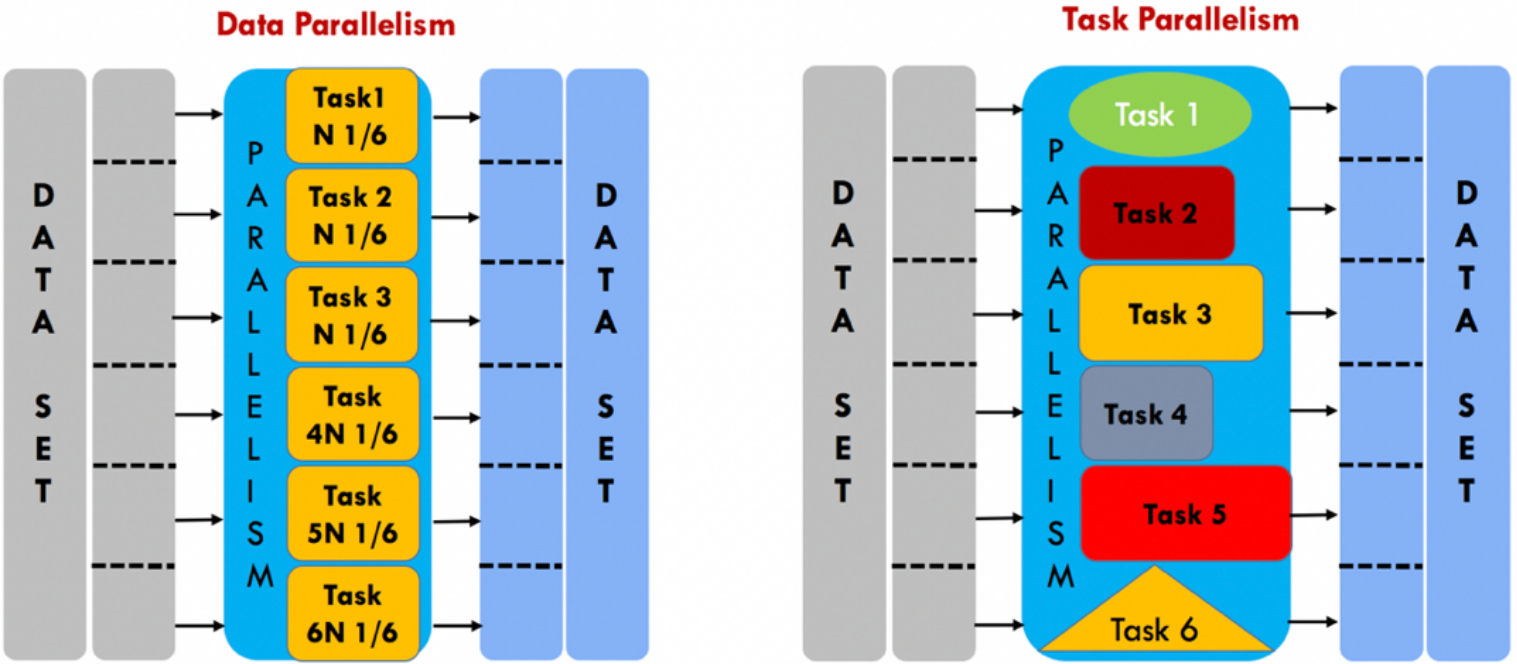
Parallelism with Task Parallel Library

TPL – PLINQ - PSeq

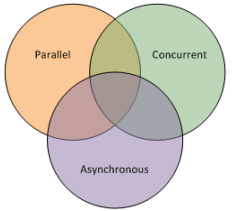
Objectives

- Increase speed of code without unwanted side effects
- Control the Immutability and Isolation are your best friends to write concurrent application

Task and Data Parallelism



Sequential Fuzzy Match

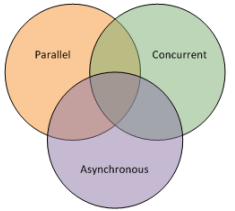



```
List<string> matches = new List<string>();
foreach (var word in WordsToSearch)
{
      var localMathes = JaroWinklerModule.bestMatch(Words, word);
      matches.AddRange(localMathes.Select(m => m.Word));
}
```

Fuzzy match 7 words against 13.4 Mb of text

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

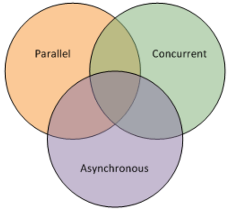
Two Threads Fuzzy Match




```
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 Thread Fuzzy M tch

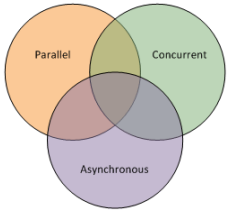



```
List<string> matches = new List<s          ();
var threads = new Thread[Environm

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


       threads[index]

       var take = Word                                    eads.Length));
       var start =                          .Cou           : index * take;


       foreach (var                          e)) {
              var lo                  oWink         le.bestMatch(Words, word);
              match                  ect(m =>        ord));




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

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

WRONG!!!

