Parallel Task Programming

Techniques & Best Practices

BY RICCARDO TERRELL - @RIKACE

Objectives

- Adapt the degree of parallelism based on the resources available (Throttle)
- Increase the speed of code without unwanted side effects
- Immutability and Isolation are your best friends to write concurrent applications

Threadpool

User:

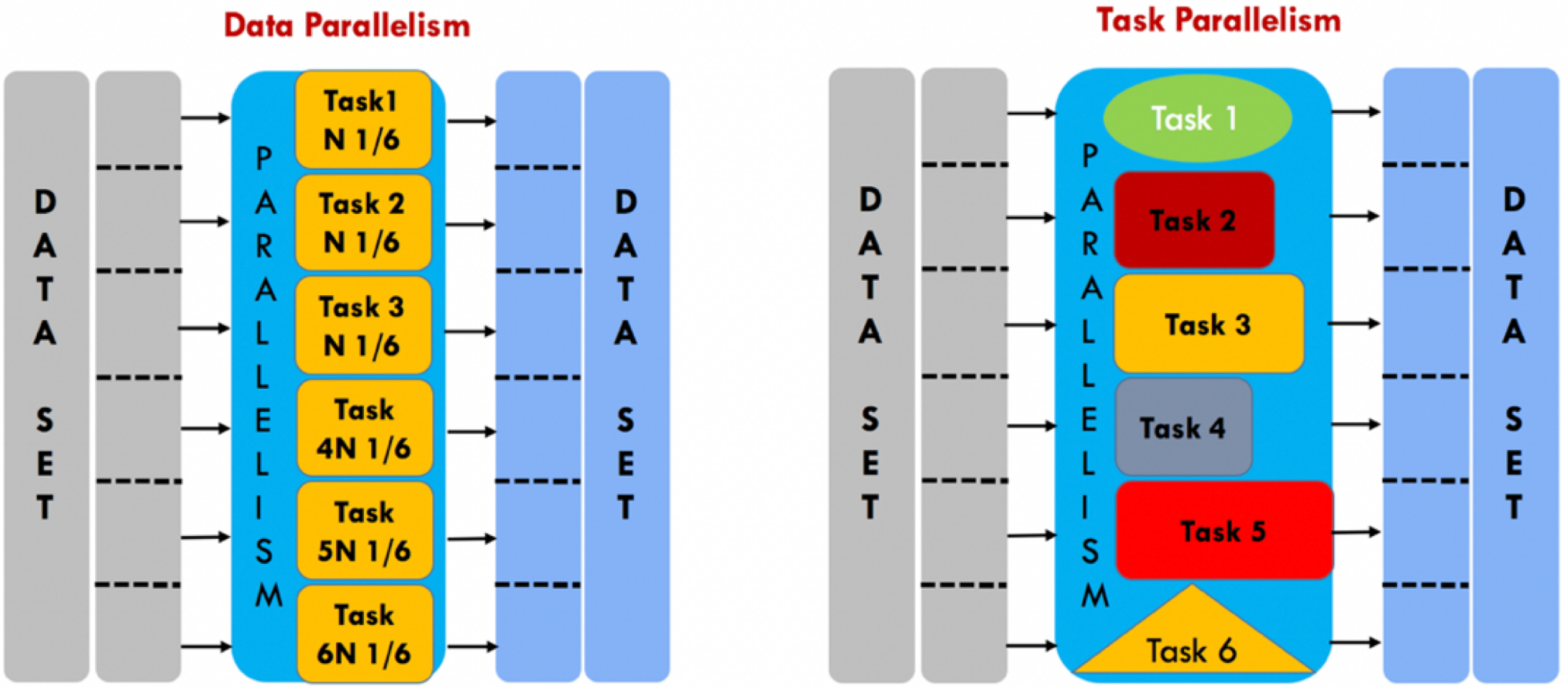
“How to parallelize my code?”

Diagnosis & Fix:

User’s code was CPU-bound:

should not use async/await, but Task or PLINQ

Task and Data Parallelism



Using Task<T> directly

Gives **fine-grained** control over parallelism

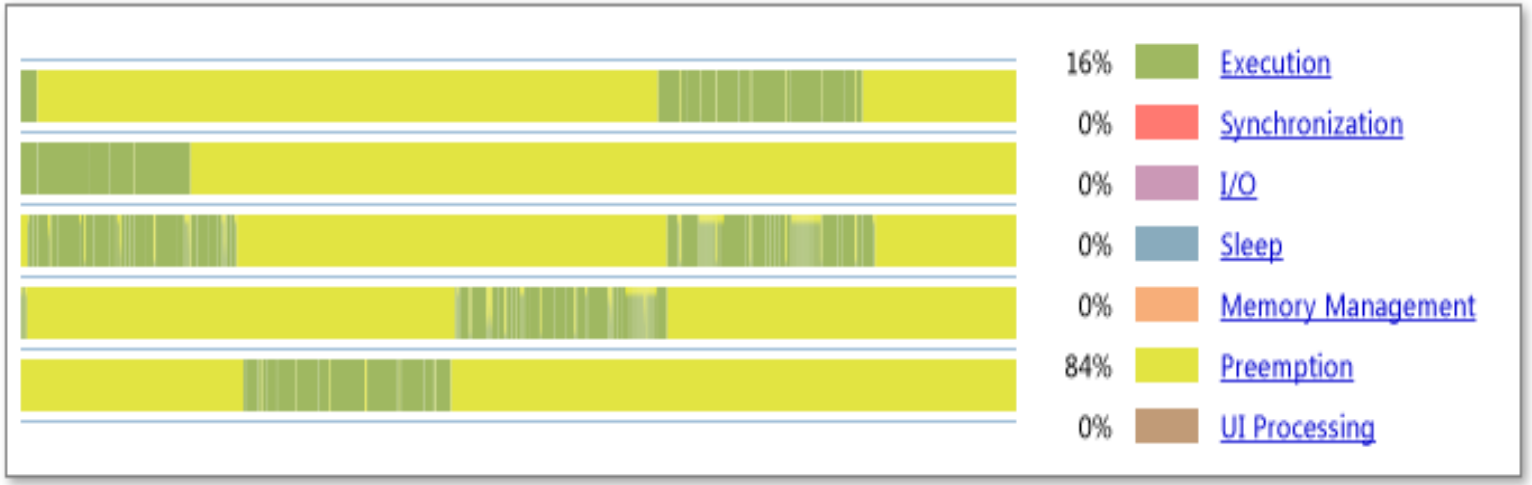
- Task represents small work item, scales very well

```
let toGray = Task.Factory.StartNew(fun () ->
    setToGray source1 layer1)
let rotate = Task.Factory.StartNew(fun () ->
    rotate source2 layer2)

Task.WhenAll(toGray, rotate)
blendTo (layer1, layer2) blended
```

Quick Sort

```
static void QuickSort_Parallel<T>(T[] items, int left, int right)
{
    if (right - left < 2)
    {
        if (left+1 == right &&
            items[left].CompareTo(items[right]) > 0)
            Swap(ref items[left], ref items[right]);
        return;
    }

    int pivot = Partition(items, left, right);

    Task leftTask = Task.Run(() => QuickSort_Parallel(items, left, pivot));
    Task rightTask = Task.Run(() => QuickSort_Parallel(items, pivot + 1, right));

    Task.WaitAll(leftTask, rightTask);
}
```

