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
- Leverage data and dynamic parallelism

Task and Data Parallelism

Task vs. Data Parallelism

Data Parallelism - Decompose the input into multiple chunks

- Partition the input data into multiple chunks
- Perform action on each chunk in parallel on different cores
- Merge the output results
- Can be scaled up with processors as data volume grows

Task Parallelism - Decompose the program into independent parts

- Identify the parts which can be executed in parallel
- Partition the input data into multiple chunks
- Assign each part to separate tasks
  - Perform the tasks in parallel on different cores
  - Each task is likely to perform different actions
- Perform action on each chunk in parallel on different cores
- Merge the output results (fork/join, map-reduce)

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);
}
```

Dynamic Parallelism

```
public class Tree<T>
{
        public T Data { get; set; }
        public Tree<T> Left { get; set; }
        public Tree<T> Right { get; set; }
}

static void ParallelWalk<T>(Tree<T> tree, Action<T> action)
{
        if (tree == null) return;
        var t1 = Task.Run(() => action(tree.Data));

        var t2 = Task.Run(() => ParallelWalk(tree.Left, action));
        var t3 = Task.Run(() => ParallelWalk(tree.Right, action));

        Task.WhenAll(t1, t2, t3);
}
```

Parallel While-not-Empty

```
public static void ParallelWhileNotEmpty<T>(
        IEnumerable<T> initialValues, Action<T, Action<T>> body)
{
        var from = new ConcurrentQueue<T>(initialValues);

       while (!from.IsEmpty)
        {
                var to = new ConcurrentQueue<T>();

                Action<T> addMethod = to.Enqueue;
                Parallel.ForEach(from, v => body(v, addMethod));
                from = to;
        }
}
```

Parallel While-not-Empty

```
void ParallelWhileNotEmpty<T>(IEnumerable<T> initialValues,
                              Action<T, Action<T>> body)
{
    var from = new ConcurrentQueue<T>(initialValues);
    while (!from.IsEmpty)
    {
        var to = new ConcurrentQueue<T>();
        Action<T> addMethod = to.Enqueue;
        Parallel.ForEach(from, v => body(v, addMethod));
        from = to;
    }
}
```

```
static void ParallelWalk<T>(Tree<T> tree, Action<T> action)
{
        if (tree == null) return;
        ParallelWhileNotEmpty(new[] { tree }, (item, adder) =>
        {
            if (item.Left != null) adder(item.Left);
            if (item.Right != null) adder(item.Right);
            action(item.Data);
        });
}
```

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

Oversubscription – approach

```
static void MyParallelFor(int inclusiveLowerBound, int exclusiveUpperBound, Action<int> body)
{
        int size = exclusiveUpperBound - inclusiveLowerBound;
        int numProcs = Environment.ProcessorCount;
        int range = size / numProcs;
        var countdown = new CountdownEvent(numProcs);

        for (int p = 0; p < numProcs; p++)
        {
                int start = p * range + inclusiveLowerBound;
                int end = (p == numProcs - 1) ? exclusiveUpperBound : start + range;
                Task.Run(() => {
                        for (int i = start; i < end; i++) body(i);
                        countdown.Signal();
                });
        }
        countdown.Wait();
}
```

Context Switches

Different schedulers for different OS Cooperative vs. Preemptive Threading

- ***Cooperative threading model***
  Once a thread is started it cannot be stopped or the control cannot be transferred to other threads until the thread has completed its task.

- Processes (tasks) are **run in an infinite loop**
- If task takes too much time (B), **need to divide** it
- **No overhead** occurs

- **Scheduling** is performed by the OS
- Long task is **suspended if needed**
- **Overhead**: caused by OS internals

- ***Preemptive Threading Model***
  The runtime is allowed to step in and hand control from one thread to another at any time. Higher priority threads are given precedence over Lower priority threads.

What is ‘Hill Climbing’

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);
    int pivot = left;
    for (int current = left + 1; current <= right; ++current)
        if (items[current].CompareTo(items[left]) < 0)
        {
            ++pivot;
            SwapElements(items, pivot, current);
        }
    SwapElements(items, left, pivot);
    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)
        );
}
```

Amdahl's law

The speed-up of a program using multiple processors in parallel computing is limited by the sequential fraction of the program.