Time execution : **6,857** ms

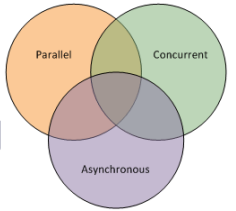
Multi Thread Fuzzy Match




```
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 : **7,731** ms

Task Parallel Library (TPL) - Task Programming

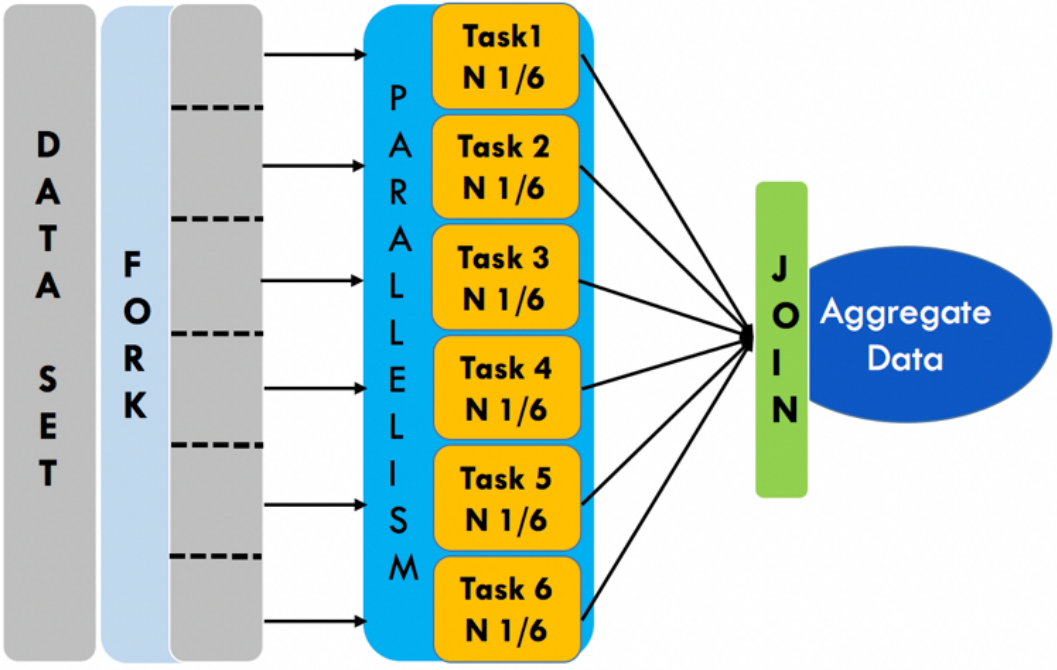


- Efficient and scalable use of system resources..
  - When used correctly
- Programmatic fine grain control
- Model largely applicable
- Easy to integrate into existing programs

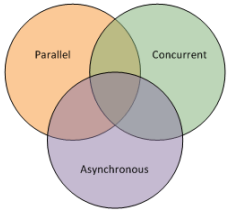
- Hard to solve complex problems
- Sometimes requires the introduction of locking primitives

Lab – Faster Fuzzy Match

The Fork/Join pattern



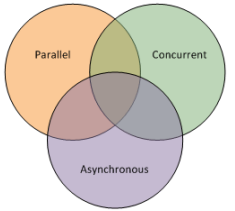
Parallel Loop Fuzzy Match




```
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, word);
                         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

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

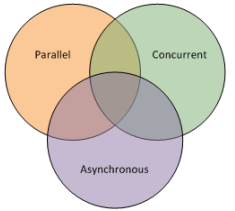
Extends **IEnumerable<T>**

"Usually" would be **Enumerable**

Results in a **ParallelQuery<T>**

Select() | Where() | Etc.

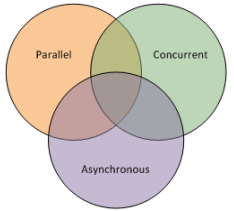
PLINQ Fuzzy Match

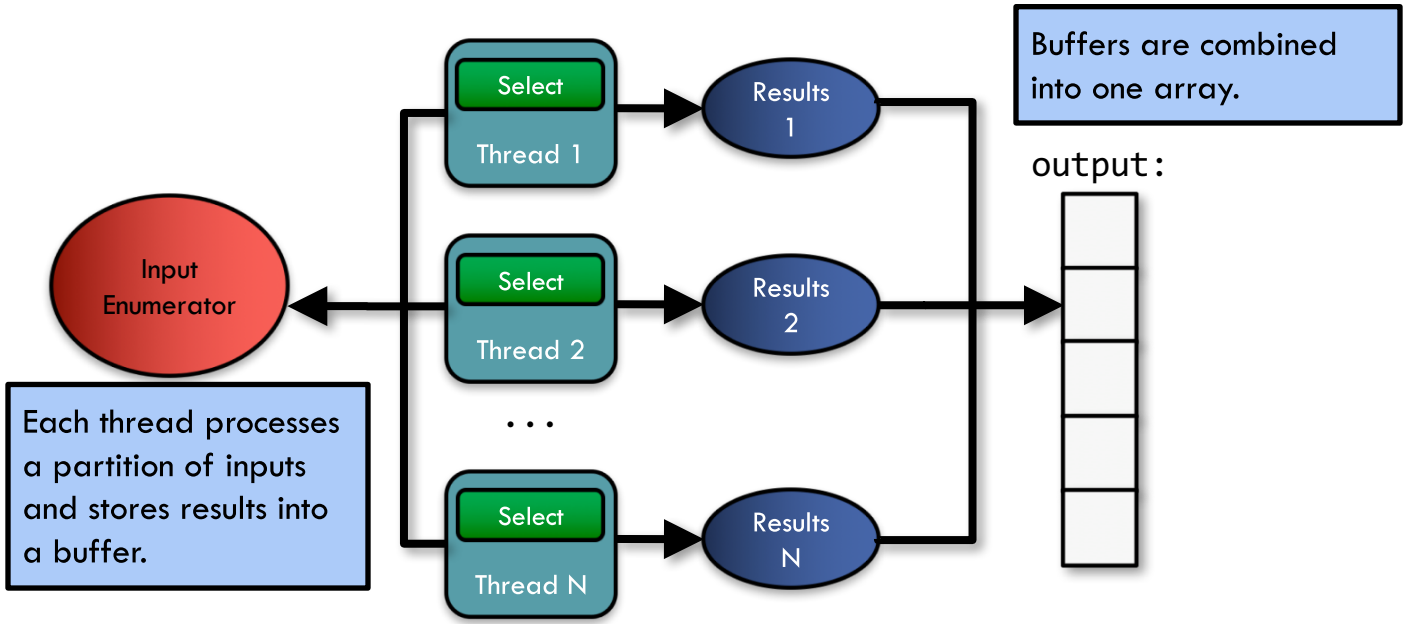



```
ParallelQuery<string> matches =
        (from word in WordsToSearch.AsParallel()
         from match in JaroWinklerModule.bestMatch(Words, word)
         select match.Word);
```