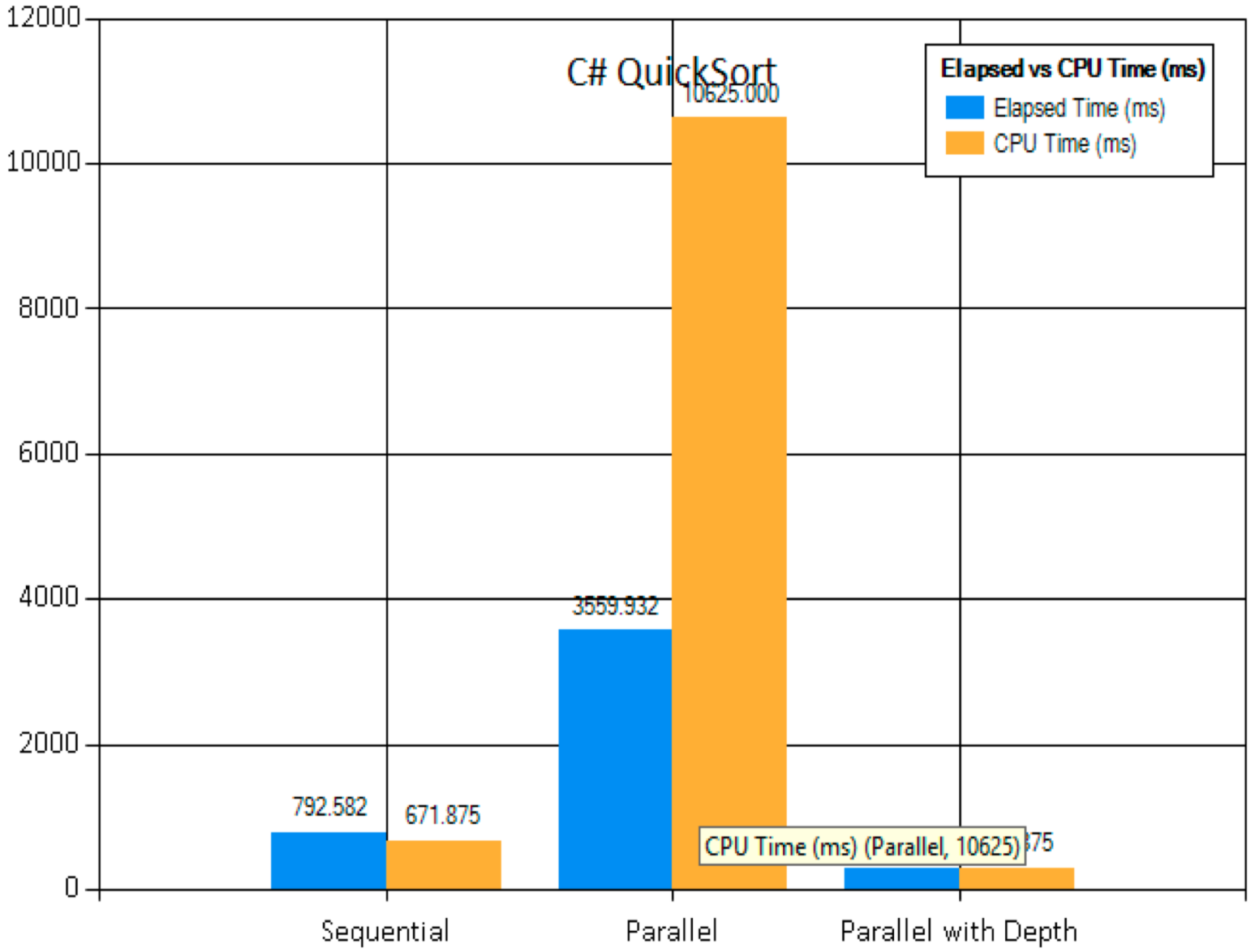
Quick Sort – Oversubscription



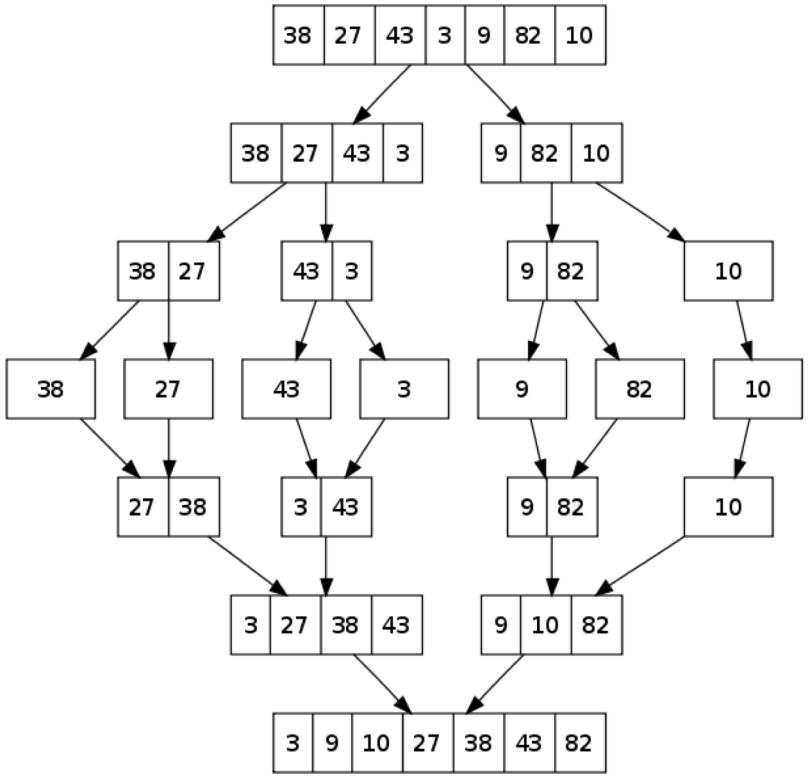
Quick Sort – with depth

```
void QuickSort_Parallel_Threshold<T>(T[] items, int left, int right, int depth)
{
    if (left >= right)  return;
    SwapElements(items, left, (left + right) / 2);
    var pivot = Partition(items);
    if (depth >= 0)     {
        QuickSort_Parallel_Threshold(items, left, pivot - 1, depth);
        QuickSort_Parallel_Threshold(items, pivot + 1, right, depth);
    }
    else
    {
        --depth;
        Parallel.Invoke(
                () => QuickSort_Parallel_Threshold(items, left, pivot - 1, depth),
                () => QuickSort_Parallel_Threshold(items, pivot + 1, right, depth)
            );
    }
```

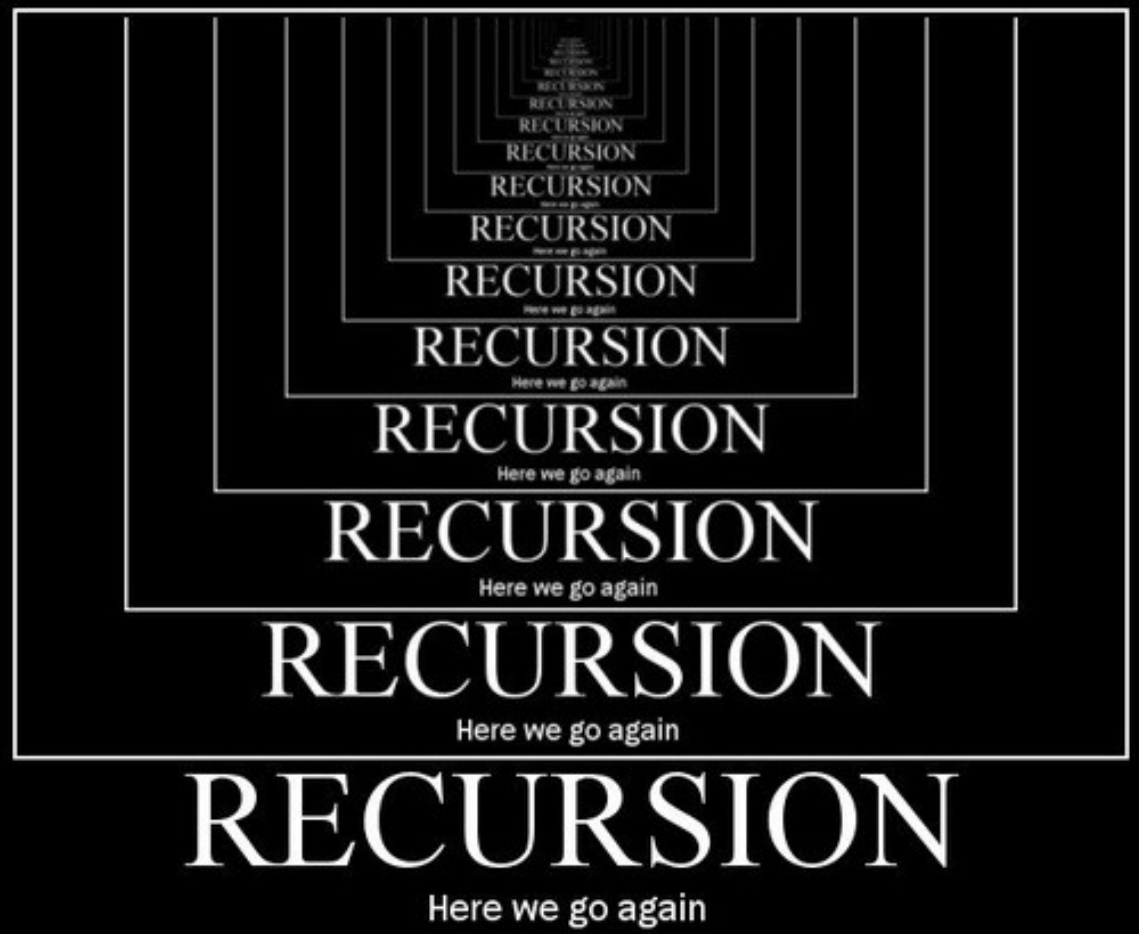
Quick Sort benchmark



Divide and Conquer

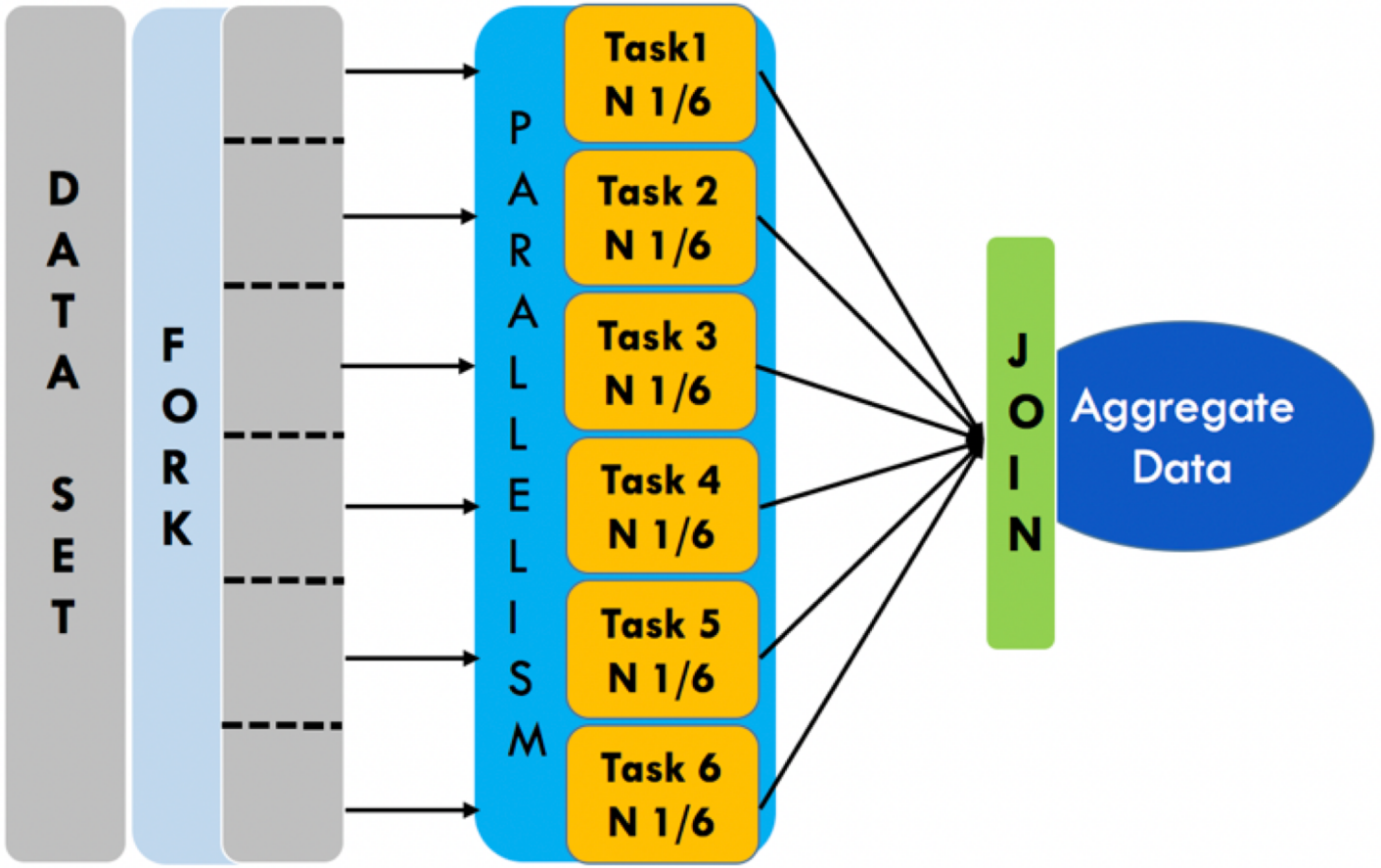



```
void QuickSort_Parallel<T>(T[] items, int left, int right)
{
    int pivot = left;


    SwapElements(items, left, pivot);

    Parallel.Invoke(
            () => QuickSort_Parallel(items, left, pivot - 1),
            () => QuickSort_Parallel(items, pivot + 1, right)
        );
}
```


RECURSION
Here we go again

Task parallel programming in action

The Fork/Join pattern



Fork/Join

```
R InvokeParChildRelationship<T, R>(Func<R, T, R> reduce, Func<R> seedInit,
                                                        params Func<T>[] operations)
{
    var results = new T[operations.Length];
    var task = Task.Run(() =>
    {
        for (int i = 0; i < operations.Length; i++)
        {
            int index = i;
            Task.Factory.StartNew(() => results[index] = operations[index](),
                                                    TaskCreationOptions.AttachedToParent);
        }
    });
    Task.WhenAny(task);
    return results.Aggregate(seedInit(), reduce);
}
```