For example if 95% of the program can be parallelized, the theoretical maximum speed-up using parallel computing would be 20 times, no matter how many processors are used

Task parallel programming in action

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

      threa

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


      foreach                                                   e)) {
                                                                bestMatch(Words, word);
                  matches                                       rd));

                      }
                  }

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

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);
```

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

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 : **4,192** ms

Data Parallelism with PLINQ (in functional programming)

```
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);
```

ForAll – ParallelQuery

(1)

```
var filteredItems = collection.AsParallel().Where(i =>
                                                i.SomePredicate());
// Now perform an action
foreach (var item in filteredItems)
        // These will now run serially
        item.DoSomething();
```

(3)

```
// Perform an action on the results of our filter
collection.AsParallel()
        .Where(i => i.SomePredicate())
        .ForAll(i => i.DoSomething());

// The ability to avoid the scheduling overhead is a compelling reason to use ForAll.
```

Partitioning

Partitioning in PLINQ

Range Partitioning

Chunk Partitioning

Striped Partitioning

Hash Partitioning

Partition the Workload Distribution

Partition the Workload Distribution

Partition the Workload Distribution

Partition the work

```
static void MyParallelFor(int inclusiveLowerBound, int exclusiveUpperBound, Action<int> body)
{
        int size = exclusiveUpperBound - inclusiveLowerBound;
        int numProcs = Environment.ProcessorCount;
        int range = size / numProcs;
        var tasks = new List<Task>(numProcs);

        for (int p = 0; p < numProcs; p++)
        {
                int start = p * range + inclusiveLowerBound;
                int end = (p == numProcs - 1)
                        ? exclusiveUpperBound : start + range;

                tasks.Add(Task.Run(()=>
                        for (int i = start; i < end; i++)
                                body(i) ));
        }
        Task.WhenAll(tasks);
}
```

Throttling Tasks with Partitioning

```
static Task ForEachAsync<T>(this IEnumerable<T> source, int dop, Func<T, Task> body)
{

    return Task.WhenAll(
        from partition in Partitioner.Create(source).GetPartitions(dop)

        select Task.Run(async () => {
            using (partition)

                while (partition.MoveNext())

                    await body(partition.Current);

                }));

}
```

Special Handling of Small Loop Bodies

```
int n = ...
double[] result = new double[n];
Parallel.ForEach(Partitioner.Create(0, n),
    (range) =>
    {
        for (int i = range.Item1; i < range.Item2; i++)
        {
            result[i] = (double)(i * i);
        }
    });
```

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 matc
```

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();
```

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

Task Parallel programming best 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.

Threads View – Concurrency Visualizer

Lock Contention and Serialized Execution

Uneven Workload Distribution

Partition the Workload Distribution

```
static void MyParallelFor(int inclusiveLowerBound, int exclusiveUpperBound, Action<int> body)
{
        int numProcs = Environment.ProcessorCount;
        int nextIteration = inclusiveLowerBound;
        const int batchSize = 3;
        var countdown = new CountdownEvent(numProcs);
        for (int p = 0; p < numProcs; p++)  {
                Task.Run(() => {
                        int index;
                        while ((index = Interlocked.Add(ref nextIteration, batchSize) - batchSize)
                                                                                < exclusiveUpperBound)
                        {
                                int end = index + batchSize;
                                if (end >= exclusiveUpperBound) end = exclusiveUpperBound;
                                for (int i = index; i < end; i++) body(i);
                        }
                        countdown.Signal();  });
        countdown.Wait();
```

Lock Convoys

Unfair locking is good… GC helps …

Inefficient I/O

Continuation Passing Style

Continuation-passing style (CPS) is a style of programming in which control is **passed explicitly** in the form of a continuation.

“continuation-passing style” - Wikipedia

Continuation Passing Style

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

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

Think Continuations, not Callbacks

Continuation Passing Style