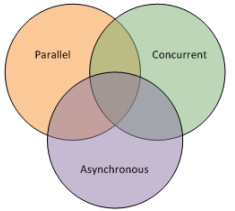
Time execution : **6,347** ms

PLINQ under the hood



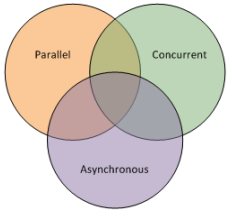


PLINQ options




```
var parallelQuery = from t in source.AsParallel()
    select t;

var cts = new CancellationTokenSource();
cts.CancelAfter(TimeSpan.FromSeconds(3));

    parallelQuery
        .WithDegreeOfParallelism(Environment.ProcessorCount)
        .WithExecutionMode(ParallelExecutionMode.ForceParallelism)
        .WithMergeOptions(ParallelMergeOptions.Default)
        .WithCancellation(cts.Token)
        .ForAll(Console.WriteLine);
```

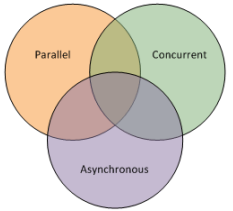
Multi Task Fuzzy Match




```
var tasks = new List<Task<List<string>>>();
var matches = new ThreadLocal<List<string>>(() => new List<string>());

foreach (var word in WordsToSearch) {

         tasks.Add(Task.Factory.StartNew<List<string>>((w) => {
         List<string> localMatches = matches.Value;
         var localMathes = FuzzyMatch.JaroWinklerModule.bestMatch(Words, w);
         localMatches.AddRange(localMathes.Select(m => m.Word));
         return localMatches;
        }, word));
}
Task.Factory.ContinueWhenAll(tasks.ToArray(), (ts) =>
      return new List<string>(tasks.SelectMany(t => t.Result).Distinct())
).Wait();
```

