Concurrency Patterns

Software Design

version: 1.0.2

AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

The menu

- Aggregation
- MapReduce

Aggregation

Concurrency Patterns

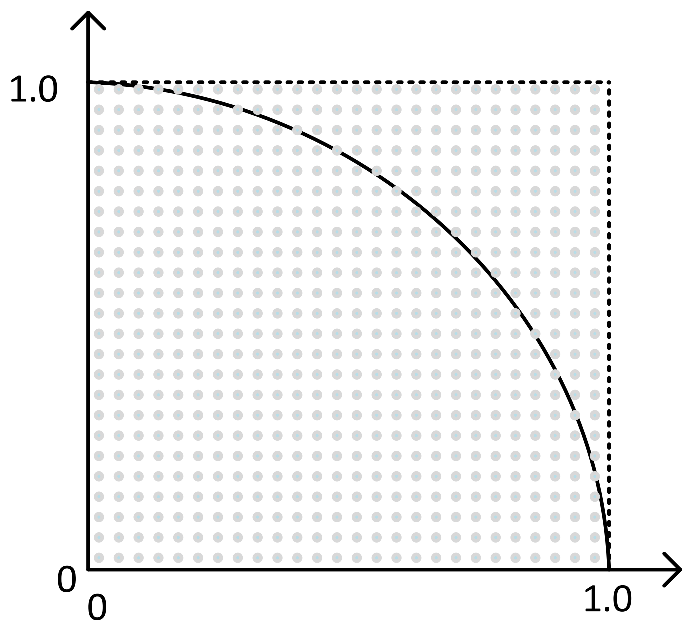
AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

Aggregation

- *Aggregation* is the action of collecting items to form a total quantity.
  - News aggregators
  - The class relationship in UML
  - …
- In the scope of concurrent programming, aggregation is the collection of sub-results to one total result.
  - Think divide-and-combine

Aggregation strategies

- Let's assume we wish to approximate *pi* = 3.14159… using the "Dartboard" algorithm.



$$\pi \cong \frac{4 \cdot n_{circle}}{n}$$

Calculating PI with darts

- https://www.youtube.com/watch?v=M34TO71SKGk

Aggregation strategies

- Sequential/serial estimation of pi:

```
private static double SerialEstimationOfPi()
{
  double nInside = 0;
  double stepSize = 1/(double)_nDarts; // nDarts – division in x and y
  for (int i = 0; i < _nDarts; i++)
  {
    var x = i * stepSize;
    for (int j = 0; j < _nDarts; j++)
    {
      var y = j * stepSize;
      if (Math.Sqrt(x*x + y*y) < 1.0) ++nInside;
    }
  }

  return 4 * nInside/(_nDarts*_nDarts);
}
```

$$\pi \cong \frac{4 \cdot n_{circle}}{n}$$

Here n_{circle} is nInside
and n is _nDarts*_nDarts

- Where is the aggregation?
- Can we parallelize this calculation? How?

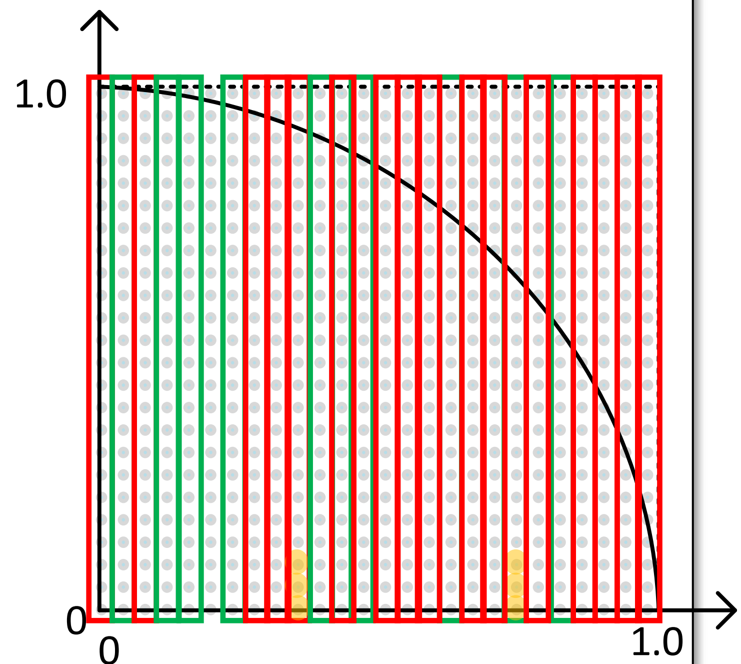
Aggregation strategies

- Parallel estimation of pi – 1st attempt

```
private static double ParallelEstimationOfPi()
{
    var locker = new object();

    double nInside = 0;
    double stepSize = 1 / (double)_nDarts;

    // 1 iteration = 1 ”strip” of darts
    Parallel.For(0, _nDarts, i =>
    {
        var x = i * stepSize;
        for (int j = 0; j < _nDarts; j++)
        {
            var y = j*stepSize;
            if (Math.Sqrt(x*x + y*y) < 1.0)
                lock(locker) ++nInside;
        }
    }
    );
    return 4 * nInside / (_nDarts * _nDarts);
}
```