```
public static int Add(int a, int b) { return a + b; }
public static int Sub(int a, int b) { return a - b; }
public static int Mul(int a, int b) { return a * b; }

public static void StandardStyle()
{
    var result1 = Add(1, 2);
    var result2 = Sub(result1, 2);
    var result3 = Mul(result2, 3);
    Console.WriteLine(result3);
}
```

Standard direct style

Use CPS

Continuation function passing to argument

```
public static void Add(int a, int b, Action<int> cont) { cont(a + b); }
public static void Sub(int a, int b, Action<int> cont) { cont(a - b); }
public static void Mul(int a, int b, Action<int> cont) { cont(a * b); }

public static void ContinuationPassingStyle()
{
    Add(1, 2, result1 =>
        Sub(result1, 2, result2 =>
            Mul(result2, 3, result3 =>
                Console.WriteLine(result3))));
}
```

Continuation implement with nested lambda expressions…

Continuation Passing Style

.NET 4.0 Task is using with ContinueWith function.

```
public static async Task<int> AddAsync(int a, int b) { return a + b; }
public static async Task<int> SubAsync(int a, int b) { return a - b; }
public static async Task<int> MulAsync(int a, int b) { return a * b; }

public static void ContinueWithStyle()
{
    AddAsync(1, 2).
        ContinueWith(task1 => SubAsync(task1.Result, 2).
            ContinueWith(task2 => MulAsync(task2.Result, 3).
                ContinueWith(task3 => Console.WriteLine(task3.Result))));
}
```

Continuation passing into Task.ContinueWith()
--> likely CPS

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

And in .NET 4.5/C# 5.0 with "async-await" style:

Using Task with async-await

```
public static async Task<int> AddAsync(int a, int b) { return a + b; }
public static async Task<int> SubAsync(int a, int b) { return a - b; }
public static async Task<int> MulAsync(int a, int b) { return a * b; }

public static async Task AwaitableStyleStyle()
{
    var result1 = await AddAsync(1, 2);
    var result2 = await SubAsync(result1, 2);
    var result3 = await MulAsync(result2, 3);
    Console.WriteLine(result3);
}
```

That means "Continuation" OK?

Continuation Passing Style

In .NET 4.0 (C# 4.0 Task only)

```
public static async Task<int> AddAsync(int a, int b) { return a + b; }
public static async Task<int> SubAsync(int a, int b) { return a - b; }
public static async Task<int> MulAsync(int a, int b) { return a * b; }

public static void ContinueWithStyle()
{
    AddAsync(1, 2).
        ContinueWith(task1 => SubAsync(task1.Result, 2).
            ContinueWith(task2 => MulAsync(task2.Result, 3).
                ContinueWith(task3 => Console.WriteLine(task3.Result))));
}
```

And .NET 4.5 (C# 5.0 async-await)

```
public static async Task<int> AddAsync(int a, int b) { return a + b; }
public static async Task<int> SubAsync(int a, int b) { return a - b; }
public static async Task<int> MulAsync(int a, int b) { return a * b; }

public static async Task AwaitableStyleStyle()
{
    var result1 = await AddAsync(1, 2);
    var result2 = await SubAsync(result1, 2);
    var result3 = await MulAsync(result2, 3);
    Console.WriteLine(result3);
}
```

Continuation Passing Style (F# Async Workflow)

```
let asyncAdd x y = async { return x + y }
let asyncSub x y = async { return x - y }
let asyncMul x y = async { return x * y }

let asyncWorkflowSample = async {
  let! result1 = asyncAdd 1 2
  let! result2 = asyncSub result1 2
  let! result3 = asyncMul result2 3
  do Console.WriteLine result3
}
```

Returned Async<'T> type

Structure nearly equals **"async-await."**
That's CPS?

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;
```