Time execution : **4,192** ms

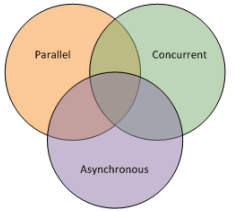
PLINQ Fuzzy Match



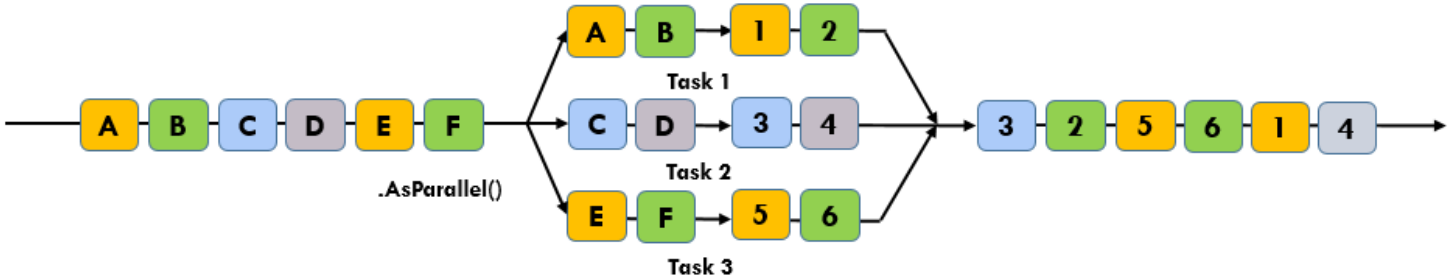

```
Create a load-balancing partitioner, specify false for static partitioning.


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

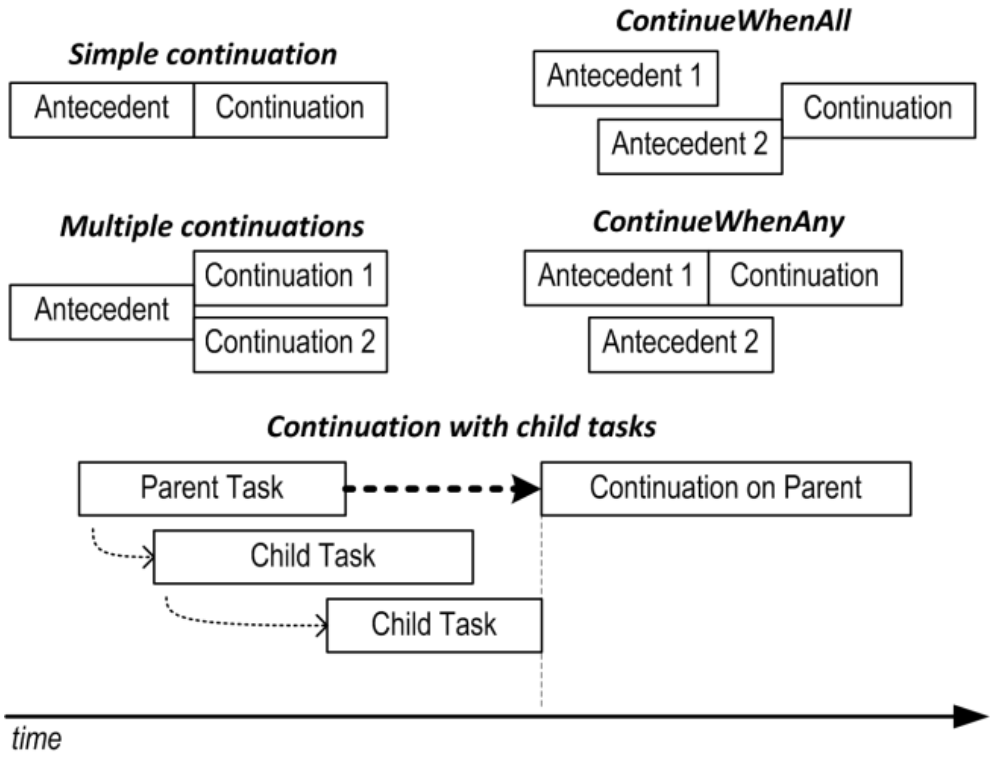
Time execution : **4,217** ms

PLINQ Fuzzy Match



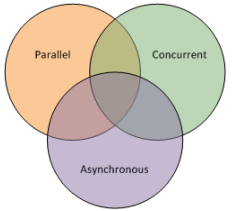

```
Create a load-balancing partitioner, specify false for static partitioning.

Partitioner<string> partitioner = Partitioner.Create(WordsToSearch, false);

ParallelQuery<string> matches =

       (from word in partitioner.AsParallel()

        from match in JaroWinklerModule.bestMatch(Words, word)

        select match.Word);
```





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);
```

* Tasks should evaluate side-effect free functions, which lead to referential transparency and deterministic code. Pure functions make the program more predictable because the functions always behave in the same way, regardless of the external state.

* Pure functions can run in parallel because the order of execution is irrelevant.

* If side effects are required, control them locally by performing the computation in a function with run-in isolation.

* Avoid sharing data between tasks by applying a defensive copy approach.

* Use immutable structures when data sharing between tasks cannot be avoided.

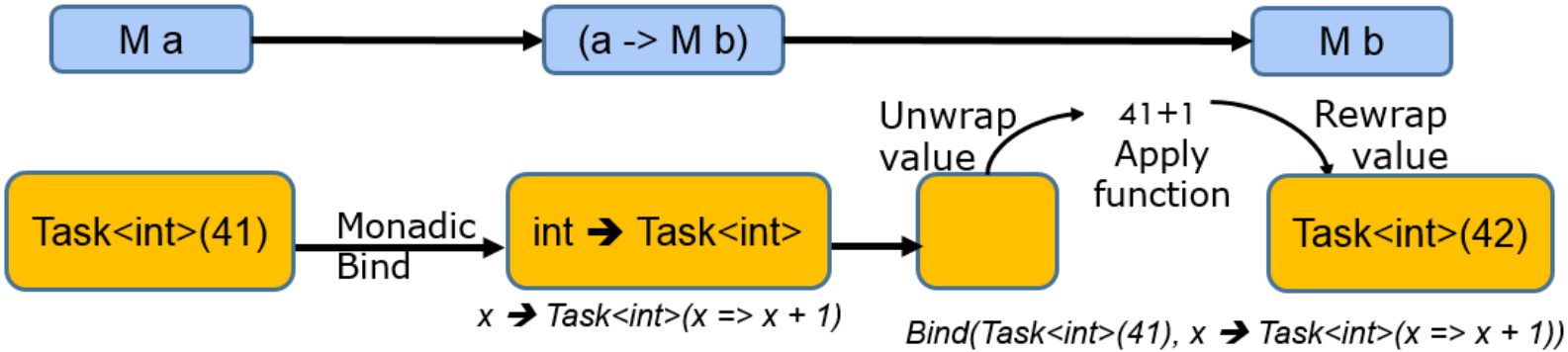
Mathematical pattern for better composition

Task is Monad

Composing Tasks

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

Task<T> Return(T value) …



Monad laws

Left identity: applying the Bind operation to a value wrapped by the Return operation and then passed into a function is the same as passing the value straight into the function:

Bind(Return value, function) = function(value)

Right identity: returning a bind-wrapped value is equal to the wrapped value directly:

Bind(elevated-value, Return) = elevated-value

Associative: passing a value into a function f whose result is passed into a second function g is the same as composing the two functions f and g and then passing the initial value:

Bind(elevated-value, f(Bind(g(elevated-value)) =
Bind(elevated-value, Bind(f.Compose(g), elevated-value))

