# Choose the concurrency model in .NET

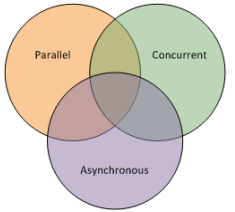
*"I have no special talents. I am only passionately curious."*

*- Albert Einstein*

Riccardo Terrell

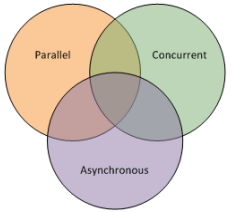
Boston Code Camp 25

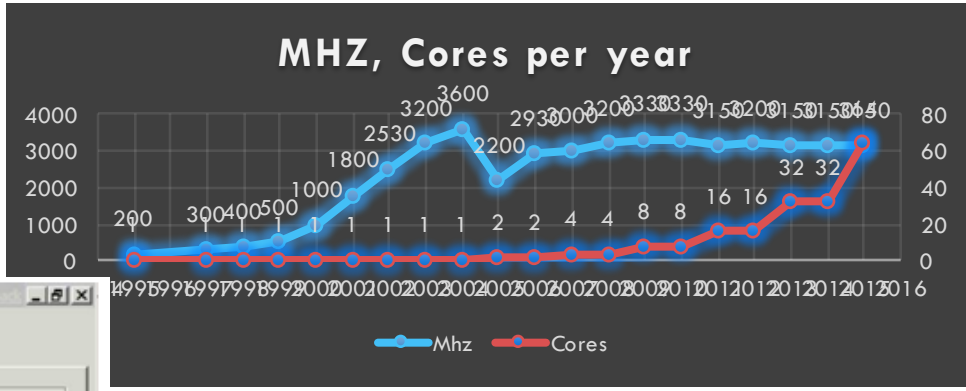
# Agenda

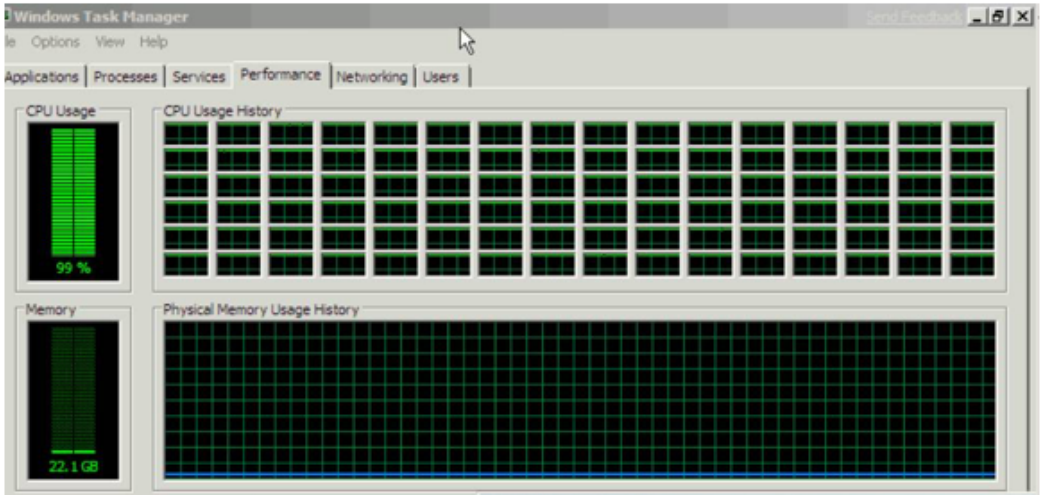



- What is concurrency – parallelism – asynchronous - multithreading
- Different programming model (pro & cons)
- Code Sample
- Code Sample – Combining different programming model

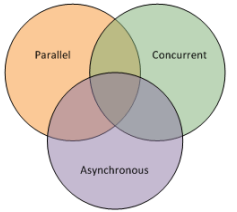
# Why Parallelism

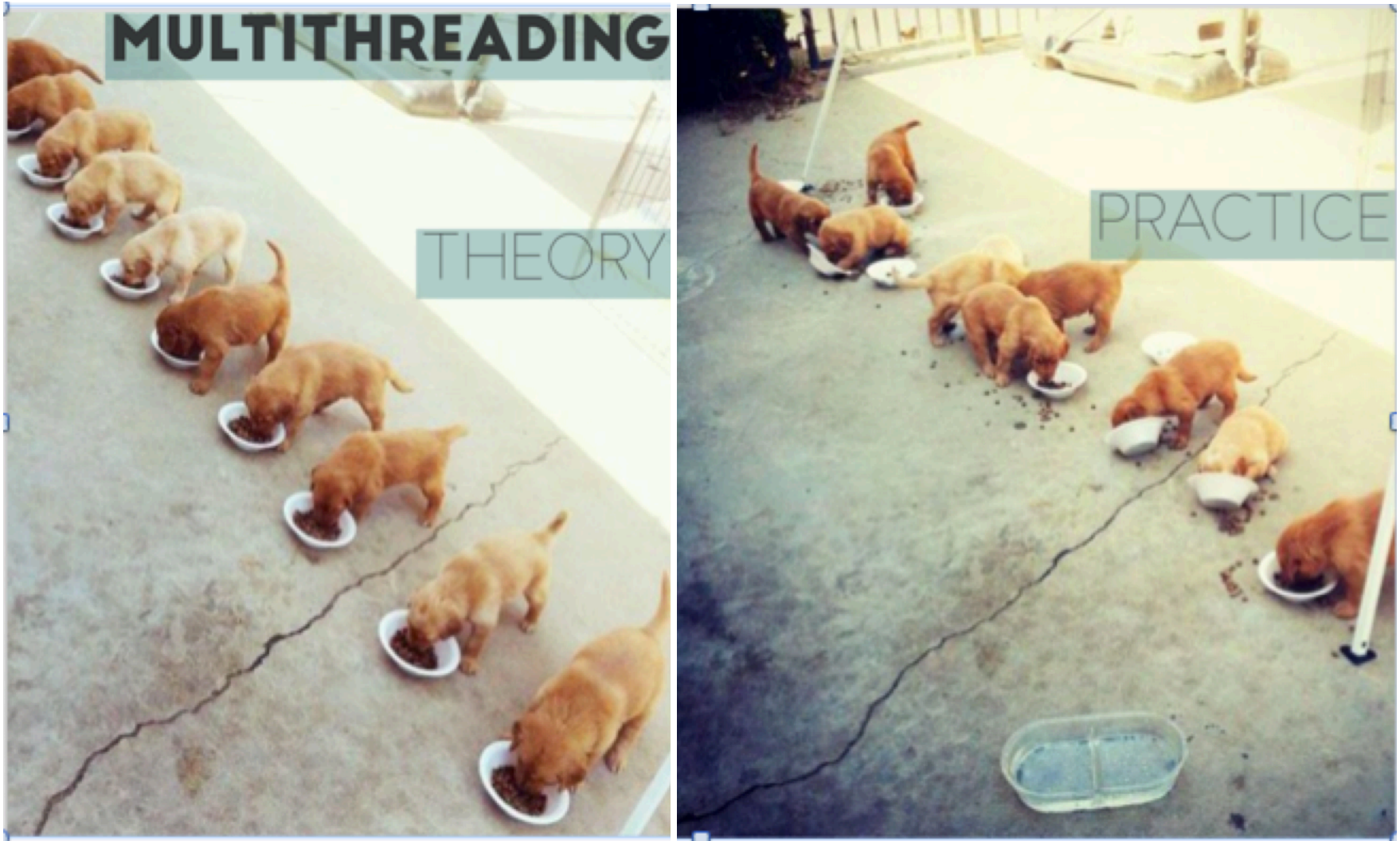


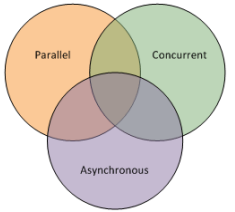



# The need Parallelism





MULTITHREADING
THEORY
PRACTICE

# Goals



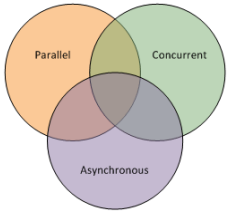

Embrace Parallelism exploiting the potential of multicore CPUs

Only one solution does not exist… combine different programming model

(righ tool for the job)

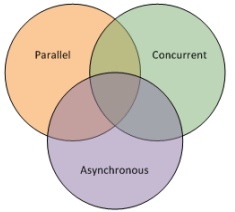
FP really **shines** in the area of concurrency