New Project
Installed
Templates
Visual C#
Visual C++
SQL Server
Visual F#
.NET Framework 4.5
Sort by:
Default
Search Installed Te
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

Mathematical patterns for better composition

Task is a Monad

Monad is a design pattern!
Monads are not a feature of the language, but they are a pattern that can be used in any language

Function Composition

Function Composition - Building with composition. Composition is the 'glue' that allows us build larger systems from smaller ones. This is the very heart of the functional style. Almost every line of code is a composable expression. Composition is used to build basic functions, and then functions that use those functions, and so on.

Composition — in the form of passed parameters plus first-class functions

How can we compose Tasks?

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

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

C#

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

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

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

C#

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))

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>
```

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) …
```

IEnumerable 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 )
```

Task SelectMany

```
// M<T> * (T -> M<U>) -> M<U>
Task<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: Tasks express in declarative way LINQ-ish style

parallel face detection

Task Inline - ExecuteSynchronously

```
public static Task<TOut> Then<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;
}
```

Composing Task

```
static member Then(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

```
// '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
```

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;
```

Composing Task async/await

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


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


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

Composing Task async/await

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

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

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

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

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

```
static Func<T, IObservable<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
Installed
Templates
Visual C#
Visual C++
SQL Server
Visual F#
.NET Framework 4.5
Sort by:
Default
Search Installed Te
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

In reality…

the Task<T> is a Comonad

Task is a comonads

	Sync	Async
Monad	IEnumerable<T>	IObservable<T>
Comonad	Lazy<T>	Task<T>

	Bind	Unit
IEnumerable<T>	SelectMany; a.k.a., flatMap	Constructors of Array, List<T>, etc.
IObservable<T>	SelectMany; a.k.a., flatMap	Observable.Return; a.k.a., just

Task is a comonads

	Bind	Unit
IEnumerable<T>	SelectMany; a.k.a., flatMap	Constructors of Array, List<T>, etc.
IObservable<T>	SelectMany; a.k.a., flatMap	Observable.Return; a.k.a., just

	Cobind	Extract
Task<T>	ContinueWith; a.k.a., then	get_Result (a property named Result)

What could be another command in .NET ?

Convert Comonad <> Monad

```
IObservable<Unit> observable1 = Task.Run(() => Console.WriteLine("Working")).ToObservable();

IObservable<string> observable2 = Task<string>.Run(() => "Working").ToObservable();


Task task1 = observable1.ToTask();

Task<string> task2 = observable2.ToTask();
```

Parallel Reducer
and
Map-Reduce

Parallel Aggregation

```
int sum = 0;
for (int i = 0; i < 100; i++)
{
      if (isPrime(i)) sum += i;
}
```

```
int sum = 0;
Parallel.For(0, 100, delegate (int i)
{
   if (isPrime(i))
       lock (this) { sum += i; }
});
```

Parallel Aggregation

```
int sum = Parallel.Aggregate(
            0, 100, // domain
            0, // initial value
            Action<int>(i => {
                  return (isPrime(i) ? i : 0)
            },
            Action<int,int>(x, y => { return x + y; }
           );
```

it is required that the combining delegate function is **associative** and **commutative**

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) => {
                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);
```

Reducer

```
static TResult Reduce<TValue, TResult>(this IEnumerable<TValue> source, Func<TValue, TResult> selector,
                  Func<TResult, TResult, TResult> reducer, CancellationToken token = new CancellationToken())
{
    var partitioner = Partitioner.Create(source, EnumerablePartitionerOptions.None);
    var pos = new ParallelOptions { CancellationToken = token, MaxDegreeOfParallelism = Environment.ProcessorCount }
    var results = new Atom<ImmutableList<TResult>>(ImmutableList<TResult>.Empty);

    Parallel.ForEach(partitioner,
        pos,                        // Parallel options
        () => new List<TResult>(),          // Thread-Local initialization
        (item, loopState, local) =>  {      // body of analysis logic
            local.Add(selector(item));
            return local;
        },
        threadlocalFinal => results.Swap(o => o.AddRange(threadlocalFinal)));   // merge logic
    return results.Value.AsParallel().Aggregate(reducer);
}
```