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

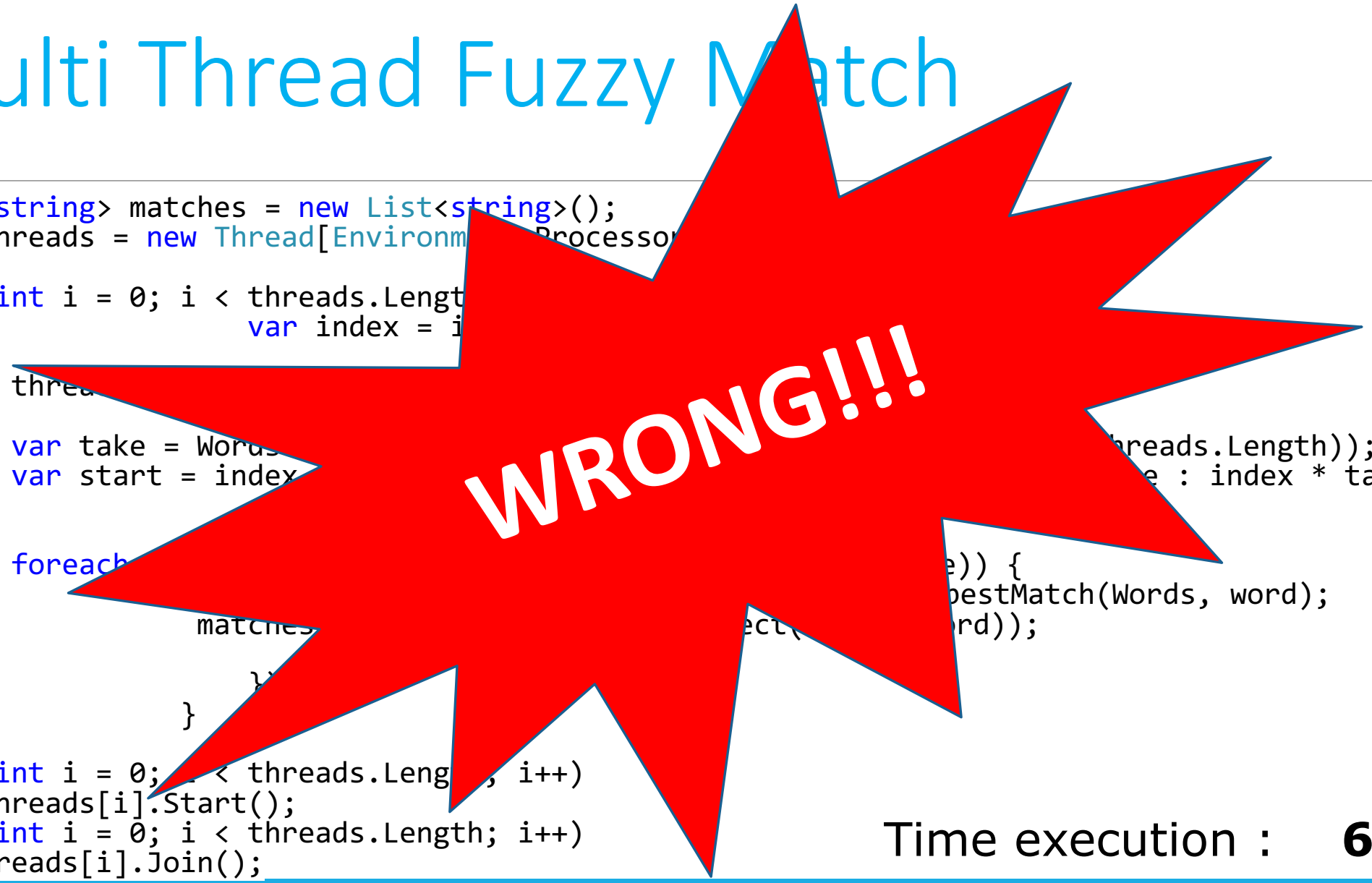
Two Threads Fuzzy Match

```
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));
            }
        });
```

Time execution : **15,436** ms

Multi Thread Fuzzy Match

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

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

      thre

      var take = Word                                    hreads.Length));
      var start = index                                  e : index * take;


      foreach                                            e)) {
                                                         estMatch(Words, word);
                  matche                                 rd));

                  }
              }

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



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

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>**

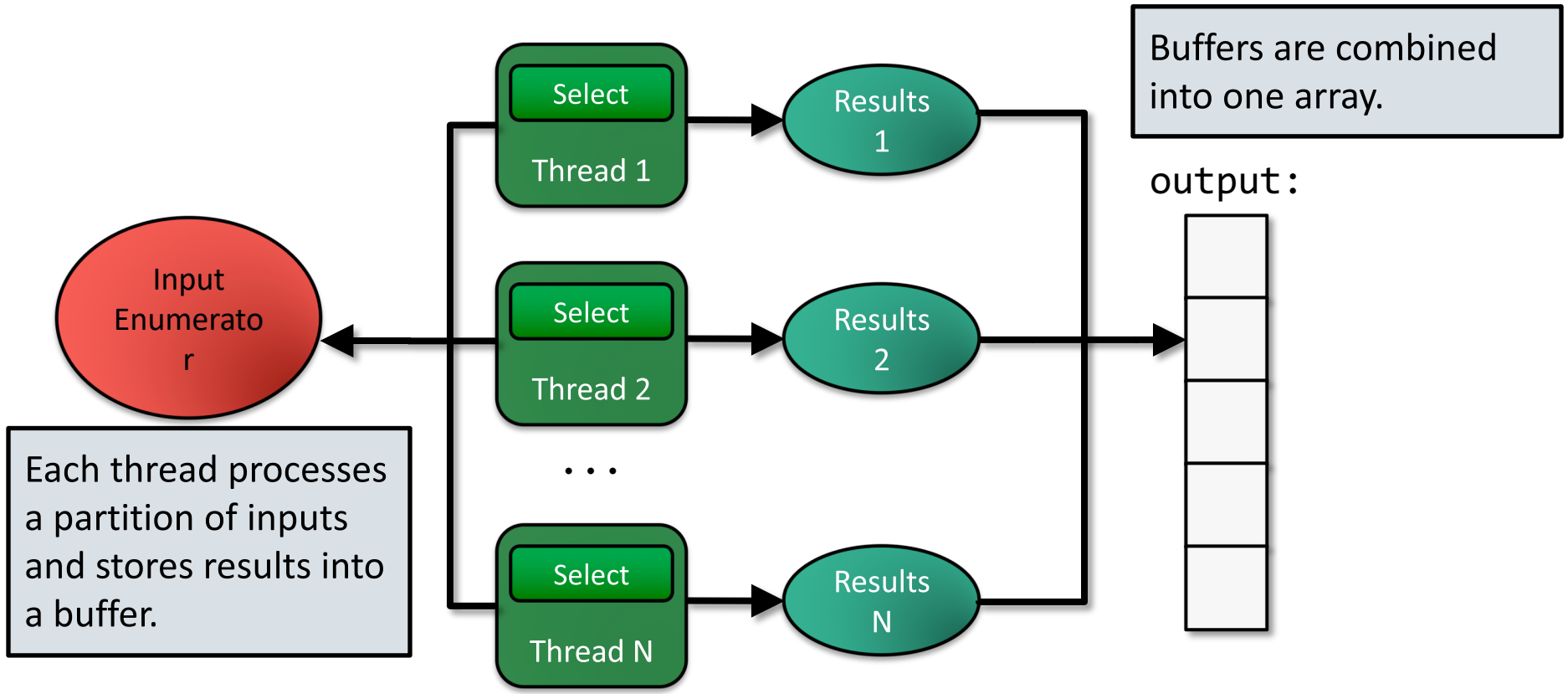
Select() | Where() | Etc.

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.WhenAll(tasks.ToArray(), (ts) =>
      return new List<string>(tasks.SelectMany(t => t.Result).Distinct()));
```

Time execution : **4,292** ms

Partitioning

Partitioning in PLINQ

Range Partitioning

Chunk Partitioning

Striped Partitioning

Hash Partitioning

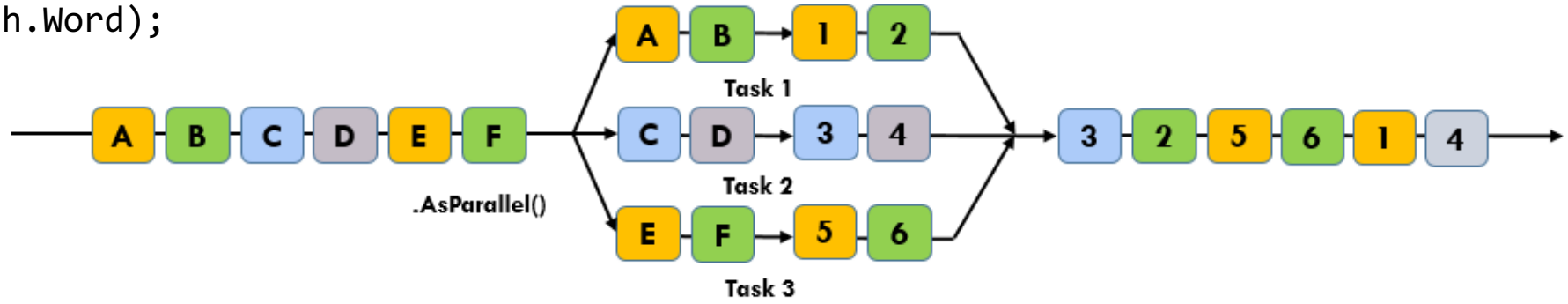
PLINQ Fuzzy Match

```
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

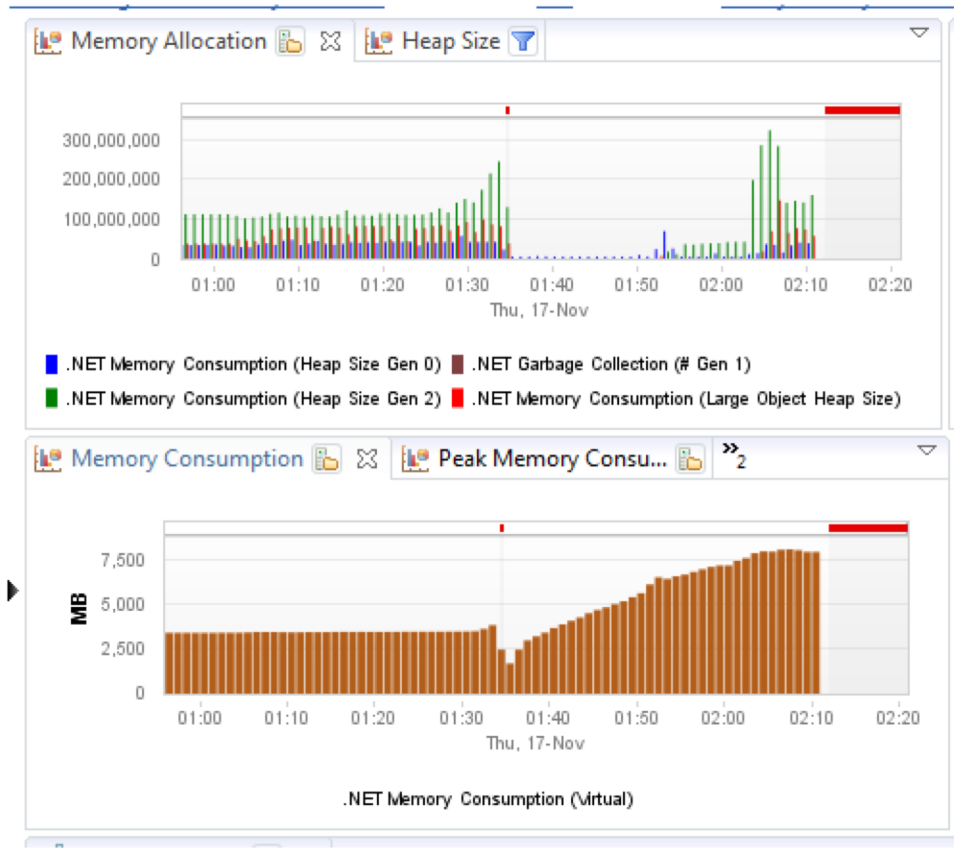
```
Create a load-balancing partitioner, specify false for static partitioning.

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);
```