# Definition




- Concurrent
- Multithreaded
- Parallel
- Asynchronous

# Definition

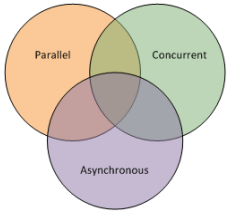



- Concurrent
- Multithreaded
- Parallel
- Asynchronous

Several things happening at once

TPL – Async – Agent - RX

# Definition



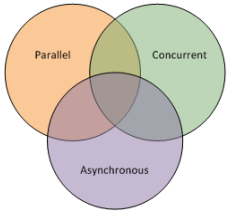

- Concurrent
- Multithreaded
- Parallel
- Asynchronous

Several things happening at once

TPL – Async – Agent - RX

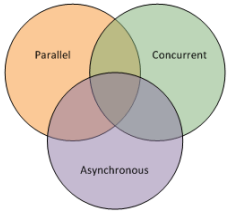
Multiple execution context

# Definition



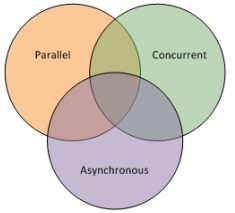

- Concurrent
  - Several things happening at once
  - TPL – Async – Agent - RX
- Multithreaded
  - Multiple execution context
- Parallel
  - Multiple simultaneous computations
  - TPL – Async – Agent – RX - GPU
- Asynchronous

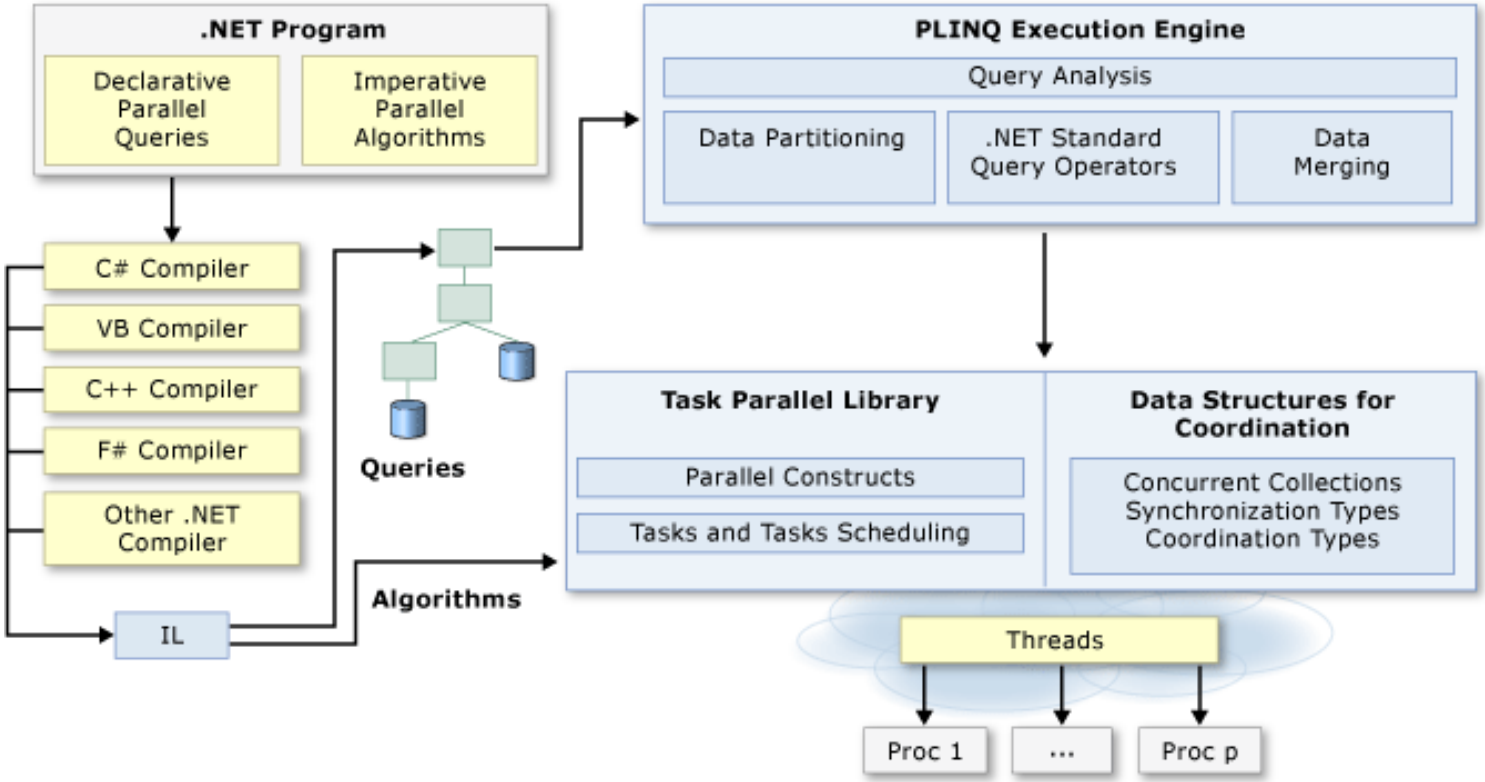
# Definition




| | |
|---|---|
| Concurrent | Several things happening at once<br>TPL – Async – Agent - RX |
| Multithreaded | Multiple execution context |
| Parallel | Multiple simultaneous computations<br>TPL – Async – Agent – RX - GPU |
| Asynchronous | Asynchronous<br>Async – Agent |

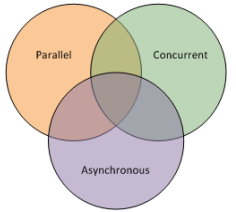
# Parallel Programming in the .NET



# Concurrent Programming Models

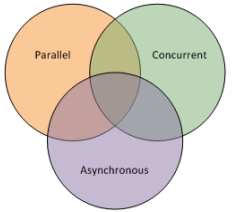
# Task Parallel Programming




- Efficient and scalable use of system resources..
  - When use correctly
- Programmatic fine grain control
- Is the model largely broad applicability
- Easy to integrate into existing programs

- Hard to solve complex problems
- Sometime require the introduction of locking primitives

# Sequential Fuzzy Match

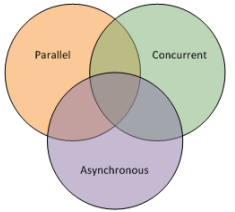



```
List<string> matches = new List<string>();
foreach (var word in WordsToSearch)
{
      var localMathes = JaroWinklerModule.bestMatch(Words, word);
      matches.AddRange(localMathes.Select(m => m.Word));
}
```

Fuzzy match 7 words against 13.4 Mb of text

Time execution in 4 Logical cores – 6 Gb Ram : **23,167** ms

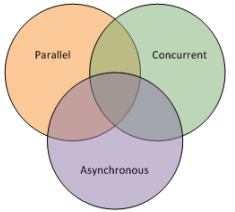
# Single Thread Fuzzy Match




```
List<string> matches = new List<string>();
var t = new Thread(() =>
{
foreach (var word in WordsToSearch) {
    var localMathes = FuzzyMatch.JaroWinklerModule.bestMatch(Words, word);
    matches.AddRange(localMathes.Select(m => m.Word));
} });

t.Start();
t.Join();
```

Time execution in 4 Logical cores – 6 Gb Ram : **22,613** ms

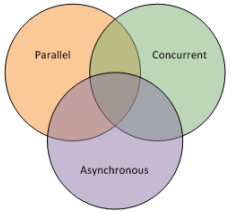
# Two Threads Fuzzy Match




```
List<string> matches = new List<string>();
    var t1 = new Thread(() =>{
        var take = WordsToSearch.Count / 2;

        foreach (var word in WordsToSearch.Take(take))  {
            var localMathes = FuzzyMatch.JaroWinklerModule.bestMatch(Words, word);
            matches.AddRange(localMathes.Select(m => m.Word));
        }
    });
    var t2 = new Thread(() =>{
        var start = WordsToSearch.Count / 2;
        var take = WordsToSearch.Count - start;

        foreach (var word in WordsToSearch.Skip(start).Take(take)) {
            var localMathes = FuzzyMatch.JaroWinklerModule.bestMatch(Words, word);
            matches.AddRange(localMathes.Select(m => m.Word));
        }
    });
    t1.Start();                    t2.Start();
    t1.Join();                     t2.Join();
```

Time execution : **15,436** ms

# Multi Threads Fuzzy Match



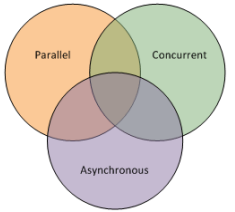

```
List<string> matches = new List<
var threads = new Thread[Environ

for (int i = 0; i < threads.Leng

      threads[index

      var take = Word
      var start =

      foreach (va
            var lo
            match

 for (int i = 0; i < threads.Length; i++)
    threads[i].Start();

for (int i = 0; i < threads.Length; i++)
   threads[i].Join();
```

```
eads.Length));
: index * take;

e)) {
le.bestMatch(Words, word);
lect(m => rd));
```

WRONG!!!