Avoid nested parallel loop?

```
Parallel.For(0, size, delegate (int i) {
        Parallel.For(0, size; delegate (int j) {
                result[i, j] = 0;
                for (int k = 0; k < size; k++)
                {
                        result[i, j] += m1[i, k] * m2[k, j];
                }
        });
```

Avoid nested parallel loop
Avoid Over-Parallelization

```
Parallel.For(0, size, delegate (int i) {
Parallel.For(0, size; delegate (int j) {
    result[i, j] = 0;
    for (int k = 0; k < size; k++)
    {
        result[i, j] += m1[i, k] * m2[k, j];
    }
});
```

```
static void For2(n1, n2, Action<int, int> body)
 {
     Parallel.For(0, n1 * n2, delegate (int i) {
         body(i / n2, i % n2);
     });
 }


Parallel.For2(size, size, delegate (int i, int j) {
     result[i, j] = 0;
     for (int k = 0; k < size; k++)
     {
         result[i, j] += m1[i, k] * m2[k, j];
     }
 });
```

Lab :
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, [])

Monid 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
```

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
```

Map-Reduce in action

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.RemoveEmptyEntri
    .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), //#1
        (localDic, line) => { //#2
            foreach (var word in line.Split(Separators, StringSplitOptions.RemoveEmptyEntries)
                                    .Where(IsValidWord))
                TrackWordsOccurrence(localDic, word);
            return localDic;
        },
        (finalResult, localDic) => { // #3
            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 //#4
            .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

Pipeline pattern

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

Parallel Pipeline

```
int BufferSize = ...
var buffer1 = new BlockingCollection<string>(BufferSize);
var buffer2 = new BlockingCollection<string>(BufferSize);
var buffer3 = new BlockingCollection<string>(BufferSize);
var f = new TaskFactory(TaskCreationOptions.LongRunning);

var stage1 = f.StartNew(() => ReadStrings(buffer1, ...));
var stage2 = f.StartNew(() => CorrectCase(buffer1, buffer2));
var stage3 = f.StartNew(() => CreateSentences(buffer2, buffer3));
var stage4 = f.StartNew(() => WriteSentences(buffer3));
Task.WaitAll(stage1, stage2, stage3, stage4);
```

Performance characteristic

Pipeline multiplexer

```
 while (!_input.All(bc => bc.IsCompleted) && !_token.IsCancellationRequested)
{
    var i = BlockingCollection<TInput>.TryTakeFromAny(_input, out var receivedItem, 50, _token);
    if (i >= 0)
    {
        TOutput outputItem =
           _processor != null ? _processor(receivedItem) : await _processoTask(receivedItem);
        BlockingCollection<TOutput>.AddToAny(Output, outputItem);
        sw.SpinOnce();
     }
     else
     {
          Thread.SpinWait(1000);
     }
}
```

Traditional Parallel Pipeline

Functional Parallel Pipeline

Cancelling Pipeline

```
static void DoPipeline(CancellationToken token)
{
    using (CancellationTokenSource cts =
            CancellationTokenSource.CreateLinkedTokenSource(token))
    {
        var f = new TaskFactory(TaskCreationOptions.LongRunning,
                                TaskContinuationOptions.None);
        var stage1 = f.StartNew(() => DoStage1(..., cts));
        var stage2 = f.StartNew(() => DoStage2(..., cts));
        var stage3 = f.StartNew(() => DoStage3(..., cts));
        var stage4 = f.StartNew(() => DoStage4(..., cts));
    }
    Task.WaitAll(stage1, stage2, stage3, stage4);
}
```