Task Bind and Task Return

```
// 'T -> M<'T>
static member Return value : Task<'T> = Task.FromResult<'T> (value)

 // M<'T> * ('T -> M<'U>) -> M<'U>
static member Bind (input : Task<'T>, binder : 'T -> Task<'U>) =
      let tcs = new TaskCompletionSource<'U>()
      input.ContinueWith(fun (task:Task<'T>) ->
         if (task.IsFaulted) then
               tcs.SetException(task.Exception.InnerExceptions)
         elif (task.IsCanceled) then tcs.SetCanceled()
         else
              try
                 (binder(task.Result)).ContinueWith(fun
(nextTask:Task<'U>) -> tcs.SetResult(nextTask.Result)) |> ignore
              with
              | ex -> tcs.SetException(ex)) |> ignore
      tcs.Task
```

Task Bind and Task Return

```
 // M<'T> * ('T -> M<'U>) -> M<'U>
static member SelectMany(input : Task<'T>, binder : 'T -> Task<'U>) =
      let tcs = new TaskCompletionSource<'U>()
      input.ContinueWith(fun (task:Task<'T>) ->
         if (task.IsFaulted) then
               tcs.SetException(task.Exception.InnerExceptions)
         elif (task.IsCanceled) then tcs.SetCanceled()
         else
              try
                 (binder(task.Result)).ContinueWith(fun
(nextTask:Task<'U>) -> tcs.SetResult(nextTask.Result)) |> ignore
              with
              | ex -> tcs.SetException(ex)) |> ignore
      tcs.Task
```

Composing Tasks

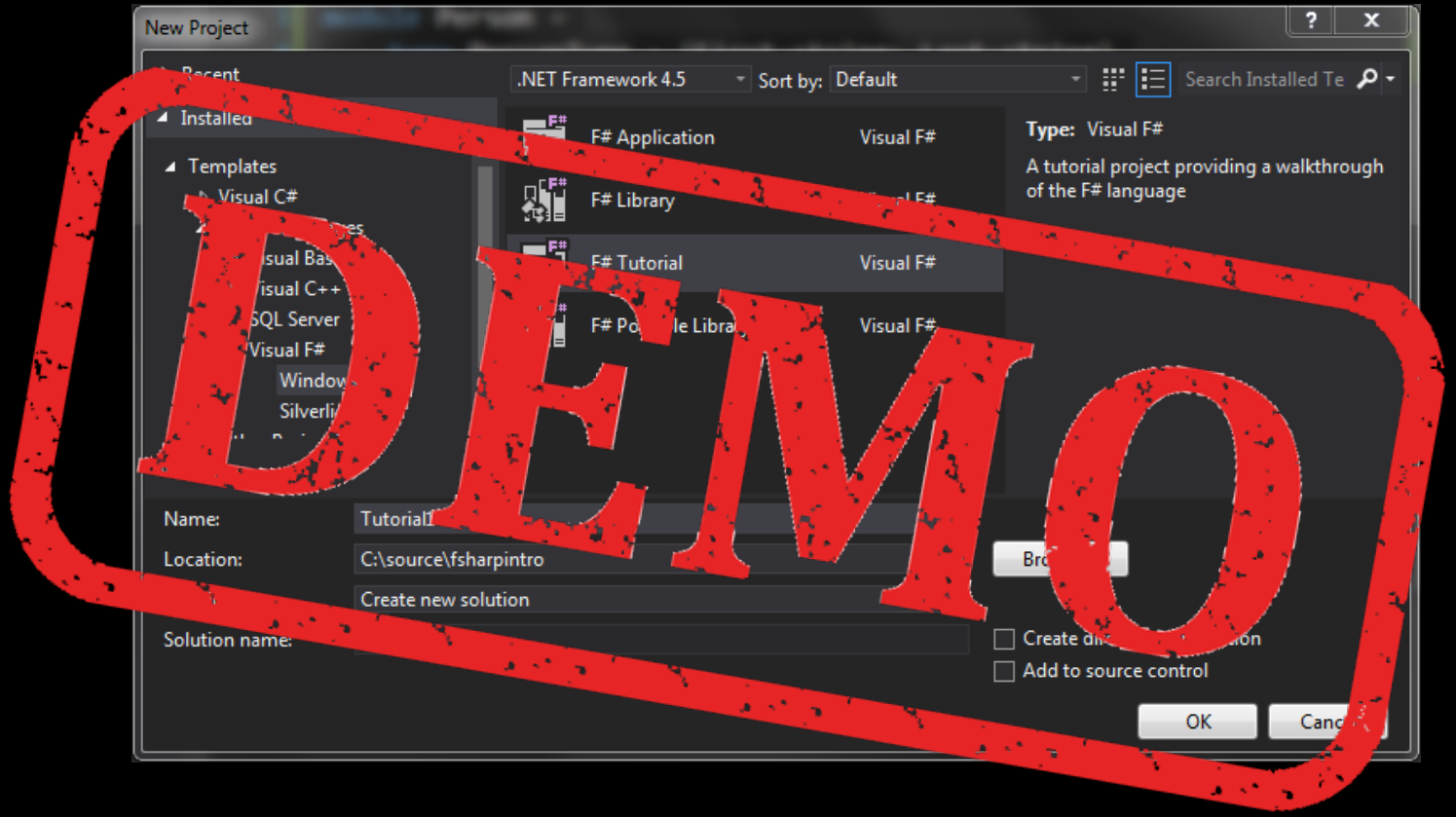
```
from image in Task.Run<Emgu.CV.Image<Bgr, byte>()
from imageFrame in Task.Run<Emgu.CV.Image<Gray, byte>>()
from faces in Task.Run<System.Drawing.Rectangle[]>()
select faces;
```