Time execution : **4,157** ms

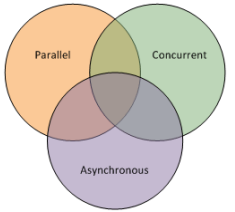
# Multi Threads Fuzzy Match




```
List<string> matches = new List<string>();
var threads = new Thread[Environment.ProcessorCount];

for (int i = 0; i < threads.Length; i++) {
                    var index = i;

      threads[index] = new Thread(() =>   {

      var take = WordsToSearch.Count / (Math.Min(WordsToSearch.Count, threads.Length));
      var start = index == threads.Length - 1 ? WordsToSearch.Count - take : index * take;


      foreach (var word in WordsToSearch.Skip(start).Take(take)) {
                var localMathes = FuzzyMatch.JaroWinklerModule.bestMatch(Words, word);
                lock (matches)
                      matches.AddRange(localMathes.Select(m => m.Word));
                    }
                });
              }

 for (int i = 0; i < threads.Length; i++)
    threads[i].Start();

for (int i = 0; i < threads.Length; i++)
   threads[i].Join();
```

Time execution : **6,831** ms

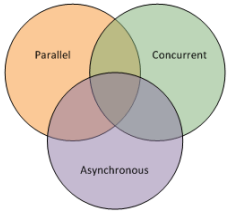
# Parallel Loop Fuzzy Match




```
List<string> matches = new List<string>();
 object sync = new object();

 Parallel.ForEach(WordsToSearch,
                  // thread local initializer
                  () => { return new List<string>(); },
                  (word, loopState, localMatches) =>  {
                         var localMathes = FuzzyMatch.JaroWinklerModule.bestMatch(Words,  wor
                         localMatches.AddRange(localMathes.Select(m  => m.Word));// same code
                         return localMatches;
                  },
     (finalResult) =>
     {
         // thread local aggregator
         lock (sync) matches.AddRange(finalResult);
     }
 );
```

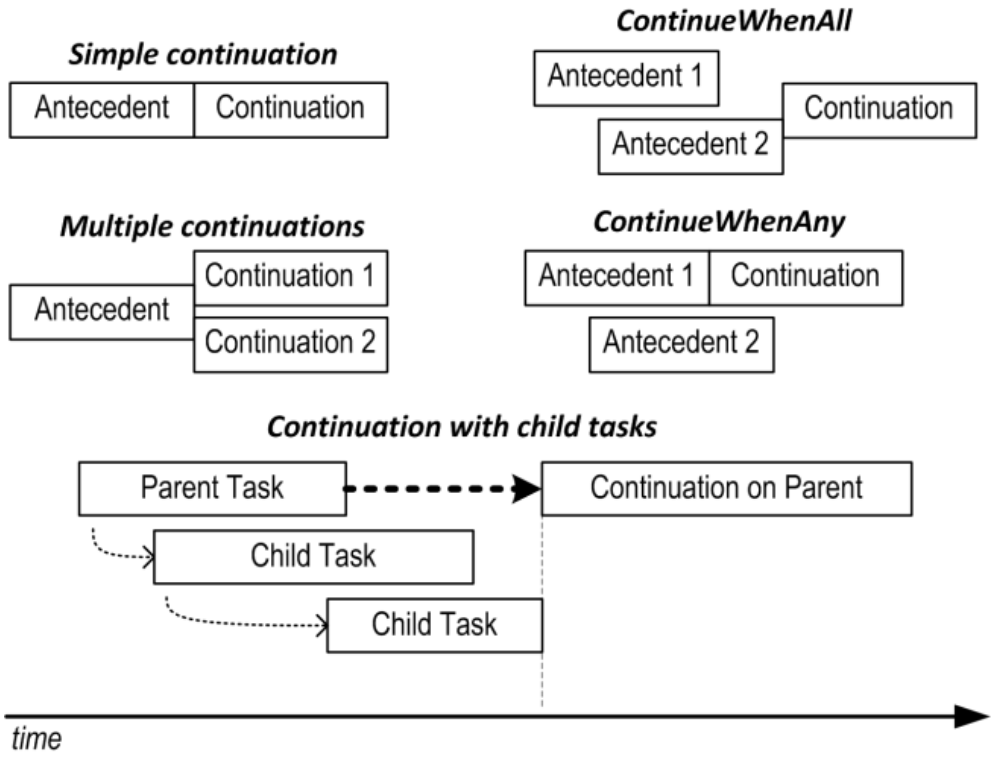
Time execution : **7,429** ms

# Multi Tasks Fuzzy Match



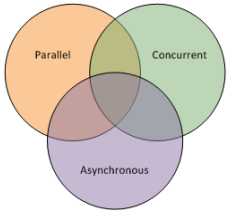

```
var tasks = new List<Task<List<string>>>();
var matches = new List<string>();
foreach (var word in WordsToSearch) {

        tasks.Add(Task.Factory.StartNew<List<string>>((w) => {
        List<string> matches = new List<string>();
        var localMathes = FuzzyMatch.JaroWinklerModule.bestMatch(Words, w);
        matches.AddRange(localMathes.Select(m => m.Word));
        return matches;
       }, word));
}

Task.Factory.ContinueWhenAll(tasks.ToArray(), (ts) => {
      matches = new List<string>(tasks.SelectMany(t => t.Result).Distinct());
}).Wait();
```

Time execution : **3,192** ms



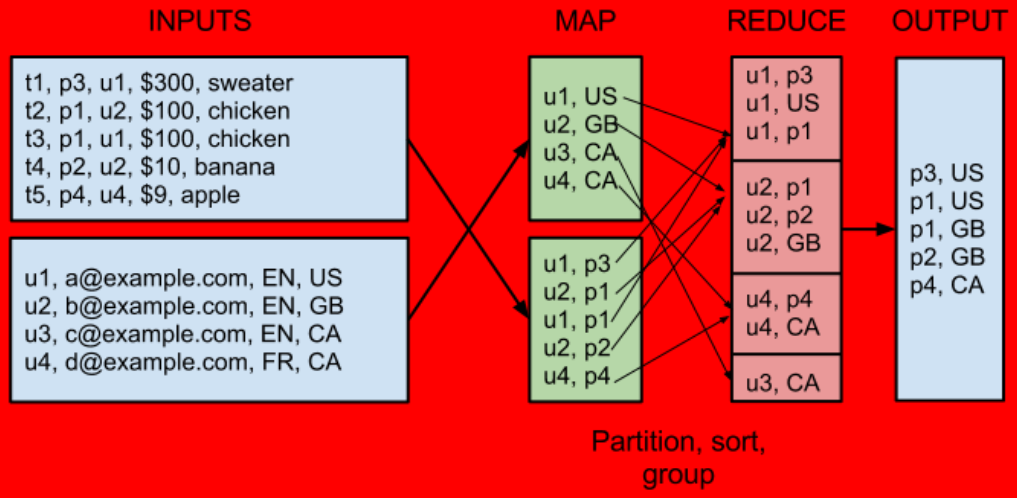

| TaskContinuationOptions |
| --- |
| AttachedToParent |
| ExecuteSynchronously |
| LazyCancellation |
| LongRunning |
| None |
| NotOnCanceled |
| NotOnFaulted |
| NotOnRanToCompletion |
| OnlyOnCanceled |
| OnlyOnFaulted |
| OnlyOnRanToCompletion |
| RunContinuationsAsynchronously |

# Task continuation

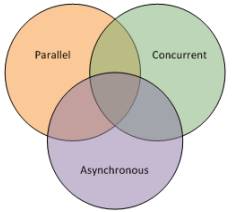



```
task1.ContinueWith(antecedent =>
        Console.WriteLine("Task #1 completion continuation."),
        TaskContinuationOptions.OnlyOnRanToCompletion);

task1.ContinueWith(antecedent =>
        Console.WriteLine("Task #1 cancellation continuation."),
        TaskContinuationOptions.OnlyOnCanceled);

task1.ContinueWith(antecedent =>
        Console.WriteLine("Task #1 on error continuation."),
        TaskContinuationOptions.OnlyOnFaulted);

task1.ContinueWith(antecedent =>
        Console.WriteLine("Task #1 continuation lomg running."),
        TaskContinuationOptions.LongRunning);
```