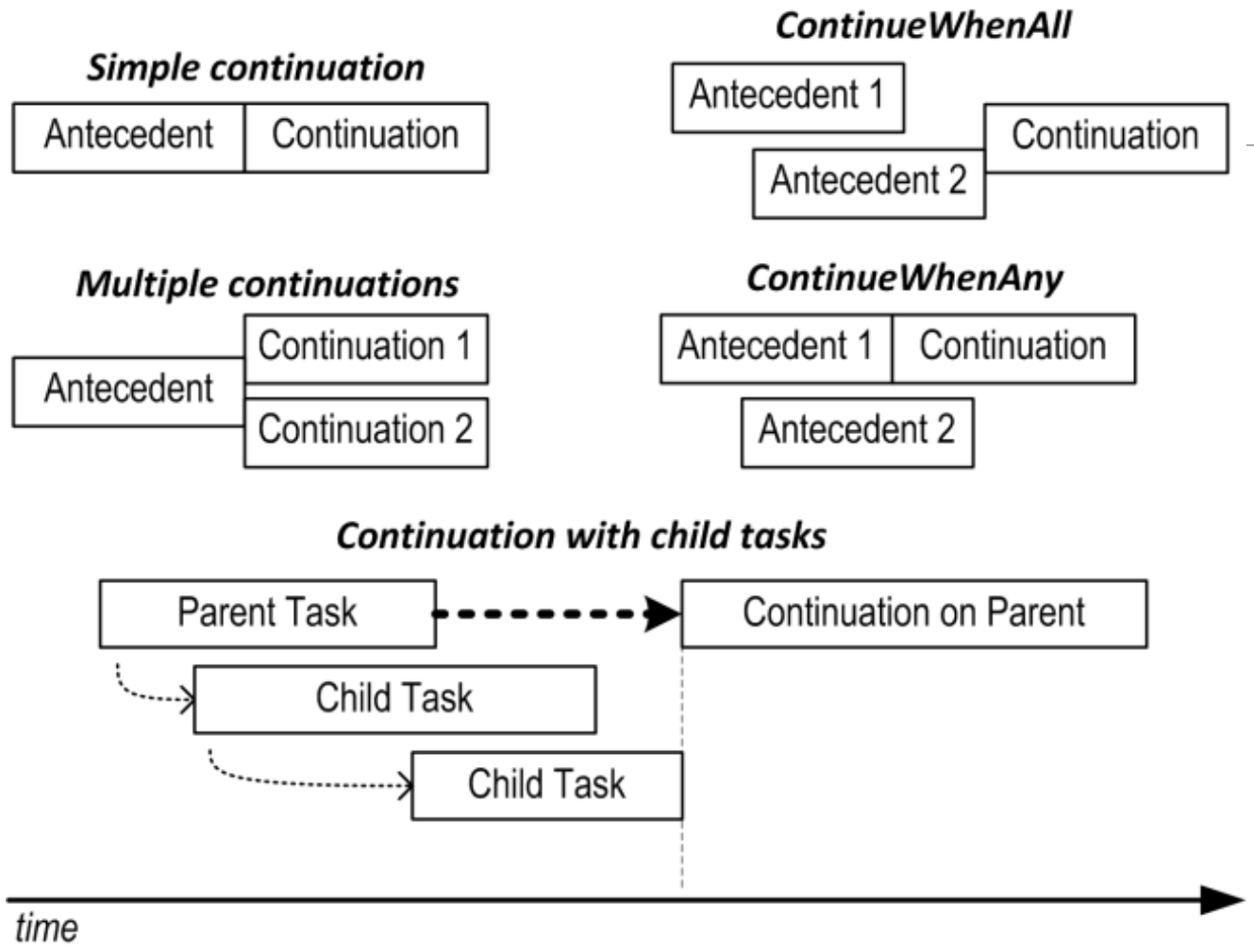


Deforesting your query

Deforesting your query

```
var data = new int[100000];
for(int i = 0; i < data.Length; i++)
    data[i]=i;


long total =
    data.AsParallel()
        .Where(n => n % 2 == 0)
        .Select(n => n + n)
        .Select(n=> (long)n)
        .Sum();
```





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);
```

Task practices

* **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**.

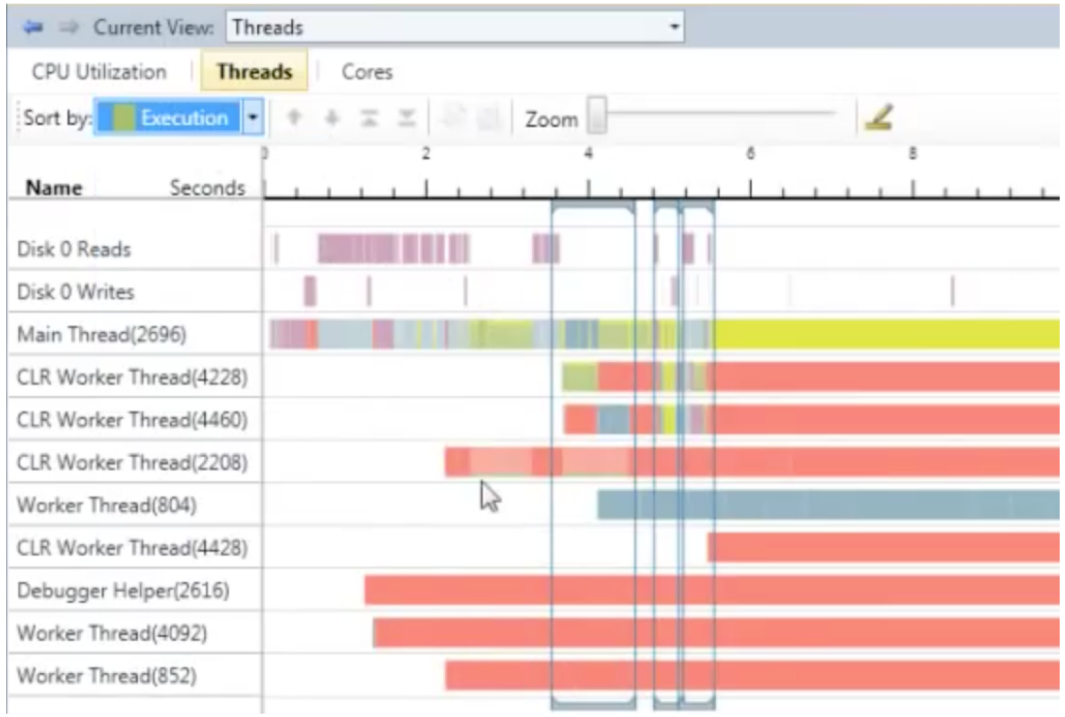
* **Avoid sharing data between tasks by applying a defensive copy approach.**

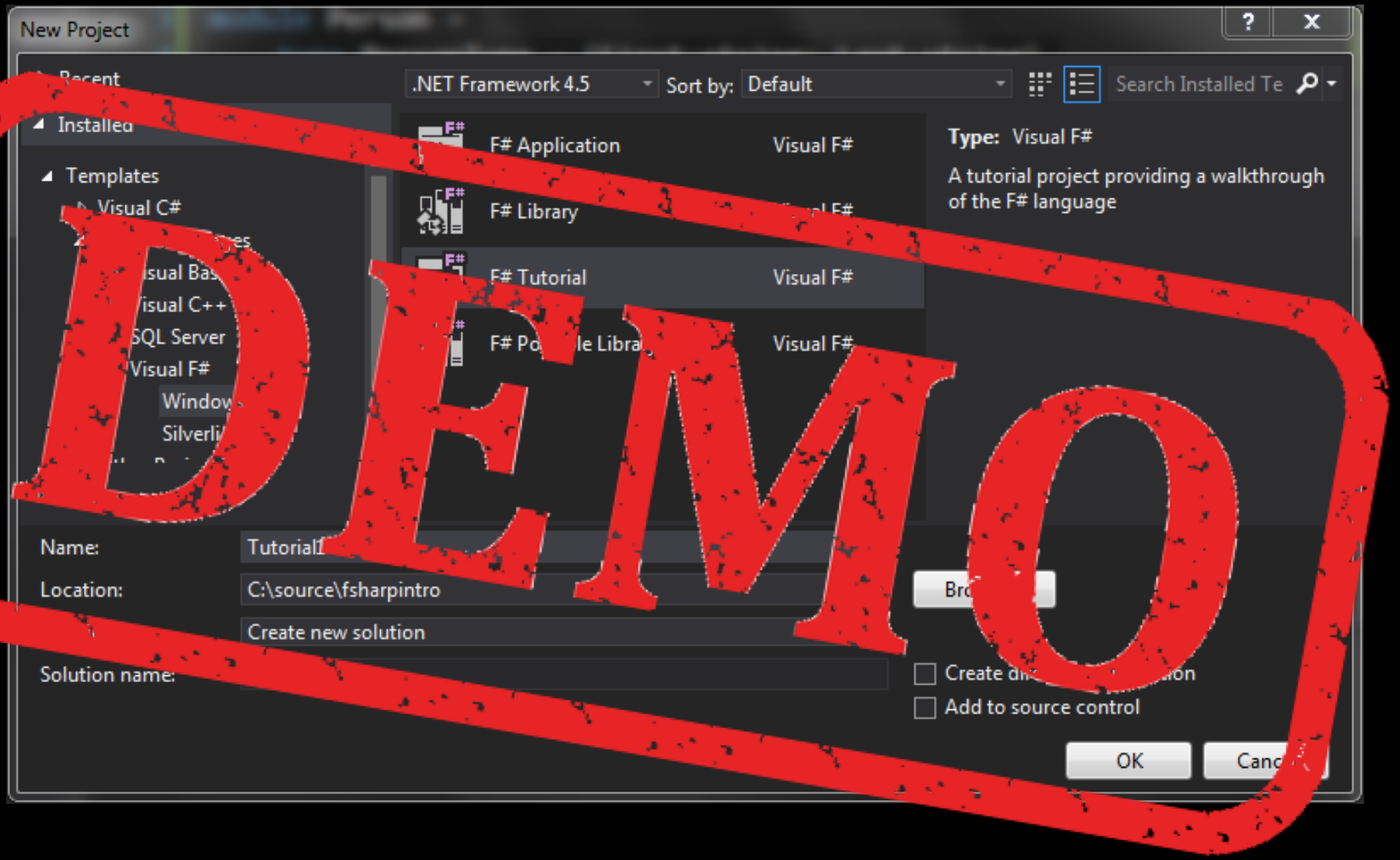
* Use immutable structures when data sharing between tasks cannot be avoided.

Measure twice and cut once

Assume nothing – Profile everything

- Be scientific
  - Do test multiple implementations
  - Don’t set out to confirm your bias
  - Instrument and profile your code




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 continuation

Task continuation

```
void RunContinuation<T>(Func<T> input, Action<T> rest)
        => ThreadPool.QueueUserWorkItem(new WaitCallback(o => rest(input())));


Action<string> printMsg = msg =>
    Console.WriteLine($"ThreadID = {Thread.CurrentThread.ManagedThreadId}, Url = {url}, {msg}");

RunContinuation(() => {
    printMsg("Creating webclient...");
    return new System.Net.WebClient();
}, (webclient) => RunContinuation(() => {
        printMsg("Downloading url...");
        return webclient.DownloadString(url);
    }, (html) => RunContinuation(() => {
            printMsg("Extracting urls...");
            return Regex.Matches(html, @"http://\S+");
        }, (matches) =>
                printMsg("Found " + matches.Count.ToString() + " links")
            )));
```

Task continuation