Lab : pipeline processing

Building Tasks Combinators

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;
  }
```

Memoization

```
static Func<T, Task<R>> MemoizeLazyTaskSafe<T, R>(Func<T, Task<R>> func)
{
     ConcurrentDictionary<T, Lazy<Task<R>>> cache =
                    new ConcurrentDictionary<T, Lazy<Task<R>>>();

     return arg => cache.GetOrAdd(arg, a => new Lazy<Task<R>>(() =>
                                                          func(a))).Value;
}
```

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 ThrottledTasks(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)
            Thread.BeginCrtiticalRegion();
            return new SemaphoreSlimRelease(semaphore);
        throw new Exception("couldn't acquire a semaphore");
    }
    private class SemaphoreSlimRelease : IDisposable
    {
        SemaphoreSlim semaphore;
        public SemaphoreSlimRelease(SemaphoreSlim semaphore) => this.semaphore = semaphore;
        public void Dispose() {
                Thread.EndCrtiticalRegion();
                semaphore.Release();
    }
}
```

The RequestGate

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

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

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());
```

Task.WhenAll – rendezvous point

```
IEnumerable<Task> asyncOps = from addr in addrs
                             select SendMailAsync(addr);
await Task.WhenAll(asyncOps);
```

Task.WhenAny – fine control

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

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

Throttling: Allowing additional operations to begin as others complete

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

Task.WhenAny - Redundancy

```
var cts = new CancellationTokenSource();

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

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

Task.WhenAny - Interleaving

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

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

Task.WhenAny – Early Bailout

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

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

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

else

       delayCts.Cancel();
```

Process Tasks as they complete

Handle Tasks as they complete

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

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

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

     stockHistoryTasks.Remove(stockHistoryTask);

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

    await ProcessStock(stockHistory);
}
```

Throttling Tasks as they complete

```
const int CONCURRENCY_LEVEL = 15;
Uri[] urls = ...;
int nextIndex = 0;
var imageTasks = new List<Task<Bitmap>>();

while (nextIndex < CONCURRENCY_LEVEL && nextIndex < urls.Length)
{
    imageTasks.Add(GetBitmapAsync(urls[nextIndex]));
    nextIndex++;
}
while (imageTasks.Count > 0)
{
    Task<Bitmap> imageTask = await Task.WhenAny(imageTasks); imageTasks.Remove(imageTask);
    Bitmap image = await imageTask;
    imageTasks.Remove(imageTask);
    ProcessImage(image);
    if (nextIndex < urls.Length)
        imageTasks.Add(GetBitmapAsync(urls[nextIndex])); nextIndex++;
}
```

Performance overheads with Task.WhenAny

```
public static Task<Task<T>>[] Interleaved<T>(IEnumerable<Task<T>> tasks)
{
    var inputTasks = tasks.ToList();
    var buckets = new TaskCompletionSource<Task<T>>[inputTasks.Count];
    var results = new Task<Task<T>>[buckets.Length];
    for (int i = 0; i < buckets.Length; i++)
    {
        buckets[i] = new TaskCompletionSource<Task<T>>();
        results[i] = buckets[i].Task;
    }
    int nextTaskIndex = -1;
    Action<Task<T>> continuation = completed =>  {
        var bucket = buckets[Interlocked.Increment(ref nextTaskIndex)];
        bucket.TrySetResult(completed);
    };

    foreach (var inputTask in inputTasks)
        inputTask.ContinueWith(continuation, TaskContinuationOptions.ExecuteSynchronously, TaskScheduler.Default);
    return results;
}
```

