphore.Release();
    }
}
```

Asynchronous Caching

```
static Func<T, Task<R>> MemoizeLazyTaskSafe<T, R>(Func<T, Task<R>> func)
{
    ConcurrentDictionary<T, Lazy<Task<R>>> cache = new ConcurrentDictionary<T, Lazy<Task<R>>>();
    return arg => cache.GetOrAdd(arg, a => new Lazy<Task<R>>(() => func(a))).Value;
}

var client = new HttpClient();
var getData = MemoizeLazyTaskdSafe<string, string>(url =>
{
  Console.WriteLine($"Get String Async {url} - {DateTime.Now.ToString("hh:mm:ss.fff")}");
  return client.GetStringAsync(url);
});

var result_1 = await getData("http://www.google.com");
var result_2 = await getData("http://www.microsoft.com");
var result_3 = await getData("http://www.google.com");
var result_4 = await getData("http://www.microsoft.com");
```

Asynchronous Streams

Asynchronous Streams and Task<T>

Asynchronous Streams and IEnumerable<T>

Asynchronous Streams and Task<IEnumerable<T>>

Asynchronous Streams and IObservable<T> (backpressure)

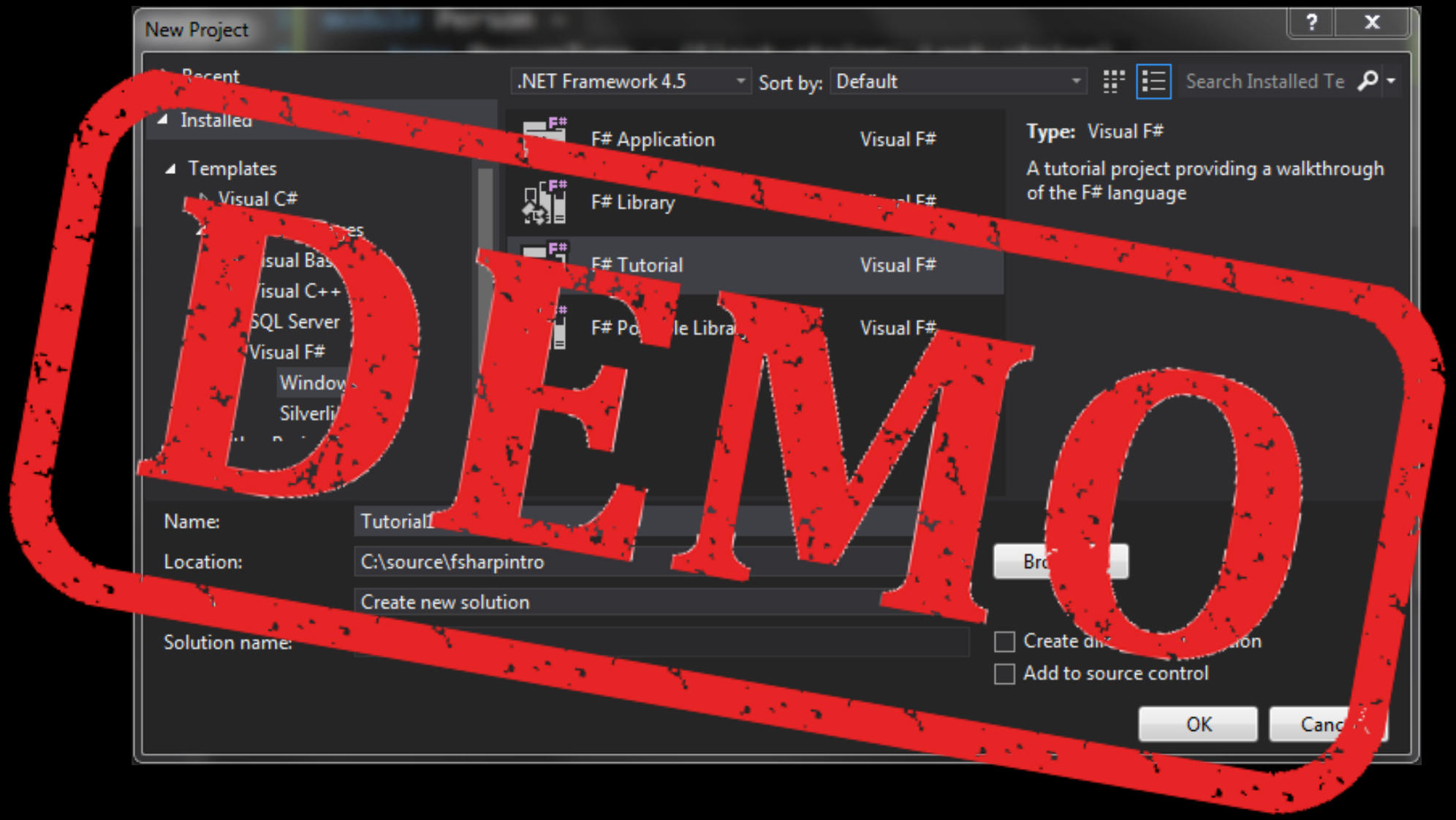
Type	Single or Multiple Value	Asynchronous or Synchronous	Push or Pull
T	Single Value	Synchronous	N/A
IEnumerable<T>	Multiple Values	Synchronous	N/A
Task<T>	Single Value	Asynchronous	Pull
IAsyncEnumerable<T>	Multiple Values	Asynchronous	Pull
IObservable<T>	Single or Multiple	Asynchronous	Push

you can call ToAsyncEnumerable() on any IEnumerable<T>, and then you'll have an asynchronous stream interface

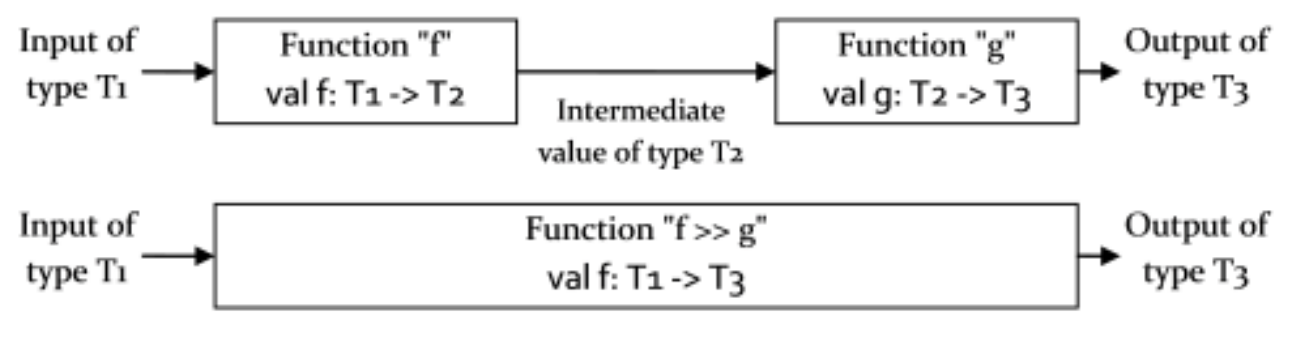
Asynchronous Streams

Creating Asynchronous Streams