```
ThreadID = 19, Url = http://www.google.com/, Creating webclient...
ThreadID = 21, Url = http://microsoft.com/, Creating webclient...
ThreadID = 16, Url = http://www.wordpress.com/, Creating webclient...
ThreadID = 16, Url = http://www.wordpress.com/, Downloading url...
ThreadID = 19, Url = http://www.google.com/, Downloading url...
ThreadID = 21, Url = http://microsoft.com/, Downloading url...
ThreadID = 16, Url = http://www.wordpress.com/, Extracting urls...
ThreadID = 16, Url = http://www.wordpress.com/, Found 15 links
ThreadID = 16, Url = http://www.peta.org, Creating webclient...
ThreadID = 16, Url = http://www.peta.org, Downloading url...
ThreadID = 19, Url = http://www.google.com/, Extracting urls...
ThreadID = 19, Url = http://www.google.com/, Found 14 links
ThreadID = 20, Url = http://www.peta.org, Extracting urls...
ThreadID = 20, Url = http://www.peta.org, Found 60 links
ThreadID = 21, Url = http://microsoft.com/, Extracting urls...
ThreadID = 21, Url = http://microsoft.com/, Found 1 links
```

Task continuation

```
Task.Run(() =>
{
    printMsg("Creating webclient...");
    return new System.Net.WebClient();
}).ContinueWith(taskWebClient =>
{
    var webclient = taskWebClient.Result;
    printMsg("Downloading url...");
    return webclient.DownloadString(url);
}).ContinueWith(htmlTask =>
{
    var html = htmlTask.Result;
    printMsg("Extracting urls...");
    return Regex.Matches(html, @"http://\S+");
}).ContinueWith(rgxTask =>
{
    var matches = rgxTask.Result;
    printMsg("Found " + matches.Count.ToString() + " links");
});
```

Continuation Passing Style

```
void GetMaxCPS(int x, int y, Action<int> f)
{
    if (x > y)
        f(x);
    else
        f(y);
}
```

Task continuation

```
var matches = from webclient in Task.Run(() => new System.Net.WebClient())
              from html in Task.Run(() => webclient.DownloadString(url))
              from rgx in Task.Run(() => Regex.Matches(html, @"http://\S+"))
              select rgx;
```

Mathematical patterns for better composition

Task is a Monad

How can we compose Tasks?

C#

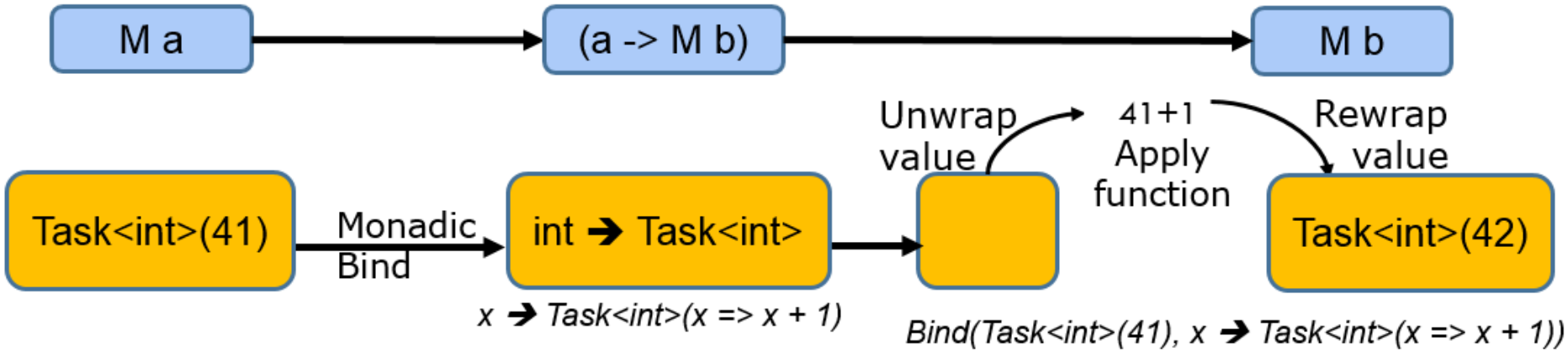
```
Task<int> RunOne() => // ...
Task<decimal> RunTwo(int input) => // ...

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

C#

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

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

Composing Tasks

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

Task<T> Return(T value) …
```

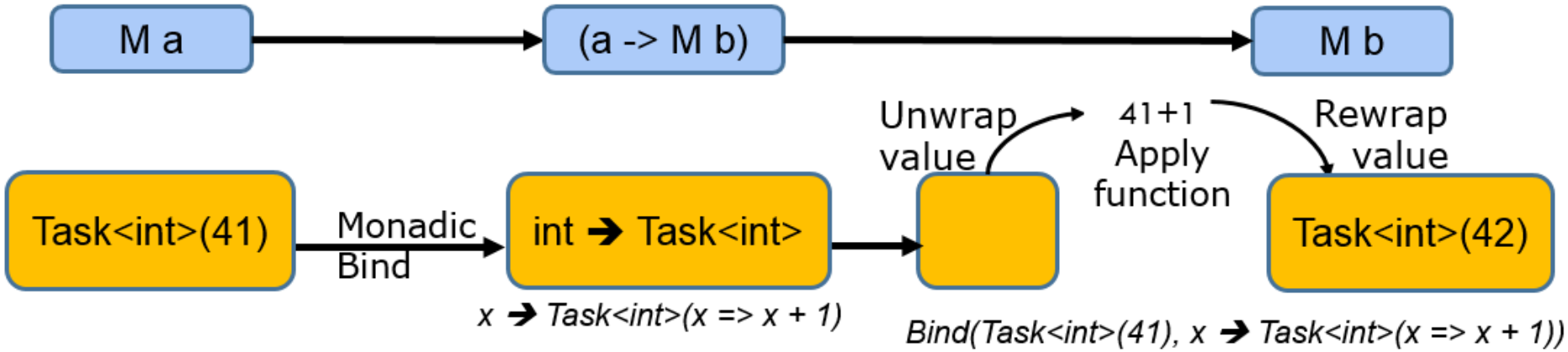


Functional Design Patterns – Monad

Generic type

With function (>>=): bind : M<‘a> -> (‘a -> M<‘b>) -> M<‘b>

With function: return : ‘a -> M<‘a>



Functional Design Patterns – 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))

Functional Design Patterns – Monad

F#

```
let bind (m:Option<a>, f:a -> Option<b>) =
    match m with
    | Some x -> f x
    | None -> None

let return' x = Some(x)
```

Computation Expression motivation

- Way to extend regular F# syntax
- Nice syntax for continuation passing
- Hiding the chaining logic for us
- Abstract side effects

Computation Expression basics

Any type with specific functions

Method	Typical signature	Description
Bind	M<'T> * ('T -> M<'U>) -> M<'U>	Called for let! and do! in computation expressions.
Return	'T -> M<'T>	Called for return in computation expressions.

Task Bind and Task Return

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

 // M<'T> * ('T -> M<'U>) -> M<'U>
static member Bind (input : Task<'T>, binder : 'T -> Task<'U>) : Task<'U>



 // M<'T> * ('T -> M<'U>) -> M<'U>
static member SelectMany (input : Task<'T>, binder : 'T -> Task<'U>) : Task<'U>
```

SelectMany

```
// M<T> * (T -> M<U>) -> M<U>
IEnumerable<R> SelectMany<T, R>( this IEnumerable<T> source,
                                 Func<T, IEnumerable<R>> selector )


// M<T> * (T -> M<U>) -> (T -> M -> U) -> M<U>
IEnumerable<R> SelectMany<T, M, R>( this IEnumerable<T> source,
                Func<T, IEnumerable<M>> collectionSelector, Func<T, M, R> resultSelector )


// M<T> * (T -> M<U>) -> M<U>
IEnumerable<R> Select<T, R>( this IEnumerable<T> source, Func<T, R> selector )
```

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) …
```

Task SelectMany

```
// M<T> * (T -> M<U>) -> M<U>
IEnumerable<R> Task<T, R>( this Task<T> source,
                                        Func<T,  Task<R>> selector )


// M<T> * (T -> M<U>) -> (T -> M -> U) -> M<U>
Task<R> SelectMany<T, M, R>( this Task<T> source,
                Func<T,  Task<M>> collectionSelector, Func<T, M, R> resultSelector )

// M<T> * (T -> M<U>) -> M<U>
Task<R> Select<T, R>( this Task<T> source, Func<T, R> selector )
```

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;
```

LAB :
parallel face detection

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 SelectMany

```
 // 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
```

Task Inline - ExecuteSynchronously

```
public static Task<TOut> SelectMany<TIn, TOut>(this Task<TIn> first, Func<TIn,
Task<TOut>> next)
{
    var tcs = new TaskCompletionSource<TOut>();
    first.ContinueWith(delegate
    {
        if (first.IsFaulted) tcs.TrySetException(first.Exception.InnerExceptions);
        else if (first.IsCanceled) tcs.TrySetCanceled();
        else
        {
                next(first.Result).ContinueWith(t =>
                {
                    if (t.IsFaulted) tcs.TrySetException(t.Exception.InnerExceptions);
                    else if (t.IsCanceled) tcs.TrySetCanceled();
                    else tcs.TrySetResult(t.Result);
                }, TaskContinuationOptions.ExecuteSynchronously);
    }, TaskContinuationOptions.ExecuteSynchronously);
    return tcs.Task;
}
```

How can we compose Tasks?

C#

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

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

C#

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

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

Composing Tasks (Point free)

```
Task<string> DownloadStockHistoryAsync(string url) => // ...

Task<StockData[]> ConvertStockHistory(string symbol) => // ...
```

Composing Tasks (Point free) - Kleisli

```
Task<string> DownloadStockHistoryAsync(string url) => // ...

Task<StockData[]> ConvertStockHistory(string symbol) => // ...
```

```
static Func<T, Task<U>> Kleisli<T, R, U>(
Func<T, Task<R>> task1,
Func<R, Task<U>> task2) =>
          value => task1(value).Bind(task2);
```

Composing Tasks (Point free)

```
Task<string> DownloadStockHistoryAsync(string url) => // ...

Task<StockData[]> ConvertStockHistory(string symbol) => // ...

static Func<T, Task<U>> Kleisli<T, R, U>(
Func<T, Task<R>> task1,
Func<R, Task<U>> task2) =>
            value => task1(value).Bind(task2);


Task<StockData[]> ProcessStockHistory =>
     DownloadStockHistoryAsync.Kleisli(ConvertStockHistory);
```