# Data Parallelism & Lambda Architecture `



# Parallel LINQ ( PLINQ )




```
var query =
    from i in Enumerable.Range(1, 10).AsParallel()
```

LINQ to Objects
LINQ to XML
**not LINQ to SQL,EF**

Extension method in System.LINQ

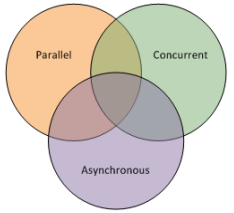
Extends **IEnumerable<T>**

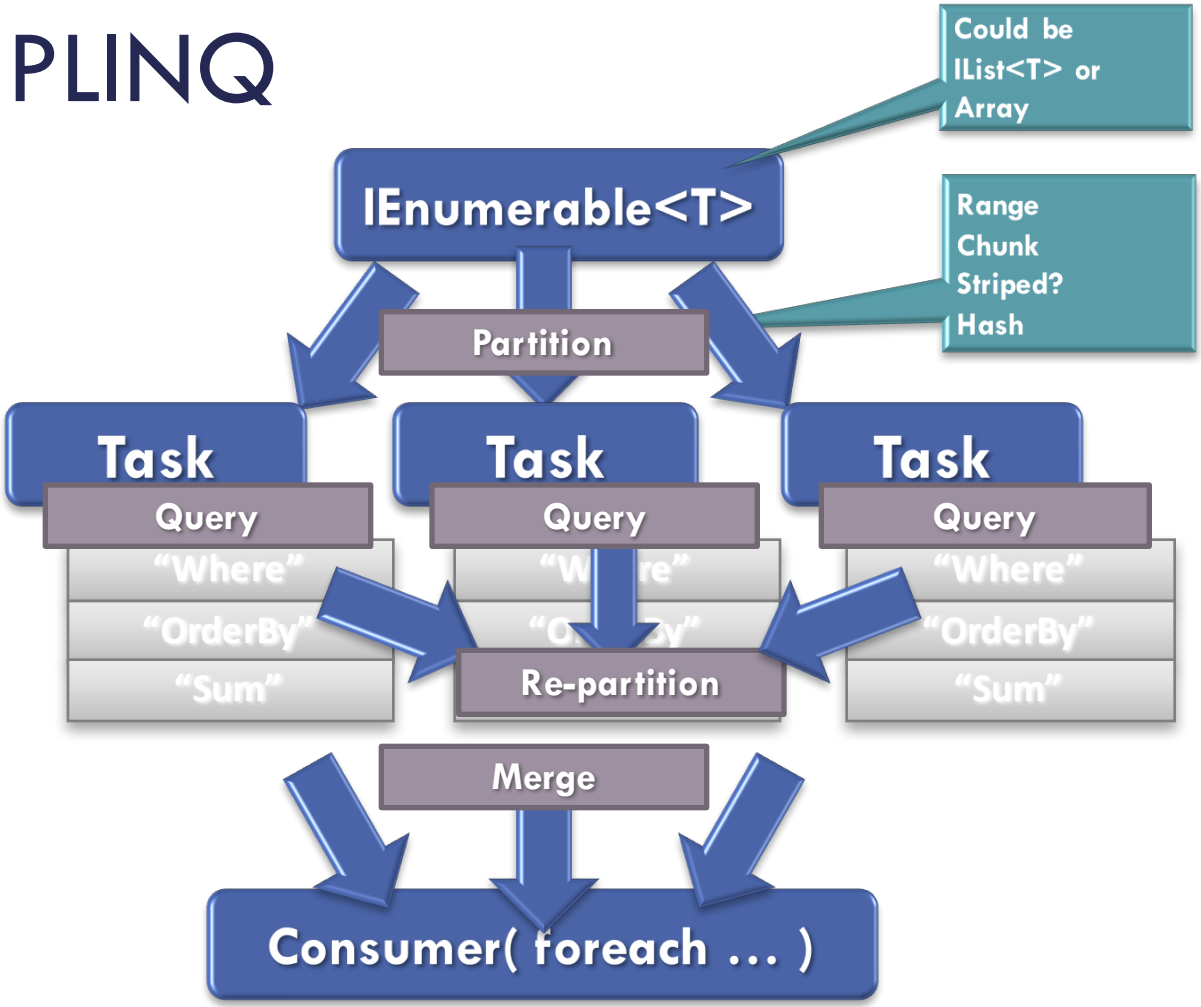
"Usually" would be **Enumerable**

Results in a **ParallelQuery<T>**

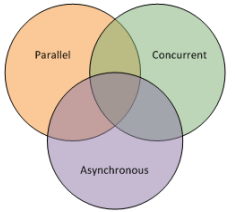
Select() | Where() | Etc.

# Parallel PLINQ



# PLINQ Fuzzy Match



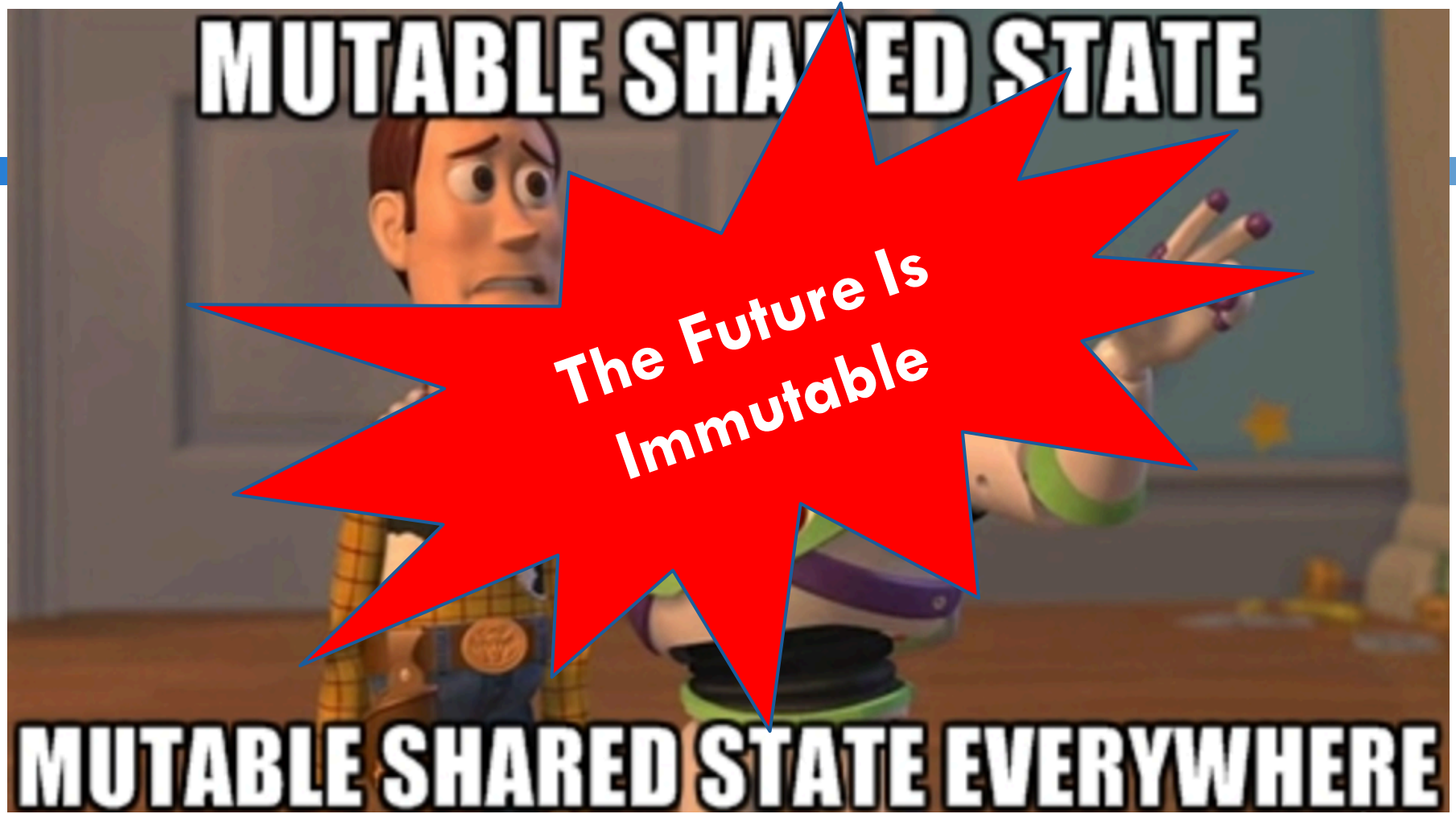

```
ParallelQuery<string> matches =
      (from word in WordsToSearch.AsParallel()
       from match in JaroWinklerModule.bestMatch(Words, word)
       select match.Word);
```