The hidden Functor – Map

Map : (T -> R) -> [T] -> [R]

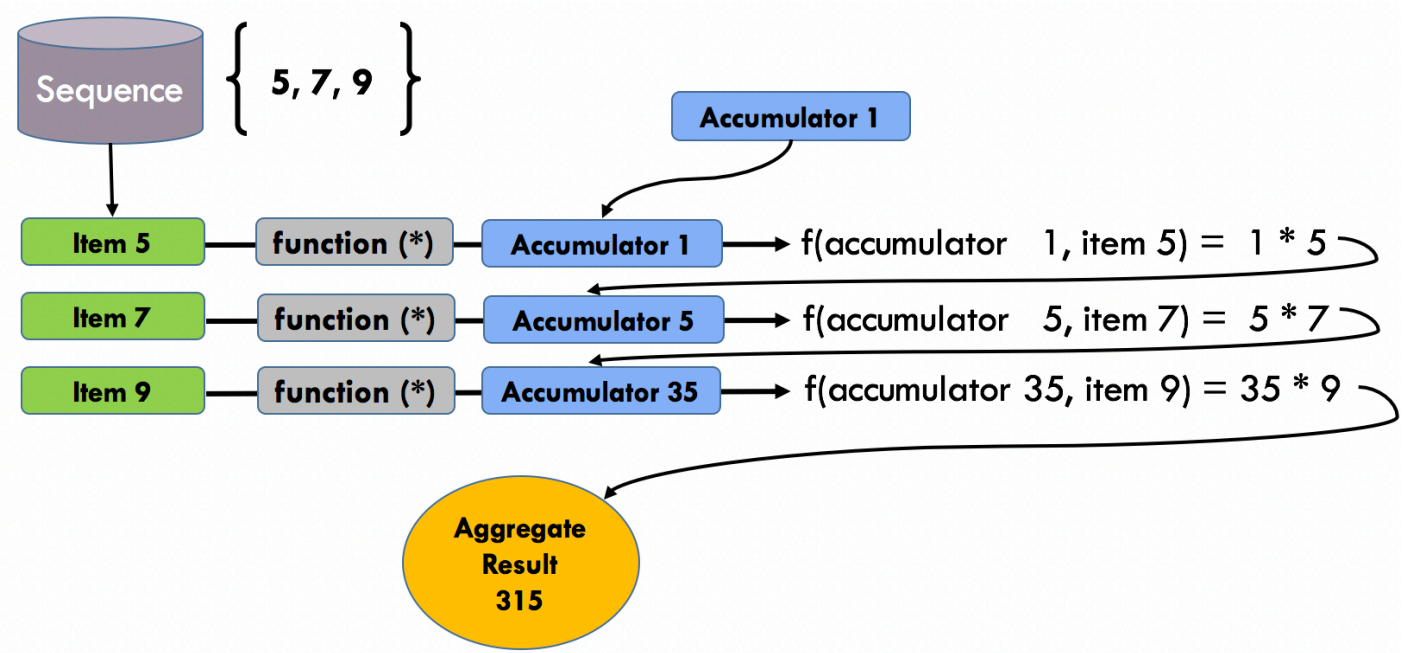
Select(this IEnumerable<T> task, Func<T , R> map) …

Select(this Task<T> task, Func<T , R> map) …


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

Parallel Reducer and Map-Reduce

Aggregating and reducing



Fold function

```
let map (projection:'a -> 'b) (sequence:seq<'a>) =
    sequence |> Seq.fold(fun acc item -> (projection item)::acc) []

let max (sequence:seq<int>) =
    sequence |> Seq.fold(fun acc item -> max item acc) 0

let filter (predicate:'a -> bool) (sequence:seq<'a>) =
    sequence |> Seq.fold(fun acc item ->
            if predicate item = true then item::acc else acc) []

let length (sequence:seq<'a>) =
    sequence |> Seq.fold(fun acc item -> acc + 1) 0
```

Aggregate function

```
IEnumerable<T> Map<T, R>(IEnumerable<T> sequence, Func<T, R> projection){
    return sequence.Aggregate(new List<R>(), (acc, item) => {
                acc.Add(projection(item));
                return acc;
    });
}

int Max(IEnumerable<int> sequence) =>  sequence.Aggregate(0, (acc, item) => Math.Max(item, acc));

IEnumerable<T> Filter<T>(IEnumerable<T> sequence, Func<T, bool> predicate){
     return sequence.Aggregate(new List<T>(), (acc, item) => {  // C
          if (predicate(item))
              acc.Add(item);
              return acc;
     });
}
```