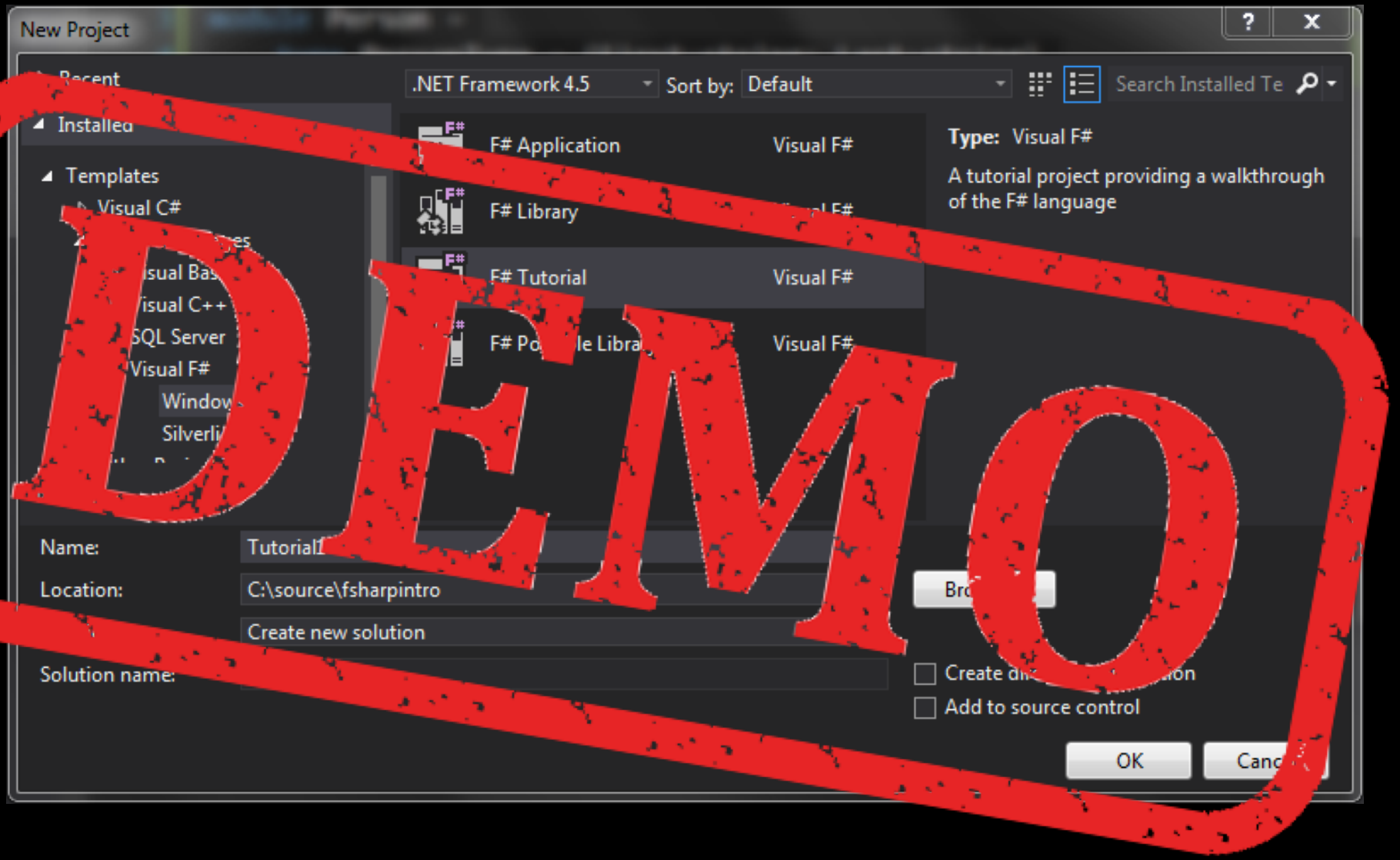
Composing Tasks with other elevated types

```
static Func<T, Task<U>> Kleisli<T, R, U>(
Func<T, Task<R>> task1,
Func<R, IEnumerable<U>> en) => // ...
```

```
static Func<T, Task<U>> Kleisli<T, R, U>(
Func<T, Task<R>> task1,
Func<R, IObservable<U>> obs) => // ...
```

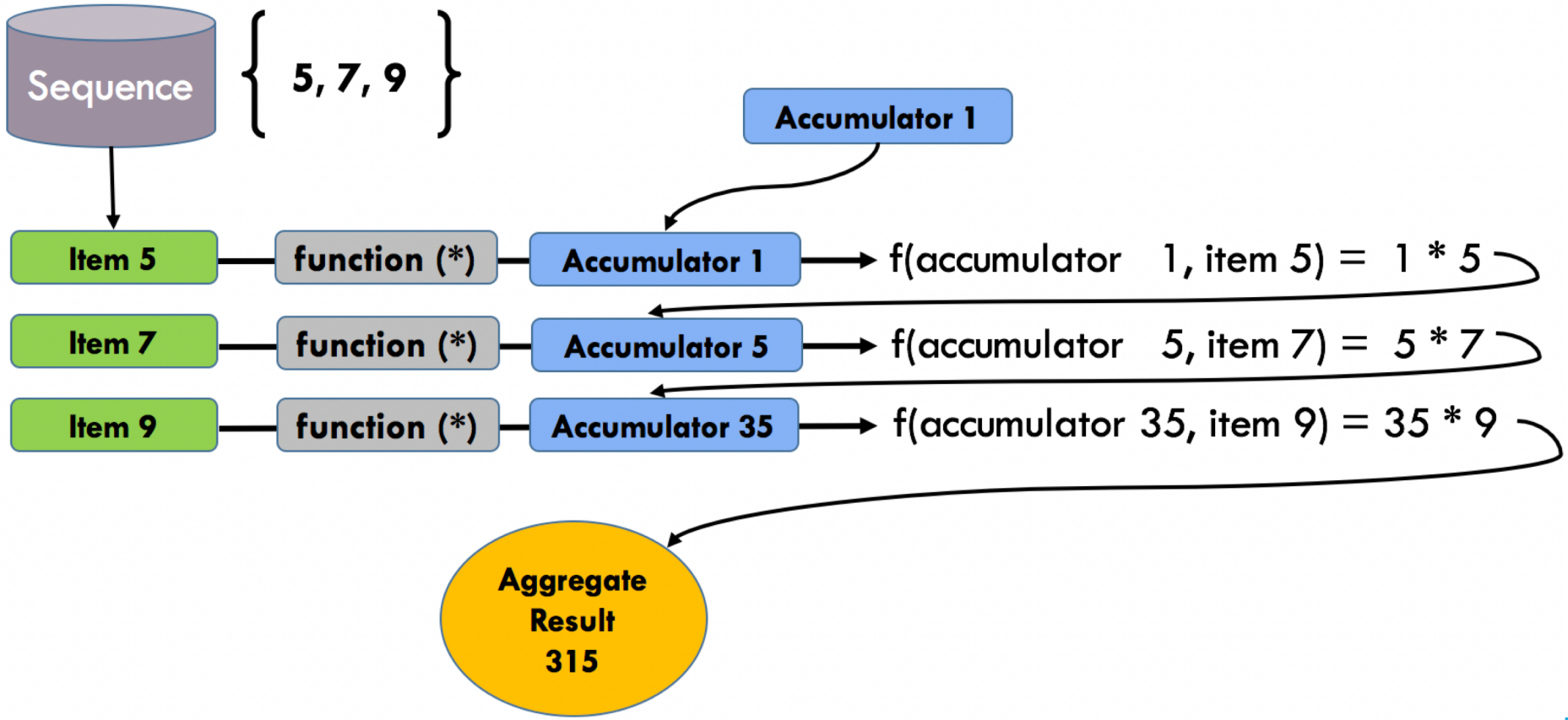
Kliesli operator in F#

```
fAsync:('a -> Async<'b>) -> gAsync:('b -> Async<'c>) -> arg:'a -> Async<‘c>

let Kliesli (fAsync:'a -> Async<'b>) (gAsync:'b -> Async<'c>) (arg:'a) =
async {
     let! f = Async.StartChild (fAsync arg)
     let! result = f
     return! gAsync result }


let (>=>) = Kliesli


let processStockHistory =
      DownloadStockHistoryAsync >=> ConvertStockHistory
```


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

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));
}
```

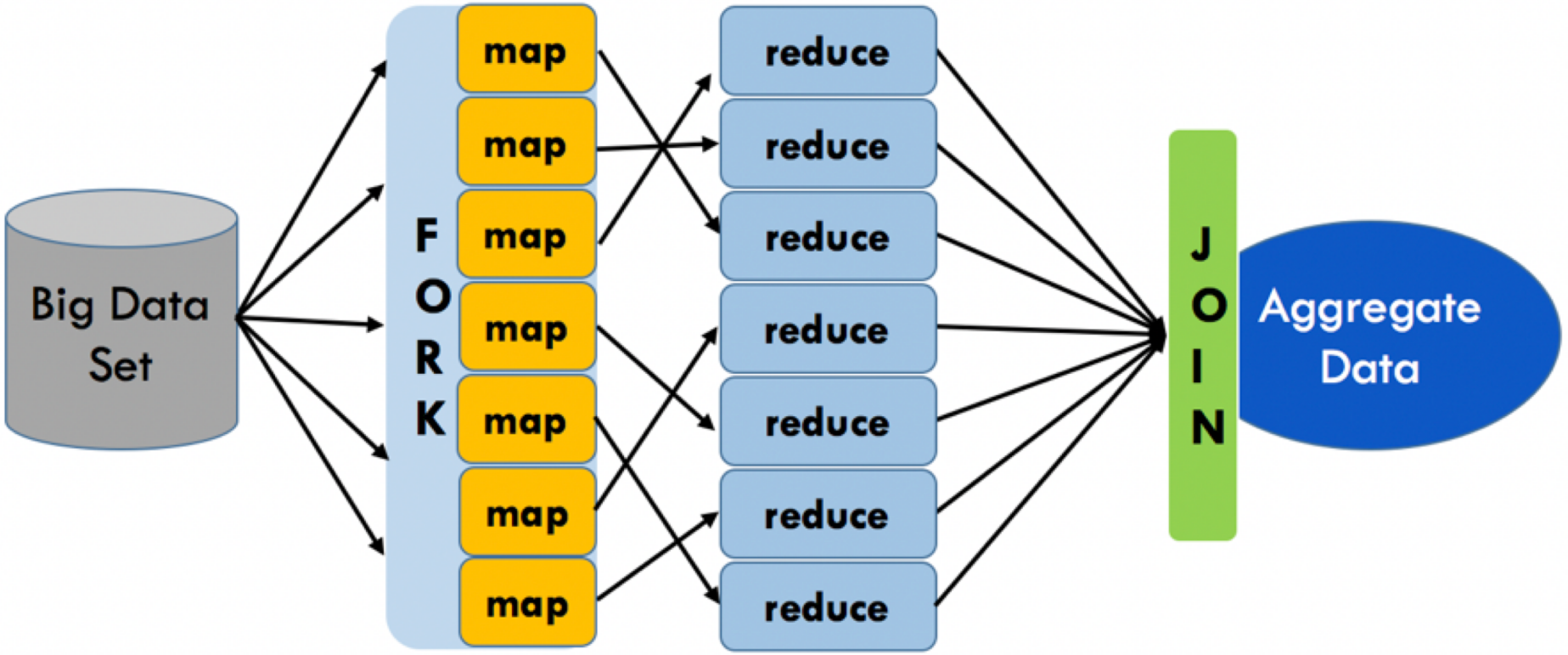
```
int[] source = Enumerable.Range(0, 100000).ToArray();
int result = source.AsParallel()
        .Reduce((value1, value2) => value1 + value2);
```

Map Reduce

The programming model is based upon five simple concepts:

1. Iteration over input
2. Computation of key/value pairs from each input
3. Grouping of all intermediate values by key
4. Iteration over the resulting groups
5. Reduction of each group

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, [])

Monoid law: Associativity

- Ex: 1 + (2 + 3) = (1 + 2) + 3

Map Reduce implementations

The work required to calculate the result can be summarized in this steps:

1. Read every line in the file
2. Convert each line into words
3. Exclude invalid words e.g. "a", "and", "with" etc.
4. Count and track the occurrence of each word
5. Sort the words from the highest count to the lowest
6. Return the top *n* words e.g. *100*

For each of the patterns implemented will be measured the total execution time and the count of times *Garbage Collection* occurs across different generations (0, 1 and 2)

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

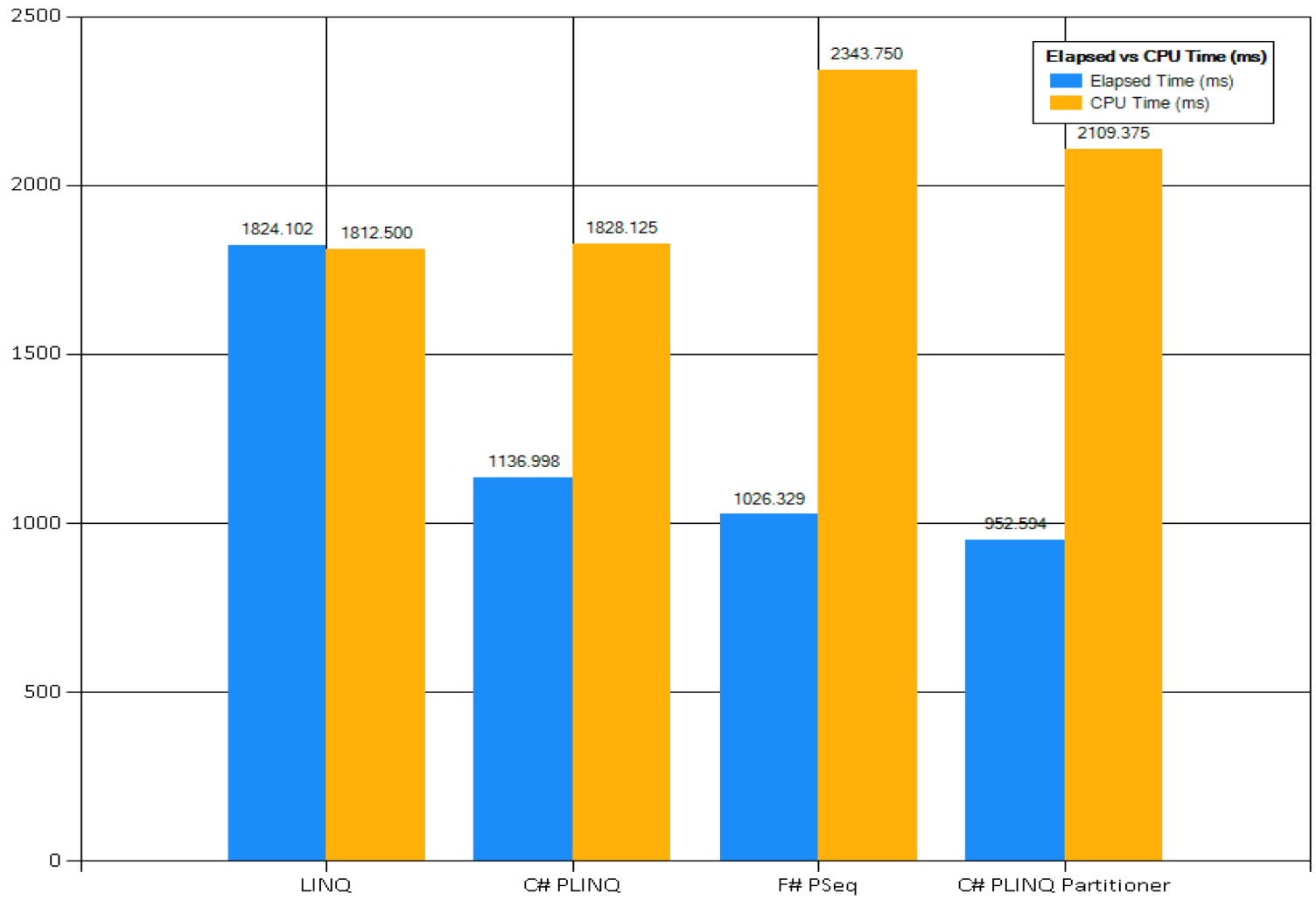
```
Directory.GetFiles(path, "*.*", SearchOption.AllDirectories).AsParallel()
    .Select(f =>
        {   using (var fs = new FileStream(f, FileMode.Open, FileAccess.Read))
                return new
                {
                    FileName = f,
                    FileHash = BitConverter.ToString(SHA1.Create().ComputeHash(fs))
                };
        })