Time execution : **6,347** ms

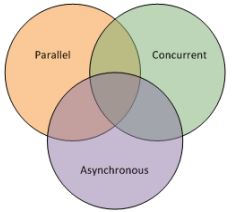
# Functional Programming in Concurrency

- declarative
- functions as values
- side-effects free

- referential transparency
- immutable
- composition

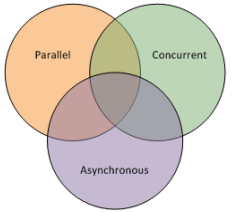

MUTABLE SHARED STATE
The Future Is Immutable
MUTABLE SHARED STATE EVERYWHERE

# Immutability



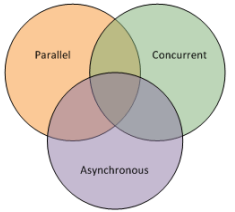

```
 public class ImmutablePerson
{
        public ImmutablePerson(string firstName, string lastName, int age)
        {
            FirstName = firstName;
            LastName = lastName;
            Age = age;
        }
        public string LastName { get; private set; }
        public string FirstName { get; private set; }
        public int Age { get; private set; }
}

type Person = {FirstName:string; LastName:string; Age:int}
```

# Side Effects

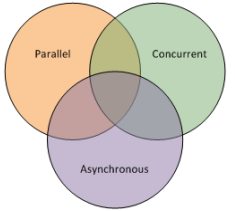

```
int WordCounterLinq(string dirPath) {
    var wordTable= (from file in Directory.EnumerateFiles(dirPath, "*.*")
                    from line in File.ReadAllLines(file)
                    from word in Regex.Split(line, @"\W+")
                    where !string.IsNullOrEmpty(word) && word.Length > 0
                    select word.ToUpper())
                .GroupBy(s => s).ToDictionary(k => k.Key, v => v.Count());
    return wordTable;
}
```

# Side Effects Free


```
int WordCounterLinq(string[] lines) {
    var wordTable= (    from line in lines
                        from word in Regex.Split(line, @"\W+")
                        where !string.IsNullOrEmpty(word) && word.Length > 0
                        select word.ToUpper())
                   .GroupBy(s => s).ToDictionary(k => k.Key, v => v.Count());
    return wordTable;
}
```

# Why FP



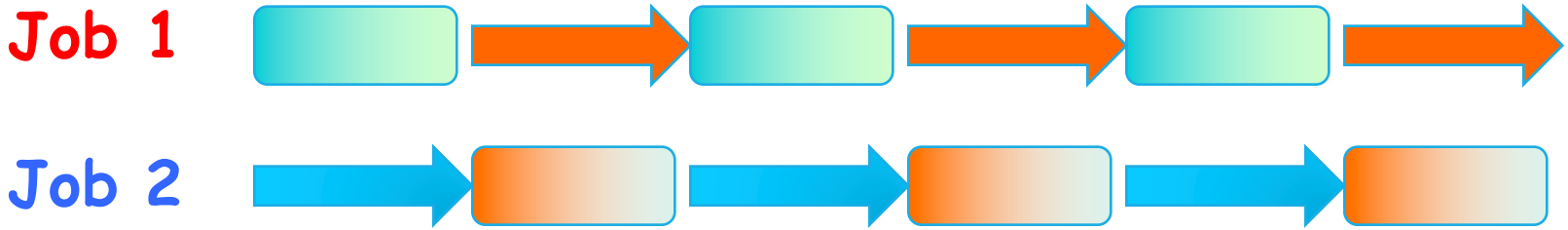
# Asynchronous Programming

# Asynchronous Programming

- *Software is often I/O-bound, it provides notable performance benefits*
  - Connecting to the Database
  - Leveraging web services
  - Working with data on disk
- *Network and disk speeds increasing slower*
- *Not Easy to predict when the operation will complete* (no-deterministic)
- ***IO bound functions can scale regardless of threads***
  - ***IO bound computations can often "overlap"***
  - ***This can even work for huge numbers of computations***