Lab – implement a parallel reducer

Parallel reducer

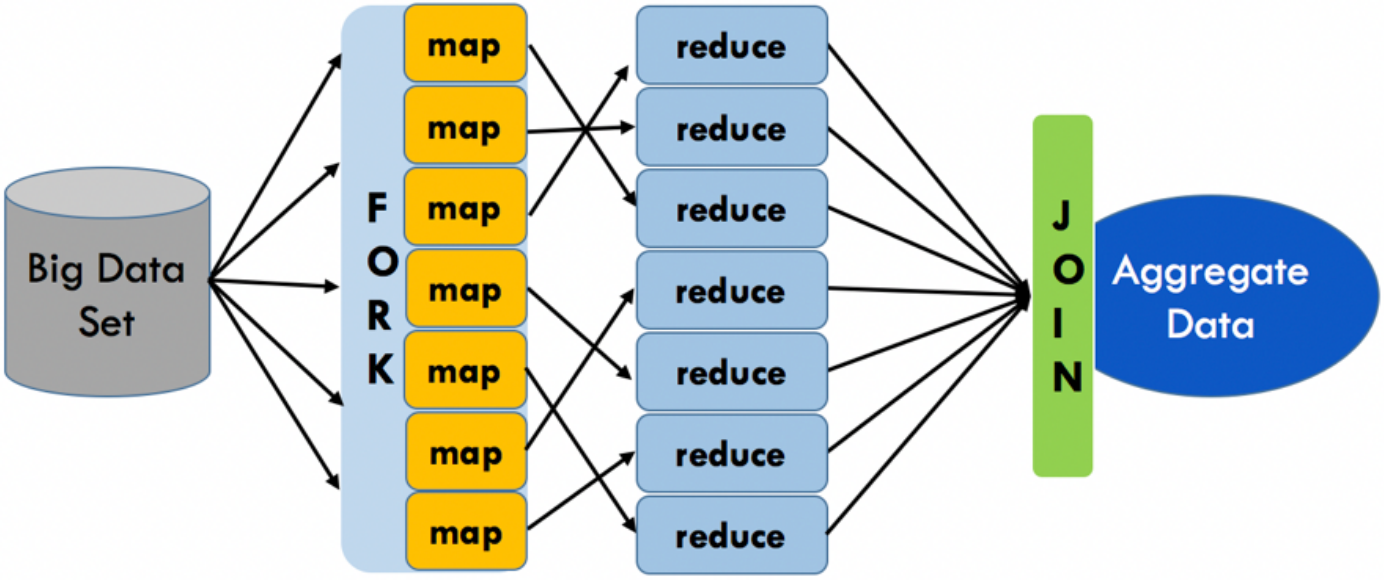
```
double Reduce(double[] arr, Func<double, double, double> reducer, double init)
{
            double accum = init;
           for (int i = 0; i < arr.Length; ++i)
                 accum = reducer(arr[i], accum);
           return accum;
}

double[] Map(double[] src, Func<double, double> fnapply)
{
         double[] ret = new double[src.Length];
         Parallel.For(0, ret.Length, new ParallelOptions {
                          MaxDegreeOfParallelism = Environment.ProcessorCount }
         , (i) => { ret[i] = fnapply(src[i]); }
          );
         return ret;
}
```

Parallel reducer

```
TSource Reduce<TSource>(this ParallelQuery<TSource>
source,  Func<TSource, TSource, TSource> func)
{
    return ParallelEnumerable.Aggregate(source,
        (item1, item2) => func(item1, item2));
}


int[] source = Enumerable.Range(0, 100000).ToArray();
int result = source.AsParallel()
        .Reduce((value1, value2) => value1 + value2);
```

Lab – Map Reduce

Map Reduce


Big Data Set
FORK
map
map
map
map
map
map
reduce
reduce
reduce
reduce
reduce
reduce
JOIN
Aggregate Data

Functional Design Patterns – Monoid

- Monoid is a binary operator with identity element
  - Ex: (+, 0) (*, 1), (List.concat, [])
- Monid law: Associativity
  - Ex: 1 + (2 + 3) = (1 + 2) + 3

Map >> Reduce

```
Directory.GetFiles(path, "*.*", SearchOption.AllDirectories)
    .Select(f =>
        {   using (var fs = new FileStream(f, FileMode.Open, FileAccess.Read))
                return new
                {
                    FileName = f,
                    FileHash = BitConverter.ToString(SHA1.Create().ComputeHash(fs))
                };
        })
    .Where(f => Path.GetFileExtesion(f) == “.txt”)
    .GroupBy(f => f.FileHash, EqualityComparer<string>.Default)
    .Select(g => new { FileHash = g.Key, Files = g.Select(z => z.FileName)  })
    .Where(g => g.Files.Count > 1)
    //.Skip(1).SelectMany(f => f.Files)
    .ToList();
```

Map >> Reduce

```
Directory.GetFiles(path, "*.*", SearchOption.AllDirectories).AsParallel()
    .Select(f =>
        {   using (var fs = new FileStream(f, FileMode.Open, FileAccess.Read))
                return new
                {
                    FileName = f,
                    FileHash = BitConverter.ToString(SHA1.Create().ComputeHash(fs))
                };
        })
    .Where(f => Path.GetFileExtesion(f) == ".txt")
    .GroupBy(f => f.FileHash, EqualityComparer<string>.Default)
    .Select(g => new { FileHash = g.Key, Files = g.Select(z => z.FileName)  })
    .Where(g => g.Files.Count > 1)
    //.Skip(1).SelectMany(f => f.Files)
    .ToList();
```

