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;
     });
}
```

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
F O R K
map
map
map
map
map
map
reduce
reduce
reduce
reduce
reduce
reduce
J O I N
Aggregate Data

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
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

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());
```