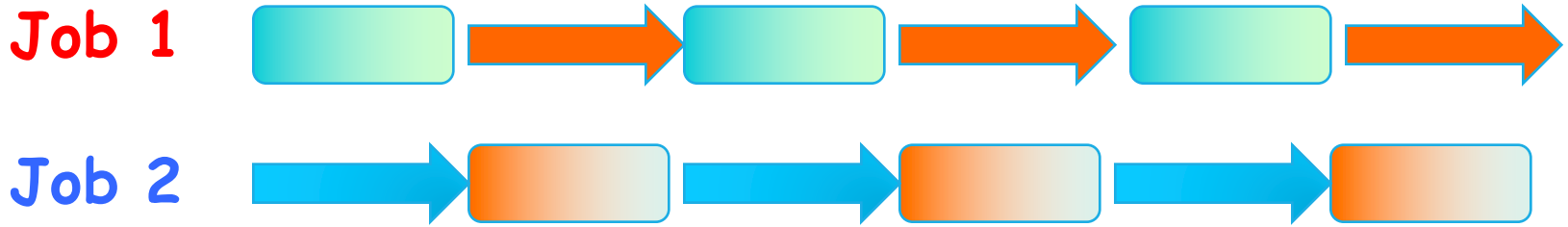
# Classic Synchronous programming

- We are used to writing code linearly

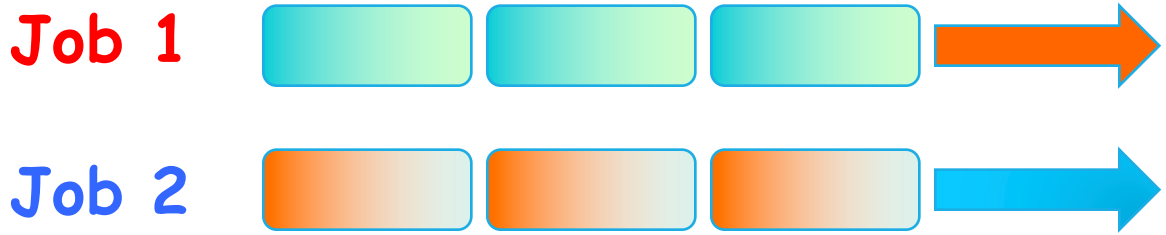


# Classic Synchronous programming

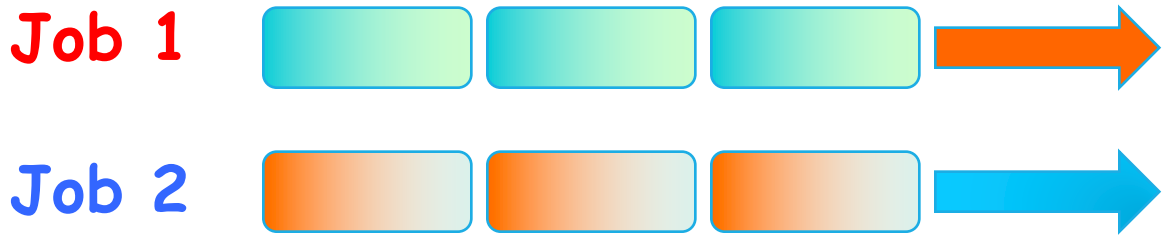
- We are used to writing code linearly




- Jobs executing in parallel

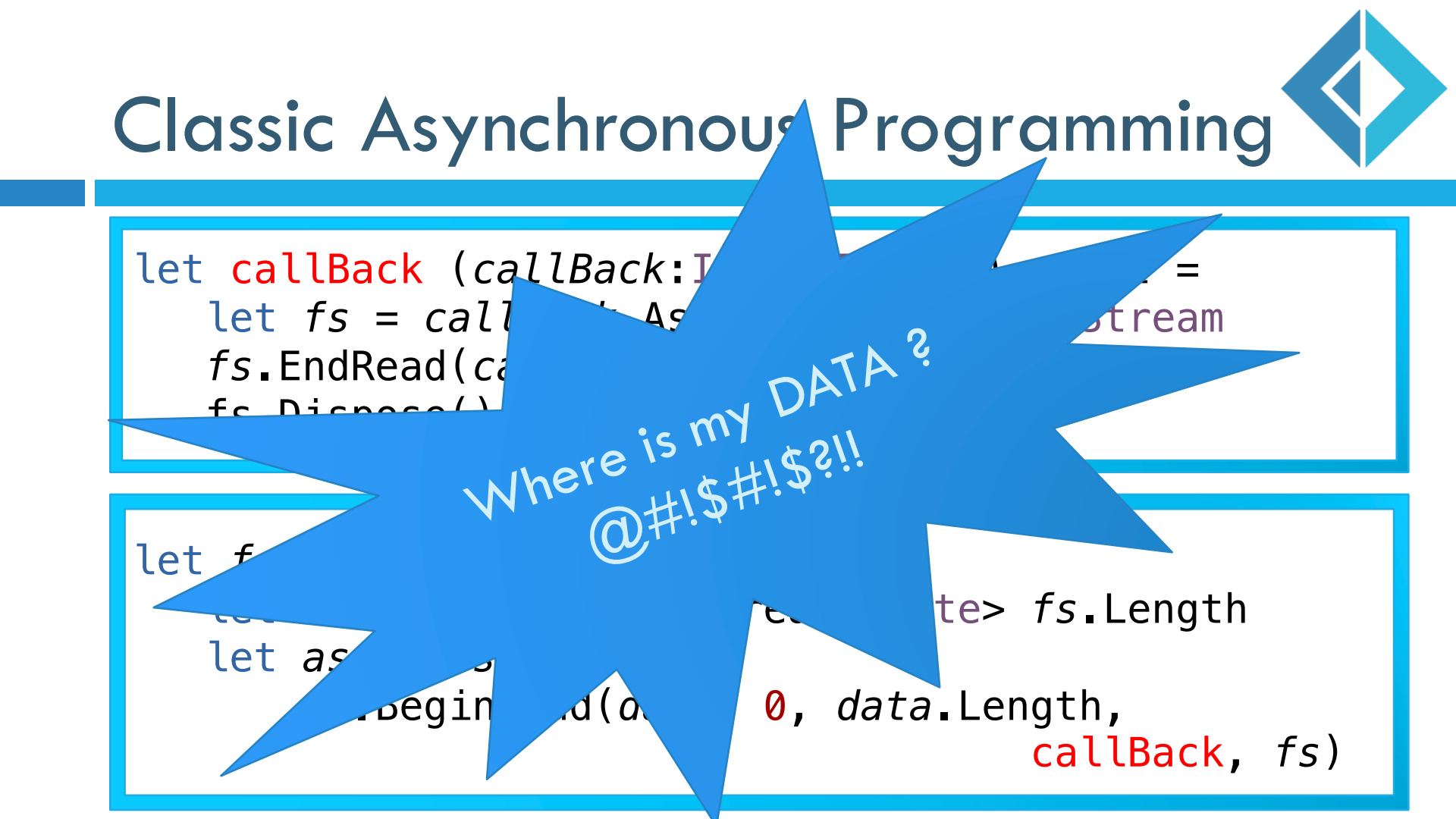


# Classic Asynchronous programming

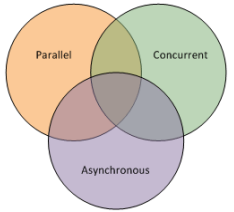



- Classic Asynchronous programming requires decoupling Begin from End
- Very difficult to:
  - Combine multiple asynchronous operations
  - Deal with exceptions, cancellation and transaction

# Classic Asynchronous Programming



# Asynchrouns Programming




```
static Task<string> GetTitleTplAsync(string url) {
            var w = new WebClient();
            Task<string> contentTask = w.DownloadStringTaskAsync(url);
            return contentTask.ContinueWith(t => {
                string result = ExtractTitle(t.Result);
                w.Dispose();
                return result; });
        }
static async Task<string> GetTitleCsAsync(string url) {
             using (var w = new WebClient()){
                 string content = await w.DownloadStringTaskAsync(url)
                 return ExtractTitle(content);
             }
         }
```

# Anatomy of Async Workflows

```
let readData path = async {
  let stream = File.OpenRead(path)
  let! data = stream.AsyncRead(stream.Length)
  return data }
```

- Easy transition from synchronous
  - Wrap in asynchronous workflow with the ***async*** keyword, use **let!** for async calls and add **return**
  - No need of explicit callback
  - Easy to debug
- Supports loops, recursion, exceptions, cancellation, resource management
- **Operation complete in the same scope**

# Anatomy of Async Workflows

```
let readData path : Async<byte[]> = async {
  let stream = File.OpenRead(path)
  let! data = stream.AsyncRead(stream.Length)
  return data }
```

- Async defines a block of code which execute on demand
- Easy to compose

# Async Composition

```
let readData path = async {  // string -> Async<byte[]>
      let stream = File.OpenRead(path)
      return! stream.AsyncRead(stream.Length) }
```

```
string -> byte[] -> Async<unit>
let writeData path data = async {
      let stream = File.OpenWrite(path)
      do! stream.AsyncWrite(data, 0, data.Length) }
```

```
// string -> string -> Async<unit>
let copyFile source destination = async {
      let! data = readData source
      do!  writeData destination data }
```

# Async Composition

```
let readData path = async {  // string -> Async<byte[]>
      let stream = File.OpenRead(path)
      return! stream.AsyncRead(stream.Length) }
```

```
string -> Async<byte[]> -> Async<unit>
let writeData path getData = async {
      let! data = getData
      let stream = File.OpenWrite(path)
      do! stream.AsyncWrite(data, 0, data.Length) }
```

```
// string -> string -> Async<unit>
let copyFile source destination : Async<unit> =
   readData source |> writeData destination // Async.Start
```

# Parallelizing Async Composition

- Parallel composition of workflows (Fork-Join)

```
let! docs = [ for url in urls -> downloadPage url ]
            |> Async.Parallel
```

- Task-based parallelism (Promise-Based)

```
async {
  let! dp1 = Async.StartChild(downloadPage(url1))
  let! dp2 = Async.StartChild(downloadPage(url2))
  let! page1 = dp1
  let! page2 = dp2 }
  return (page1, page2) }
```

# Exceptions & Parallel

Creates an asynchronous computation that executes all the given asynchronous computations queueing each as work items and using a fork/join pattern.

```
let rec fib x = if x <= 2 then 1 else fib(x-1) + fib(x-2)

let fibs =
  Async.Parallel[for i in 0..40 -> async { return fib(i) }]
  |> Async.Catch
  |> Async.RunSynchronously
  |> function
    | Choice1Of2 result > printfn "Successfully %A" result
    | Choice2Of2 exn -> printfn "Exception occured %s" exn.Message
```

# Asynchronous Workflows - Cancel

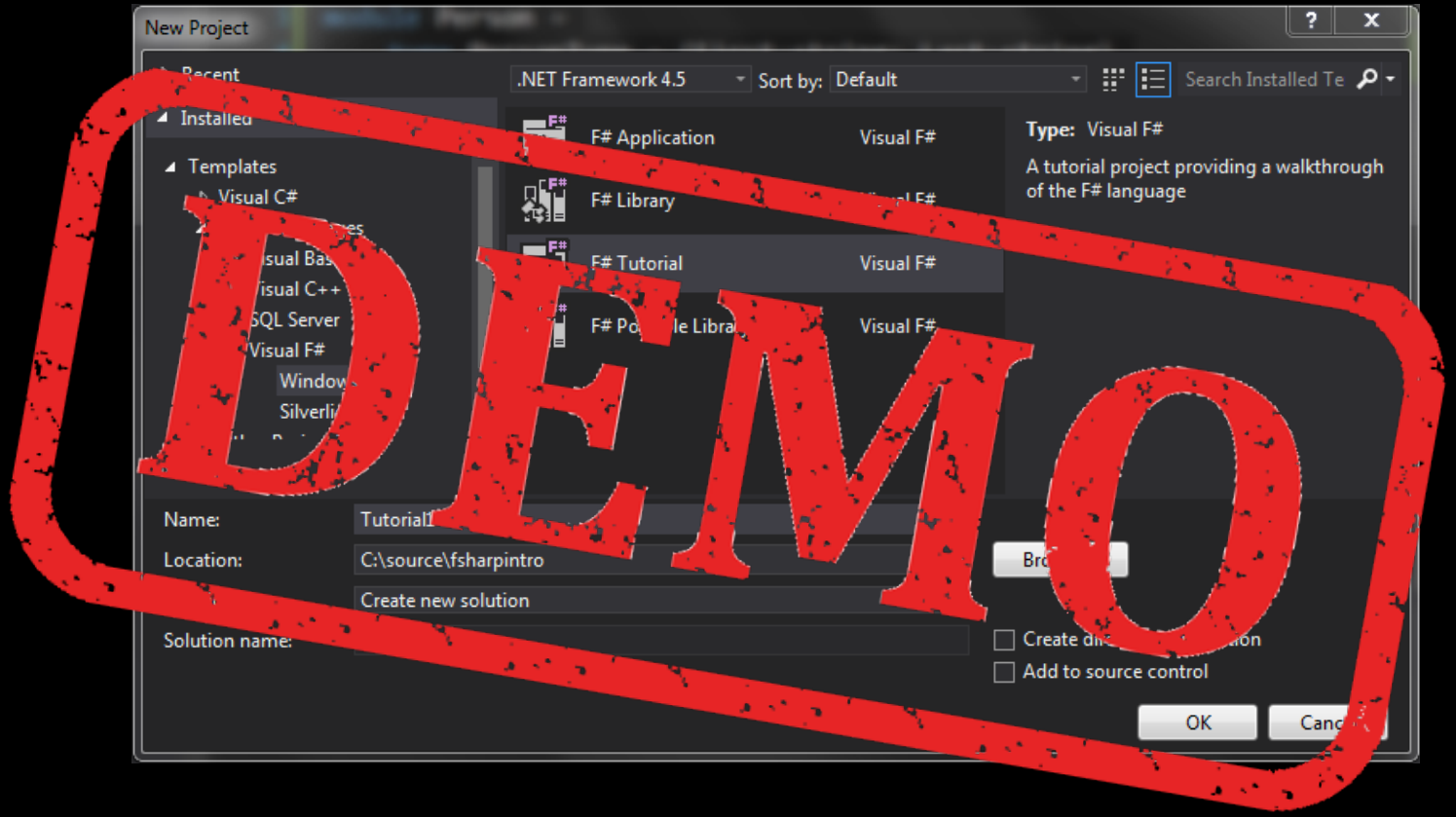
***The Asynchronous workflows can be cancelled… correctly!***

```
let token = new CancellationTokenSource()

    Async.Start(readFileAsynchronous, token.Token)

token.Cancel()
```