Map-Reduce in F#

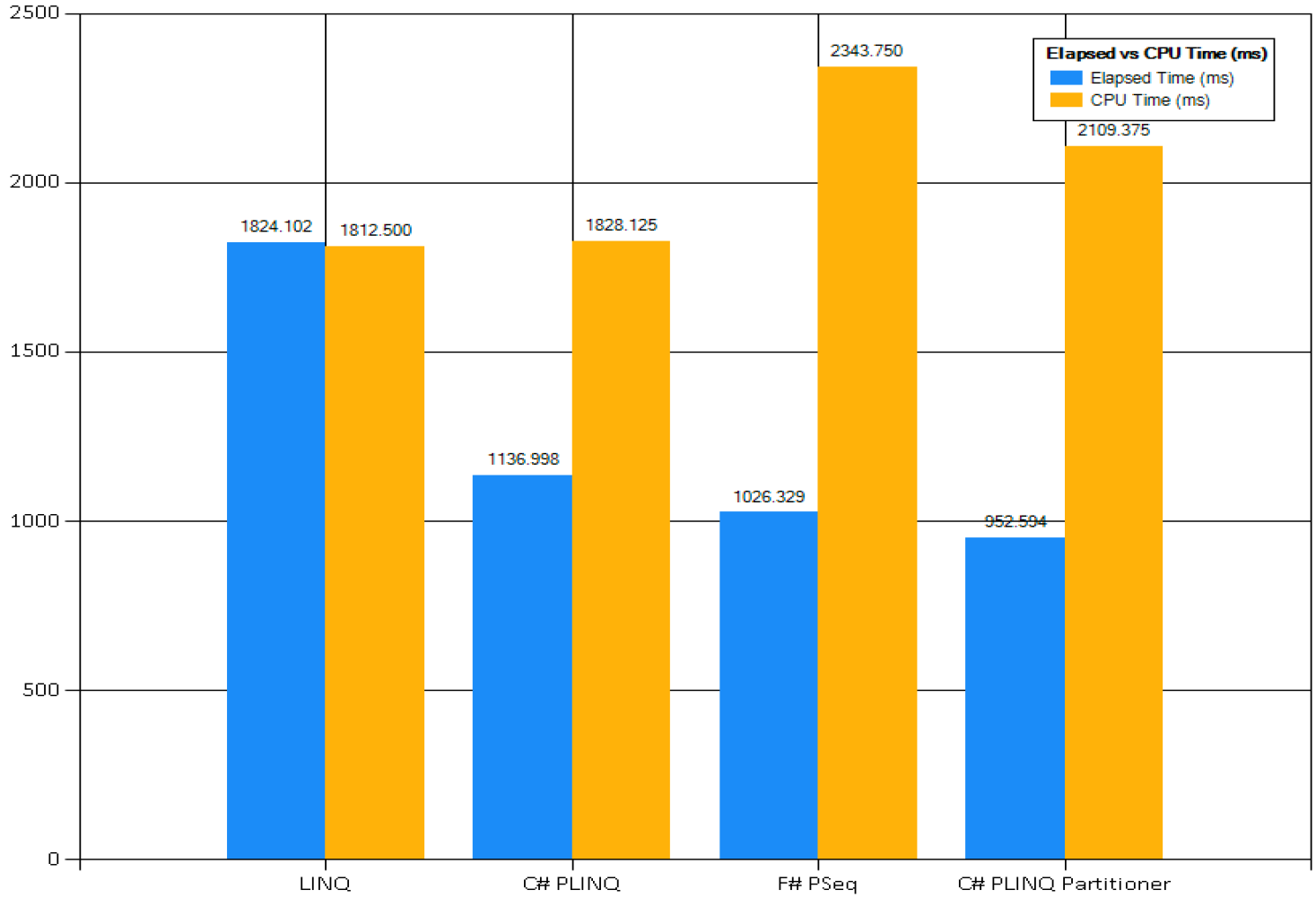
```
let mapF  M (map:'in_value -> seq<'out_key * 'out_value>)  (inputs:seq<'in_value>) =
    inputs
    |> PSeq.withExecutionMode ParallelExecutionMode.ForceParallelism
    |> PSeq.withDegreeOfParallelism M
    |> PSeq.collect (map)
    |> PSeq.groupBy (fst)
    |> PSeq.toList

let reduceF R (reduce:'key -> seq<'value> -> 'reducedValues)  (inputs:('key * seq<'key * 'value>) seq) =
    inputs
    |> PSeq.withExecutionMode ParallelExecutionMode.ForceParallelism
    |> PSeq.withDegreeOfParallelism R
    |> PSeq.map (fun (key, items) ->
       items
       |> Seq.map (snd)
       |> reduce key)
    |> PSeq.toList
```

Map-Reduce in F#

```
let mapReduce
        (inputs:seq<'in_value>)
        (map:'in_value -> seq<'out_key * 'out_value>)
        (reduce:'out_key -> seq<'out_value> ->
'reducedValues)
        M R =

    inputs |> (mapF M map >> reduceF R reduce) //#A
```

Map-Reduce in action

```
let executeMapReduce (ranks:(string*float)seq) =
    let M,R = 10, 5
    let data = Data.loadPackages()
    let pg = MapReduce.Task.PageRank(ranks)

    mapReduce data (pg.Map) (pg.Reduce) M R
```


Elapsed vs CPU Time (ms)
Elapsed Time (ms)
CPU Time (ms)
2500
2000
1500
1000
500
0
1824.102
1812.500
1136.998
1828.125
1026.329
2343.750
952.594
2109.375
LINQ
C# PLINQ
F# PSeq
C# PLINQ Partitioner

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

-- Edsger Dijkstra