```
async IAsyncEnumerable<string> GetValuesAsync(HttpClient client)
{
    int offset = 0;
    const int limit = 10;
    while (true)
    {
        // Get the current page of results and parse them
        string result = await client.GetStringAsync(
            $"https://example.com/api/values?offset={offset}&limit={limit}");

        string[] valuesOnThisPage = result.Split('\n');

        // Produce the results for this page
        foreach (string value in valuesOnThisPage)
            yield return value;

        // If this is the last page, we're done
        if (valuesOnThisPage.Length != limit)
            break;

        // Otherwise, proceed to the next page
        offset += limit;
    }
}


public async Task ProcessValueAsync(HttpClient client)
{
    await foreach (string value in GetValuesAsync(client))
    {
        Console.WriteLine(value);
    }
}
```


New Project
.NET Framework 4.5
Sort by:
Default
Search Installed Te
Installed
Templates
Visual C#
Visual C++
SQL Server
Visual F#
Silverli
F# Application
Visual F#
F# Library
F# Tutorial
Visual F#
Visual F#
Type: Visual F#
A tutorial project providing a walkthrough of the F# language
Name:
Tutorial1
Location:
C:\source\fsharpintro
Create new solution
Solution name:
Add to source control
OK
DEMO

Patterns for Tasks composition



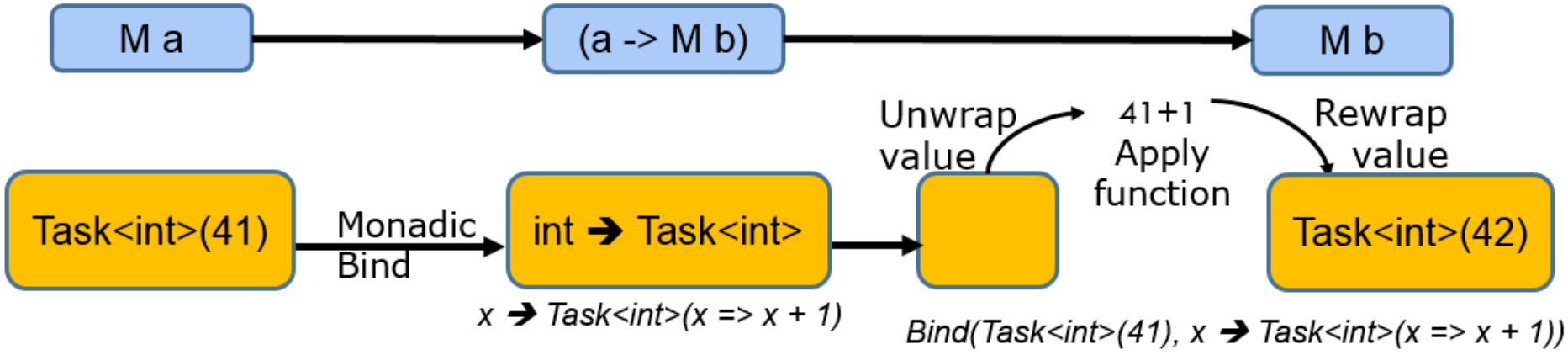
How can we compose Tasks?

```
 Task<int> RunOne() => // ...
Task<decimal> RunTwo(int input) => // ...

var result = RunTwo(RunOne()); // Error!!
```

```
Task<string> RunOne(int input) =>  // ...
Task<bool> RunTwo(string input) => // ...

Func<T1, T3> Compose<T1, T2, T3>(Func<T1, T2> f1,
                      Func<T2, T3> f2) => a => f2(f1(a));

var result = RunOne(42).Compose(RunTwo); // Error!!
```

Composing Tasks

```
Task<R> Bind<T, R>(this Task<T> m, Func<T, Task<R>> k) …
```



Task async/await composition

```
static Task<T> Return<T>(T task)=> Task.FromResult(task);


static async Task<R> Bind<T, R>(this Task<T> task, Func<T, Task<R>> cont)
     => await cont(await task.ConfigureAwait(false)).ConfigureAwait(false);


static async Task<R> Map<T, R>(this Task<T> task, Func<T, R> map)
=> map(await task.ConfigureAwait(false));
```

IEnumerable SelectMany

```
IEnumerable<R> SelectMany<T, R>( this IEnumerable<T> source,
                                        Func<T, IEnumerable<R>> selector )

IEnumerable<R> SelectMany<T, M, R>( this IEnumerable<T> source,
                Func<T, IEnumerable<M>> collectionSelector, Func<T, M, R> resultSelector )