# Async – Limitations

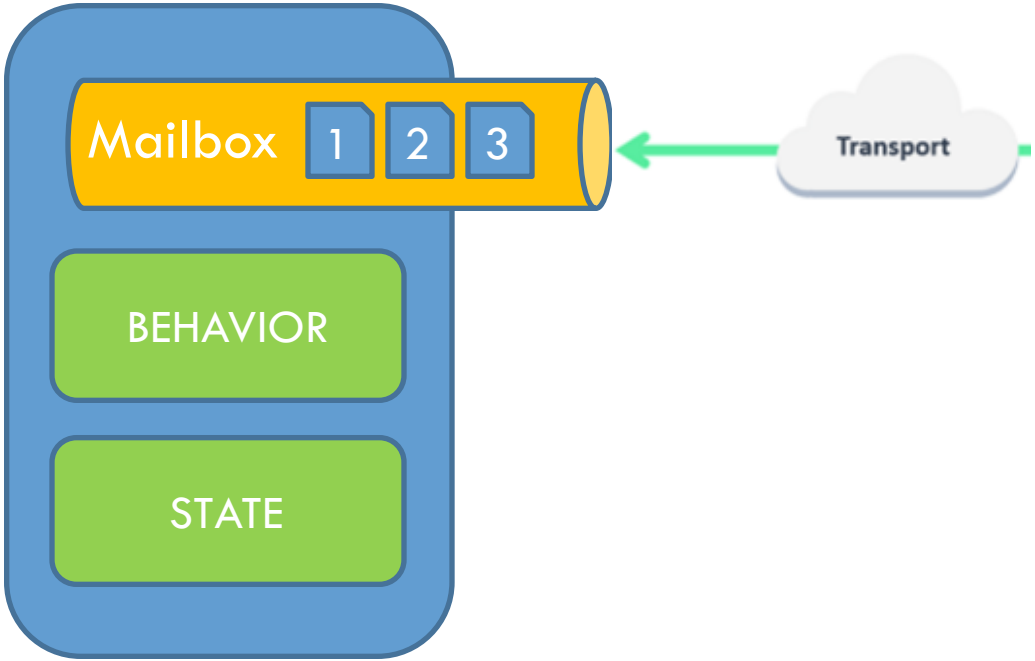
*Executing code in parallel there are numerous factors to take into account*

- The number of processor cores
- Processor cache coherency
- The existing CPU workload
- There is no throttling of executing threads to ensure an optimal usage
- For CPU-level parallelism, use the .NET Task Parallel Library


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

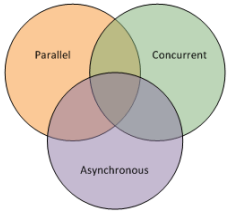
# Message Passing
# Agent (and Actor) model

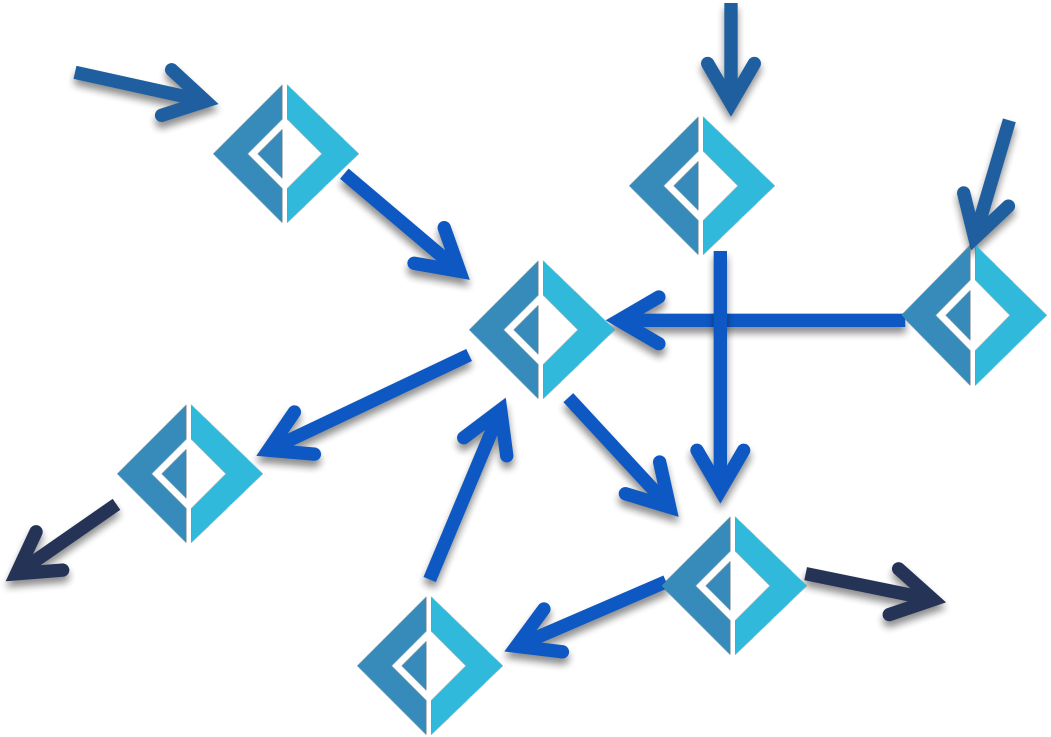
# Message Passing based concurrency




- Processing
- Storage – State
- Communication only by messages
- Share Nothing
- Message are passed by value
- Lightweight object.
- Running on it's own thread.
- No shared state.
- Messages are kept in mailbox and processed in order.
- Massive scalable and lightening fast because of the small call stack.