Performance overheads with Task.WhenAny

```
public static Task<Task<T>>[] Interleaved<T>(IEnumerable<Task<T>> tasks)
{
    var inputTasks = tasks.ToList();
    var buckets = new TaskCompletionSource<Task<T>>[inputTasks.Count];
    var results = new Task<Task<T>>[buckets.Length];
    for (int i = 0; i < buckets.Length; i++)
    {
        buckets[i] = new TaskCompletionSource<Task<T>>();
        results[i] = buckets[i].Task;
    }
    int nextTaskIndex = -1;
    Action<Task<T>> continuation = completed =>  {
        var bucket = buckets[Interlocked.Increment(ref nextTaskIndex)];
        bucket.TrySetResult(completed);
    };

    foreach (var inputTask in inputTasks)
        inputTask.ContinueWith(continuation, TaskContinuationOptions.ExecuteSynchronously, TaskScheduler.Default);
    return results;
}
```

```
foreach(var bucket in Interleaved(tasks))
{
    var t = await bucket;
    Process(await t);
}
```

Performance overheads with Task.WhenAny & Task.WhenAll

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


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

Producer Consumer again

Simple produce consumer

```
BlockingCollection<int> bCollection = new BlockingCollection<int>(10);

Task producerThread = Task.Factory.StartNew(() =>
{
    for (int i = 0; i < 10; ++i)
        bCollection.Add(i);

    bCollection.CompleteAdding();
});


Task consumerThread = Task.Factory.StartNew(() =>
{
    while (!bCollection.IsCompleted)
    {
        int item = bCollection.Take();
        Console.WriteLine(item);
    }
});


Task.WaitAll(producerThread, consumerThread);
```

Enable Multi-Producers/Multi-Consumers

```
BlockingCollection<int>[] sourceArrays = new BlockingCollection<int>[5];
for(int i = 0; i < sourceArrays.Length; i++)
        sourceArrays[i] = new BlockingCollection<int>(500);

Parallel.For(0, sourceArrays.Length * 500, (j) => {
        int k = BlockingCollection<int>.TryAddToAny(sourceArrays, j);
        if(k >=0) // do something here
});

int i = BlockingCollection<TInput>.TryTakeFromAny(m_input, out receivedItem);
if ( i >= 0) {
         // do something here
}
```

LAB: Direct Acyclic Graph of Tasks

A graph data structure consists of two basic elements:

- **Vertex** – A single node in the graph
- **Edge** – Connects one or two vertices. Can contain a value

Direct Acyclic Graph of Tasks

```
var f = Task.Factory;
      var build1 = f.StartNew(() => Build(project1));
      var build2 = f.StartNew(() => Build(project2));

   var build3 = f.StartNew(() => Build(project3));
   var build4 = f.ContinueWhenAll(new[] { build1 },
       _ => Build(project4));
   var build5 = f.ContinueWhenAll(new[] { build1, build2, build3 },
       _ => Build(project5));
   var build6 = f.ContinueWhenAll(new[] { build3, build4 },
       _ => Build(project6));
   var build7 = f.ContinueWhenAll(new[] { build5, build6 },
       _ => Build(project7));
   var build8 = f.ContinueWhenAll(new[] { build5 },
       _ => Build(project8));
   Task.WaitAll(build1, build2, build3, build4, build5, build6, build7, build8);
```

Direct Acyclic Graph of Tasks

Direct Acyclic Graph of Tasks

```
let dagAsync = Async.DAG.ParallelTasksDAG()
dagAsync.OnTaskCompleted |>
        Observable.add(fun op -> System.Console.ForegroundColor printfn "Completed %d" op.Id)

dagAsync.AddTask(1, acc1, 4,5)
dagAsync.AddTask(2, acc2, 5)
dagAsync.AddTask(3, acc3, 6, 5)
dagAsync.AddTask(4, acc4, 6)
dagAsync.AddTask(5, acc5, 7, 8)
dagAsync.AddTask(6, acc6, 7)
dagAsync.AddTask(7, acc7)
dagAsync.AddTask(8, acc8)
dagAsync.ExecuteTasks()
```

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

-- Edsger Dijkstra