IEnumerable<R> Select<T, R>( this IEnumerable<T> source, Func<T, R> selector )
```

Task SelectMany

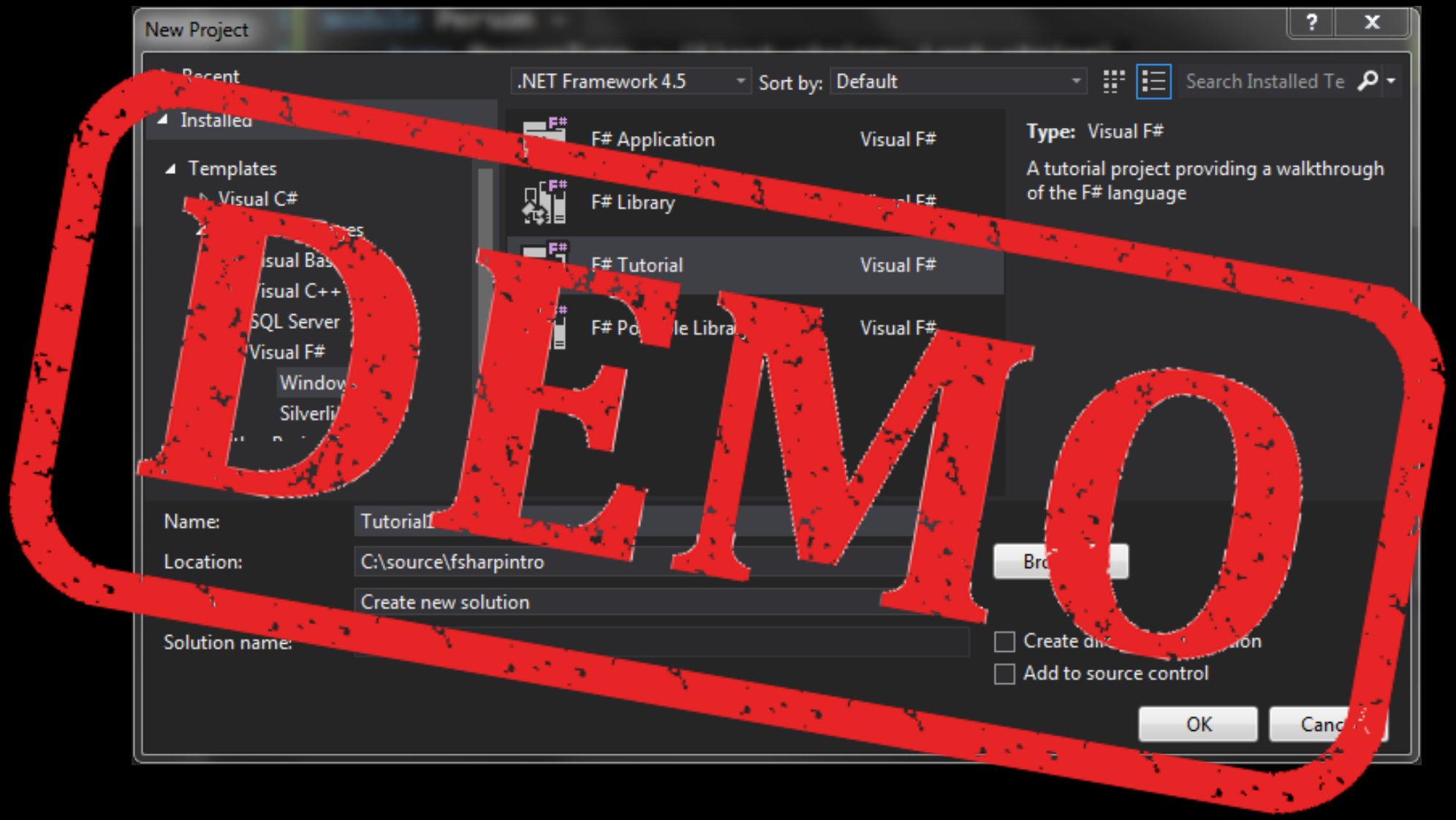
```
Task<R> Task<T, R>( this Task<T> source,
                                        Func<T,  Task<R>> selector )

Task<R> SelectMany<T, M, R>( this Task<T> source,
                Func<T,  Task<M>> collectionSelector, Func<T, M, R> resultSelector )

Task<R> Select<T, R>( this Task<T> source, Func<T, R> selector )
```

Composing Task async/await

```
static async Task<R> SelectMany<T, R>(this Task<T> task,
        Func<T, Task<R>> then) => await Bind(await task);


static async Task<R> SelectMany<T1, T2, R>(this Task<T1> task,
         Func<T1, Task<T2>> bind, Func<T1, T2, R> project)
        {
            T taskResult = await task;
            return project(taskResult, await bind(taskResult));
        }


static async Task<R> Select<T, R>(this Task<T> task, Func<T, R> project)
            => await Map(task, project);


static async Task<R> Return<R>(R value) => Task.FromResult(value);
```

Composing Tasks