    .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
```


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

Map Reduce
Some experiments

Sequential implementation as baseline

```
var result = new Dictionary<string, uint>(StringComparer.InvariantCultureIgnoreCase);

foreach (var line in File.ReadLines(InputFile.FullName))
   foreach (var word in line.Split(Separators, StringSplitOptions.RemoveEmptyEntries))
   {
        if (!IsValidWord(word)) { continue; }
        TrackWordsOccurrence(result, word);
   }

 return result.OrderByDescending(kv => kv.Value)
              .Take((int)TopCount)
              .ToDictionary(kv => kv.Key, kv => kv.Value);


TrackWordsOccurrence(IDictionary<string, uint> wordCounts, string word)
     => wordCounts[word] =
          wordCounts.TryGetValue(word, out uint count)
          ? count + 1 : 1;
```

Execution time: 127,025 ms
Gen-0: 37, Gen-1: 16, Gen-2: 7

Sequential implementation LINQ

```
File.ReadLines(InputFile.FullName)
            .SelectMany(l => l.Split(Separators, StringSplitOptions.RemoveEmptyEntries))
            .Where(IsValidWord)
            .ToLookup(x => x, StringComparer.InvariantCultureIgnoreCase)
            .Select(x => new { Word = x.Key, Count = (uint)x.Count() })
            .OrderByDescending(kv => kv.Count)
            .Take((int)TopCount)
            .ToDictionary(kv => kv.Word, kv => kv.Count);
```

Execution time: 182,236 ms
Gen-0: 21, Gen-1: 10, Gen-2: 4

Garbage Collector mode

GC mode to Server increases the application's throughput offloading a significant subset of the *Gen 2 GC* by using dedicated background threads resulting in a significant reduction of the total pause times on the user threads

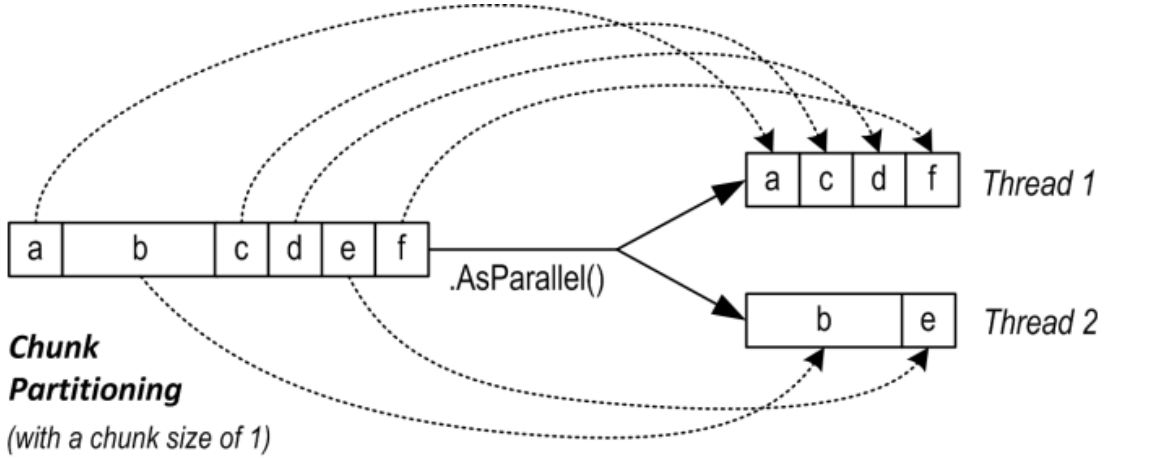
https://blogs.msdn.microsoft.com/dotnet/2012/07/20/the-net-framework-4-5-includes-new-garbage-collector-enhancements-for-client-and-server-apps/

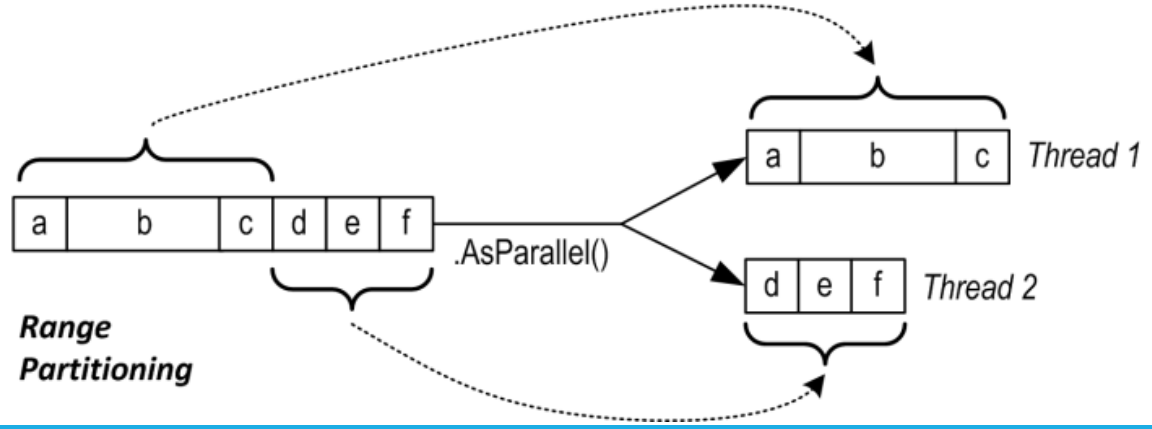
Parallel implementation PLINQ

```
var words = File.ReadLines(InputFile.FullName)
    .AsParallel()
    .SelectMany(l => l.Split(Separators, StringSplitOptions.RemoveEmptyEntries))
    .Where(IsValidWord);


var result = new Dictionary<string, uint>(StringComparer.InvariantCultureIgnoreCase);
foreach (var word in words)
    TrackWordsOccurrence(result, word);


return result
    .OrderByDescending(kv => kv.Value)
    .Take((int)TopCount)
    .ToDictionary(kv => kv.Key, kv => kv.Value);
```

Execution time: 96,409 ms
Gen-0: 11, Gen-1: 7, Gen-2: 4

PLINQ + Partition





Parallel implementation PLINQ + Partition

```
return File.ReadLines(InputFile.FullName)
    .AsParallel()
    .SelectMany(l => l.Split(Separators, StringSplitOptions.RemoveEmptyEntries))
    .Where(IsValidWord)
    .ToLookup(x => x, StringComparer.InvariantCultureIgnoreCase)
    .AsParallel()
    .Select(x => new { Word = x.Key, Count = (uint)x.Count() })
    .OrderByDescending(kv => kv.Count)
    .Take((int)TopCount)
    .ToDictionary(kv => kv.Word, kv => kv.Count);
```

Execution time: 29,638 ms
Gen-0: 6, Gen-1: 3, Gen-2: 3

Parallel implementation PLINQ MapReduce

```
return File.ReadLines(InputFile.FullName).AsParallel().Aggregate(
        () => new Dictionary<string, uint>(StringComparer.InvariantCultureIgnoreCase),
        (localDic, line) => { //#2
            foreach (var word in line.Split(Separators, StringSplitOptions.RemoveEmptyEntries)
                                     .Where(IsValidWord))
                TrackWordsOccurrence(localDic, word);
            return localDic;
        },
        (finalResult, localDic) => {
            foreach (var pair in localDic) {
                if (finalResult.ContainsKey(pair.Key))
                    finalResult[pair.Key] += pair.Value;
                else
                    finalResult[pair.Key] = pair.Value;
            }
            return finalResult;
        },
        finalResult => finalResult
            .OrderByDescending(kv => kv.Value)
            .Take((int)TopCount)
            .ToDictionary(kv => kv.Key, kv => kv.Value));
```

Execution time: 27,990 ms
Gen-0: 10, Gen-1: 6, Gen-2: 4

Producer-Consumer PLINQ (part 1)