# Message Passing based concurrency



Agents **exchange** messages

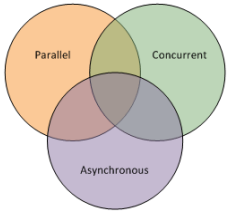
Receive message and react

**Reactive system**

Handle inputs while running

Emit results while running

# TPL DataFlow



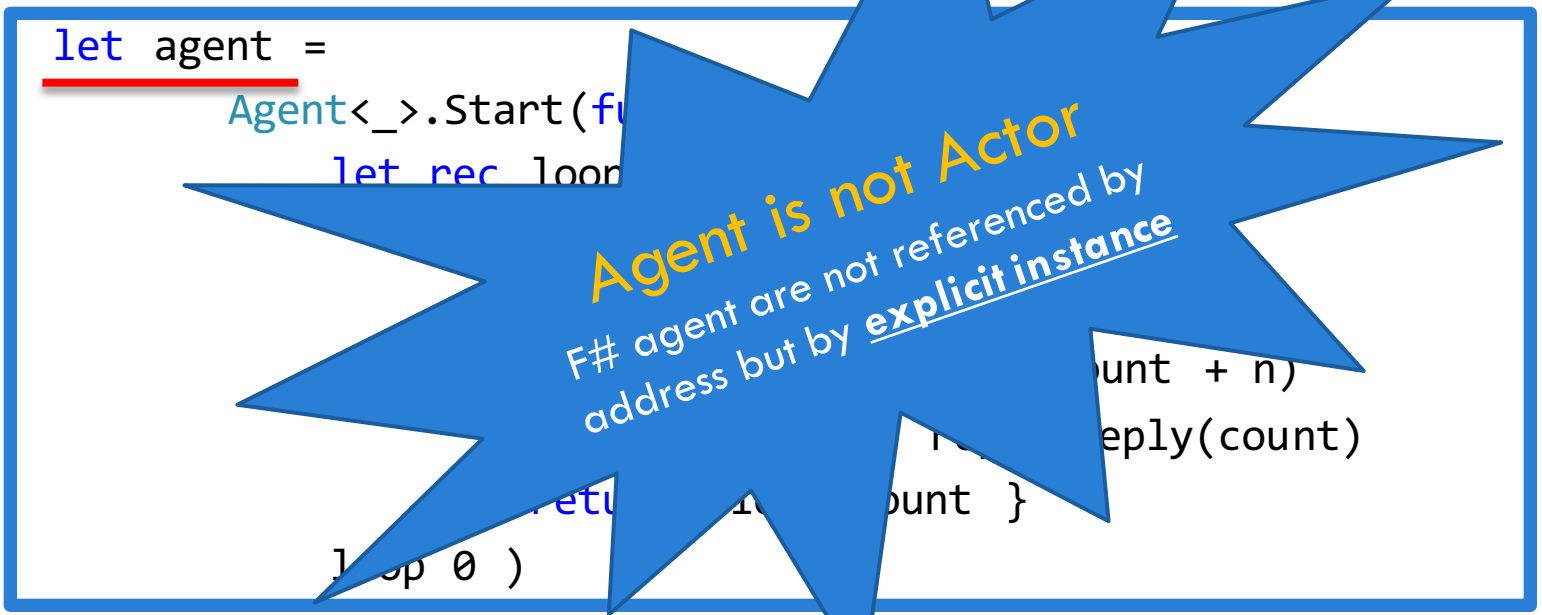

```
var InputBlock = new BufferBlock<Tuple<string, string>>();
var splitLines = new TransformBlock<Tuple<string, st
        n => /* Code ... Transfirmation */    );
var splitWords = new TransformBlock<Tuple<string, st
        n => /* Code ... Transfirmation */    );
var fuzzyMatch = new TransformBlock<Tuple<string, st
        n => /* Code ... Transfirmation */    );
var finalBlocl = new ActionBlock<Tuple<string, strin
        n => /* Code ...  */    );

InputBlock.LinkTo(splitLines, new DataflowLinkOptior
splitLines.LinkTo(splitWords, new DataflowLinkOptior
InputBlock.SendAsync(message);
```

| Platform | Dataflow support |
|---|---|
| .NET 4.5 | ✓ |
| .NET 4.0 | ✗ |
| Mono iOS/Droid | ✓ |
| Windows Store | ✓ |
| Windows Phone Apps 8.1 | ✓ |
| Windows Phone SL 8.0 | ✓ |
| Windows Phone SL 7.1 | ✗ |
| Silverlight 5 | ✗ |

# Immutability OR Isolation

```
let (lockfree:Agent<Msg<string,string>>) = Agent.Start(fun sendingInbox ->
    let cache = System.Collections.Generic.Dictionary<string, string>()
    let rec loop () = async {
        let! message = sendingInbox.Receive()
        match message with
            | Push (key,value) -> cache.[key] <- value
            | Pull (key,fetch) -> fetch.Reply cache.[key]
        return! loop ()
        }
    loop ())
```

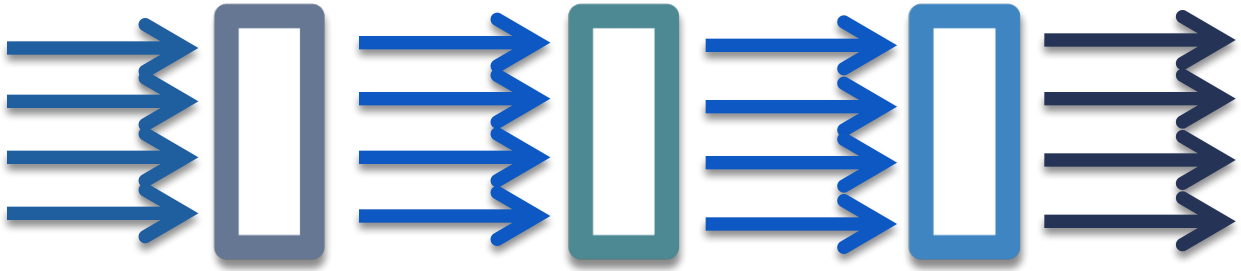
# Anatomy of an Agent



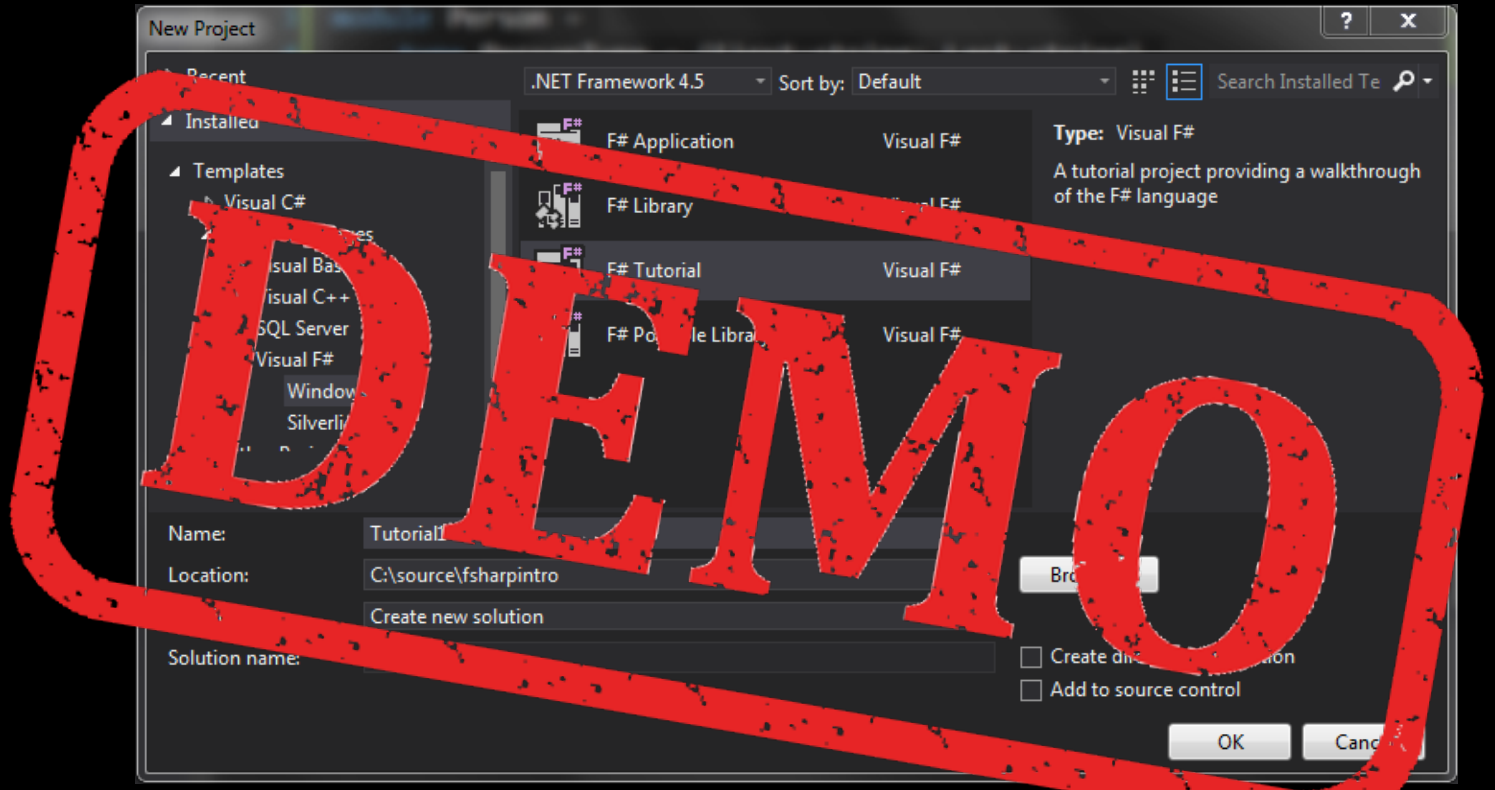

Agent is not Actor

F# agent are not referenced by address but by **explicit instance**

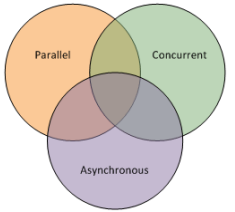
# Pipeline Processing



- Pipeline according to Wikipedia:
  - A pipeline is a set of data processing elements connected in series, so that the output of one element is the input of the next one. The elements of a pipeline are often executed in parallel or in time-sliced fashion; in that case, some amount of buffer storage is often inserted between elements


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

# Summary




Embrace Parallelism exploiting the potential of multicore CPUs

Only one solution does not exist… combine different programming model

(righ tool for the job)

FP really **shines** in the area of concurrency


That's all Folks!


*The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.*

*-- Edsger Dijkstra*

# How to reach me


github.com/rikace/Presentations/FRP-NUI

@TRikace tericcardo@gmail.com