```
from image in Task.Run<Emgu.CV.Image<Bgr, byte>()
from imageFrame in Task.Run<Emgu.CV.Image<Gray, byte>>()
from faces in Task.Run<System.Drawing.Rectangle[]>()
select faces;
```


New Project
.NET Framework 4.5
Sort by:
Default
Search Installed Te
Installed
Templates
Visual C#
Visual C++
SQL Server
Visual F#
Silverli
F# Application
Visual F#
F# Library
F# Tutorial
Visual F#
Visual F#
Type: Visual F#
A tutorial project providing a walkthrough of the F# language
Name:
Tutorial1
Location:
C:\source\fsharpintro
Create new solution
Solution name:
Add to source control
OK
DEMO

Few async/await good practices

Async void is only for event handlers!

Principles

- Async void is a “fire-and-forget” mechanism…
- The caller is unable to know when an async void has finished
- The caller is unable to catch exceptions thrown from an async void (instead they get posted to the UI message-loop)

Guidance

- Use async void methods only for top-level event handlers (and their like)
- Use async Task-returning methods everywhere else
- If you need fire-and-forget elsewhere, indicate it explicitly e.g. “FredAsync().FireAndForget()” When you see an async lambda, verify it
- Async all the way!

SynchronizationContext: ConfigureAwait

Task.ConfigureAwait(bool continueOnCapturedContext)

await t.ConfigureAwait(true) ***// default***

Post continuation back to the current context/scheduler

await t.ConfigureAwait(false)

If possible, continue executing where awaited task completes

Implications

- Performance (avoids unnecessary thread marshaling)
- Deadlock (code shouldn't block UI thread, but avoids deadlocks if it does)

Use ConfigureAwait(false)

Principles

- SynchronizationContext is captured before an await, and used to resume from await.
- In a library, this is an unnecessary perf hit.
- It can also lead to deadlocks if the user (incorrectly) calls Wait() on your returned Task.

Guidance

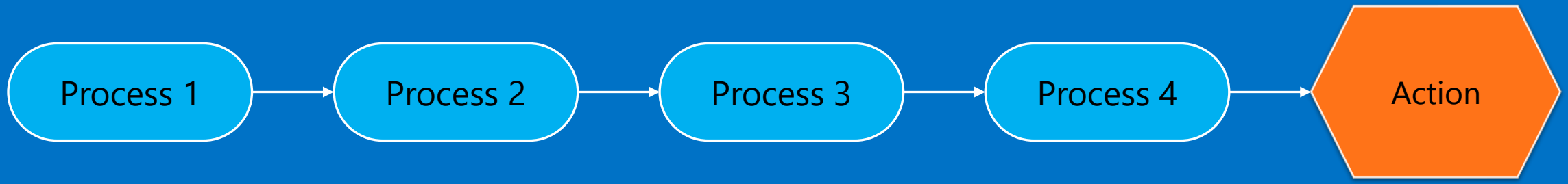
- In library methods, use "await t.ConfigureAwait(false);"

Configure Context – performance implications

```
async Task MyMethodAsync()
{
    // Code here runs in the original context.
    await Task.Delay(1000);

    // Code here runs without the original
    // context (in this case, on the thread pool).
    await Task.Delay(1000).ConfigureAwait(continueOnCapturedContext: false);
}
```

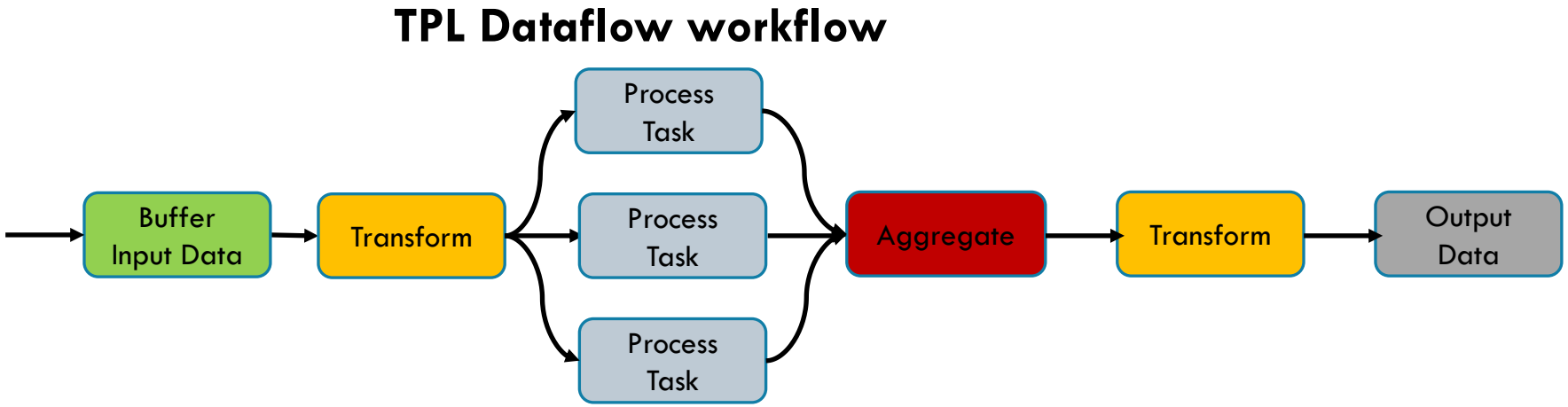

That's all Folks!

TPL Dataflow Agent in .NET and C#



TPL DataFlow blocks –design to compose

for High throughput, low-latency scenarios



TPL DataFlow blocks –design to compose

for High throughput, low-latency scenarios