```
var result = new ConcurrentDictionary<string, uint>(StringComparer.InvariantCultureIgnoreCase);
var blockingCollection = new BlockingCollection<string>(BoundedCapacity);

Action<string> work = line => {
 foreach (var word in line.Split(Separators, StringSplitOptions.RemoveEmptyEntries)
                                                                .Where(IsValidWord))
        result.AddOrUpdate(word, 1, (key, oldVal) => oldVal + 1);
};

Task.Run(() => {
    foreach (var line in File.ReadLines(InputFile.FullName))
        blockingCollection.Add(line);
    blockingCollection.CompleteAdding();
});
```

Producer-Consumer PLINQ (part 2)

```
blockingCollection
    .GetConsumingEnumerable()
    .AsParallel()
    .WithDegreeOfParallelism(WorkerCount)
    .WithMergeOptions(ParallelMergeOptions.NotBuffered)
    .ForAll(work);


return result
    .OrderByDescending(kv => kv.Value)
    .Take((int)TopCount)
    .ToDictionary(kv => kv.Key, kv => kv.Value);
```

Execution time: 21,024 ms
Gen-0: 18, Gen-1: 10, Gen-2: 5

Producer-Consumer with Tasks

```
var tasks = Enumerable.Range(1, WorkerCount).Select(n =>
      Task.Factory.StartNew(work, CancellationToken.None, TaskCreationOptions.LongRunning,
                                                          TaskScheduler.Default)).ToArray();


foreach (var line in File.ReadLines(InputFile.FullName))
    blockingCollection.Add(line);
blockingCollection.CompleteAdding();


Task.WaitAll(tasks);


return result
    .OrderByDescending(kv => kv.Value)
    .Take((int)TopCount)
    .ToDictionary(kv => kv.Key, kv => kv.Value);
```

Execution time: 44,979 ms
Gen-0: 17, Gen-1: 9, Gen-2: 6

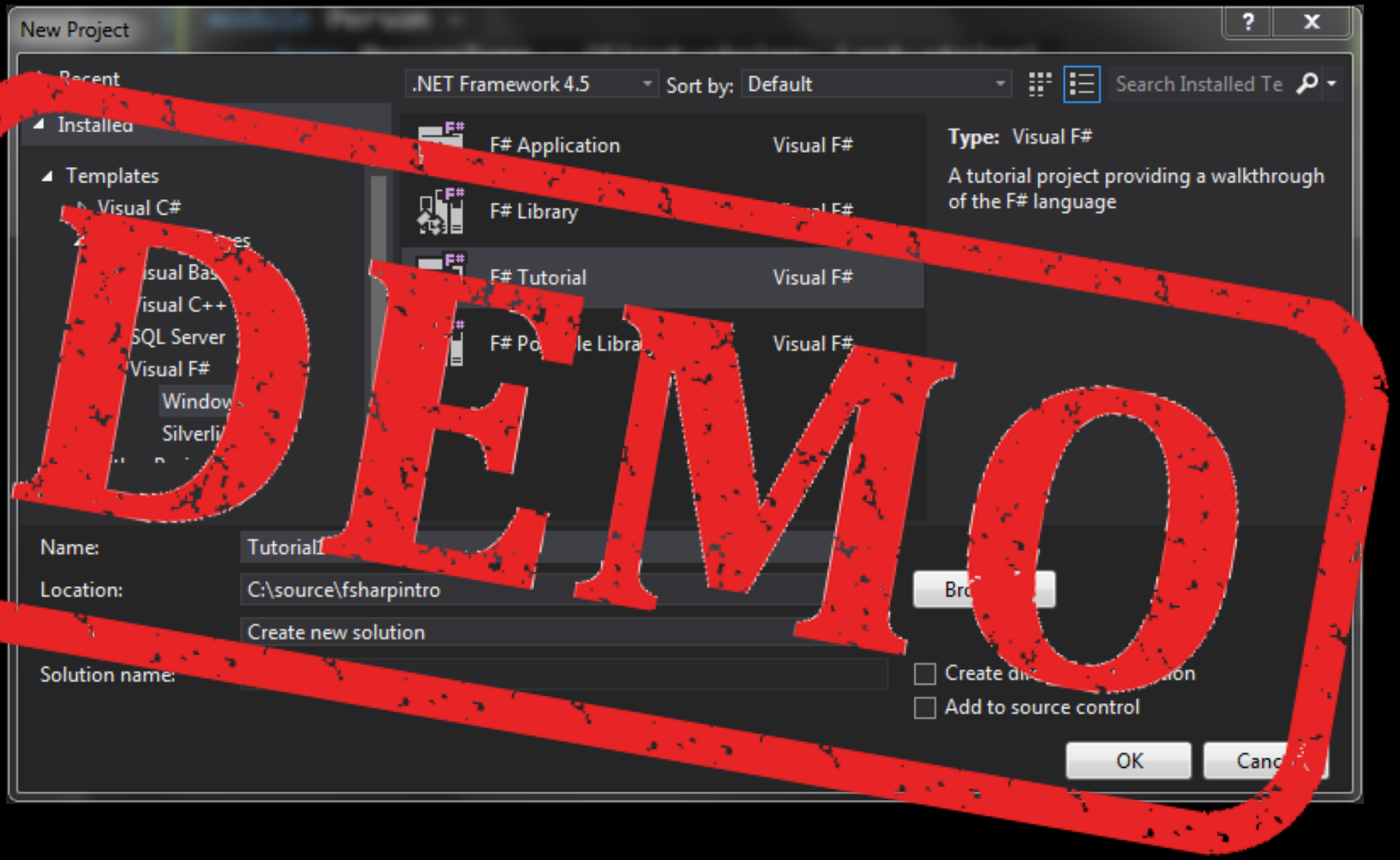
MapReduce Pipeline with TPL Dataflow (part 1)

```
var result = new ConcurrentDictionary<string, uint>(StringComparer.InvariantCultureIgnoreCase);
const int BoundedCapacity = 10000;

var bufferBlock = new BufferBlock<string>(
    new DataflowBlockOptions { BoundedCapacity = BoundedCapacity });

var splitLineToWordsBlock = new TransformManyBlock<string, string>(
    line => line.Split(Separators, StringSplitOptions.RemoveEmptyEntries),
    new ExecutionDataflowBlockOptions {
        MaxDegreeOfParallelism = 1,
        BoundedCapacity = BoundedCapacity
    });

var batchWordsBlock = new BatchBlock<string>(5000);
var trackWordsOccurrencBlock = new ActionBlock<string[]>(words => {
        foreach (var word in words.Where(word => IsValidWord(word)))
            result.AddOrUpdate(word, 1, (key, oldVal) => oldVal + 1);
    },
    new ExecutionDataflowBlockOptions { MaxDegreeOfParallelism = 8 });
```

MapReduce Pipeline with TPL Dataflow (part 2)

```
var defaultLinkOptions = new DataflowLinkOptions { PropagateCompletion = true };
bufferBlock.LinkTo(splitLineToWordsBlock, defaultLinkOptions);
splitLineToWordsBlock.LinkTo(batchWordsBlock, defaultLinkOptions);
batchWordsBlock.LinkTo(trackWordsOccurrencBlock, defaultLinkOptions);


foreach (var line in File.ReadLines(InputFile.FullName))
    bufferBlock.Post(line);


bufferBlock.Complete();
trackWordsOccurrencBlock.Completion.Wait();


return result
    .OrderByDescending(kv => kv.Value)
    .Take((int)TopCount)
    .ToDictionary(kv => kv.Key, kv => kv.Value);
```

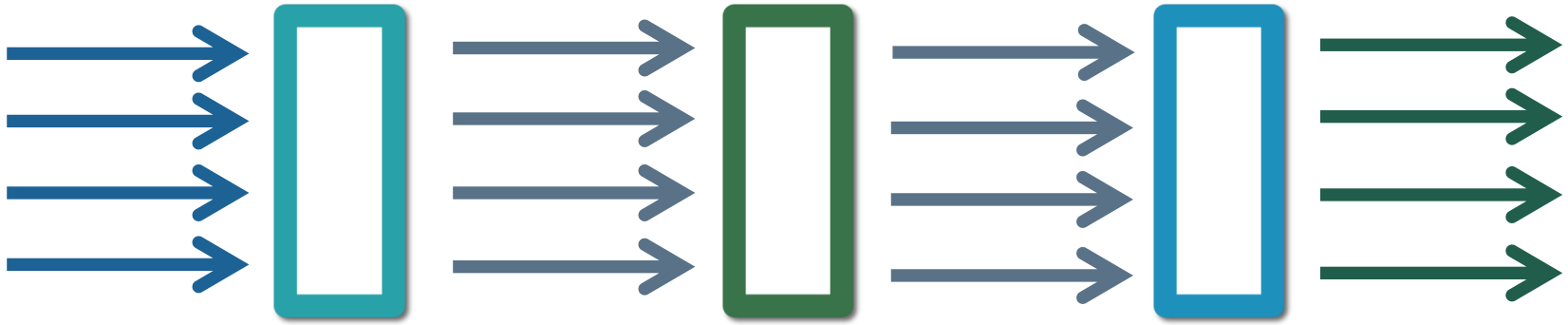
Execution time: 26,447 ms
Gen-0: 20, Gen-1: 12, Gen-2: 5


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

Lab :
Map Reduce & Producer Consumer

Lab :
Task Pipeline

Pipeline Processing



- 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

Memoization

Memoization

```
static Func<T, R> MemoizeThreadSafe<T, R>(Func<T, R> func) where T : IComparable
  {
      ConcurrentDictionary<T, R> cache = new ConcurrentDictionary<T, R>();
     return arg => cache.GetOrAdd(arg, a => func(a));
  }
```

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

Throttle Task operations

Throttling the number of concurrent Tasks

```
var listOfTasks = new List<Task>();
for (int i = 0; i < 100; i++)
{
    var count = i;
    // Note that we start the Task here too.
    listOfTasks.Add(Task.Run(() => DoSomething(count)));
}
Task.WaitAll(listOfTasks.ToArray());
```

Throttling the number of concurrent Tasks

```
async Task StartThrottledTasks(IEnumerable<Task> actions, int maxTasksToRunInParallel,
      int timeoutInMilliseconds, CancellationToken cancellationToken = new CancellationToken())
{
    var throttler = new RequestGate(maxTasksToRunInParallel);

    var tasks = from semaphore in throttler.AsyncAcquire(TimeSpan.FromMilliseconds(timeoutInMilliseconds))
                from a in actions
                select Task.Run(action).ContinueWith(tsk => semaphoreRelease.Dispose(),
                                                     TaskContinuationOptions.ExecuteSynchronously);


    await Task.WaitAll(tasks.ToArray(), cancellationToken);
}
```

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)
            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() => semaphore.Release();
    }
}
```

Throttling the number of concurrent Tasks Parallel.Invoke

```
var listOfActions = new List<Action>();
for (int i = 0; i < 100; i++)
{
    var count = i;
    // Note that we create the Action here, but do not start it.
    listOfActions.Add(() => DoSomething(count));
}

var options = new ParallelOptions {MaxDegreeOfParallelism = 3};
Parallel.Invoke(options, listOfActions.ToArray());
```

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

-- Edsger Dijkstra