Thread 1
Thread 2
…
lock

Aggregation strategies

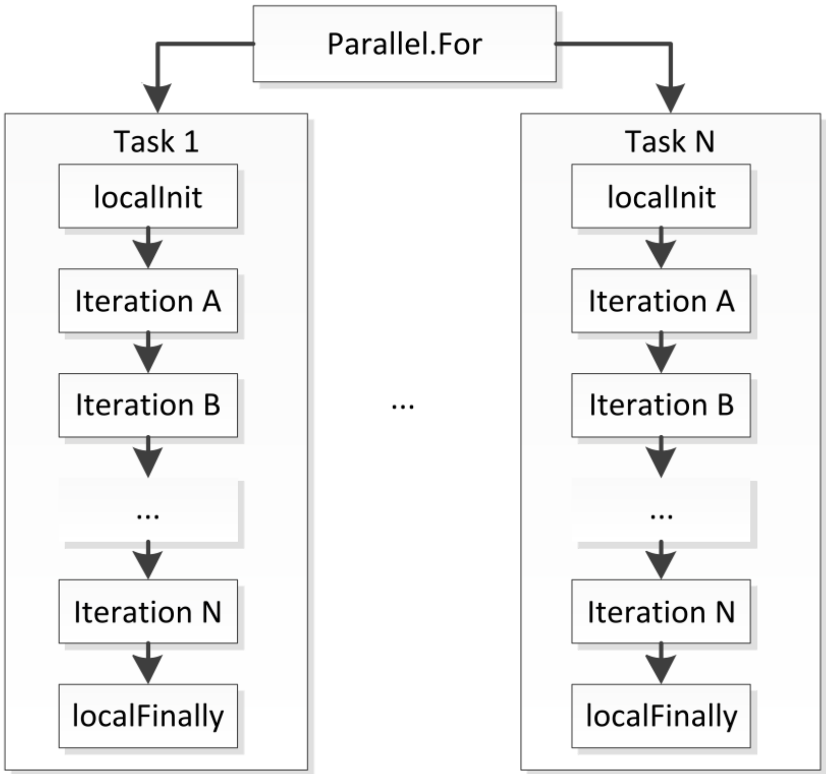
- Observe: ”Parallel” iterations running on *same* underlying thread will never content for the lock – why not?

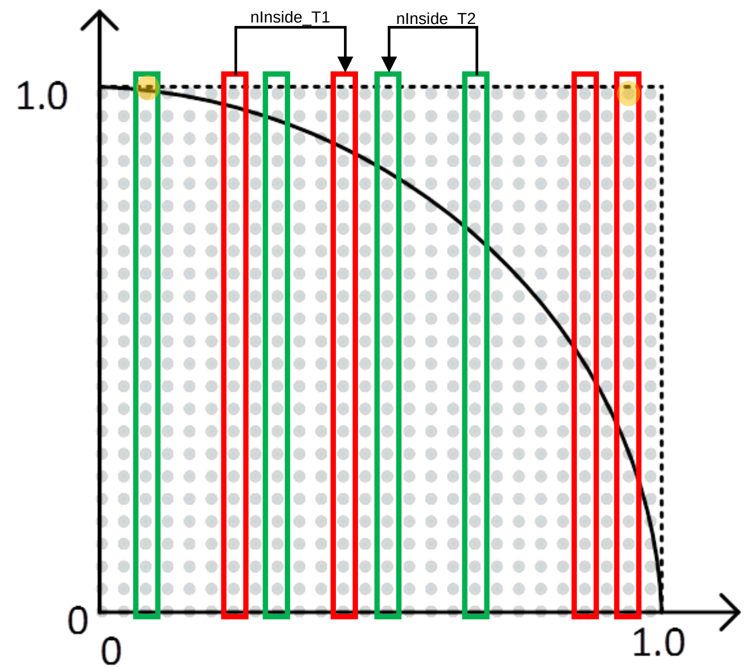
- Thus, parallel iterations running on *same* thread can be implemented as if they were serial – how does that help?

- Special `Parallel.For()` overload supports this ☺

```
public static ParallelLoopResult For<TLocal>(
    int fromInclusive, int toExclusive,
    Func<TLocal> localInit,
    Func<int, ParallelLoopState, TLocal, TLocal> body,
    Action<TLocal> localFinally);
```

Aggregation strategies






```
public static ParallelLoopResult For<TLocal>(
    int fromInclusive, int toExclusive,
    Func<TLocal> localInit,
    Func<int, ParallelLoopState, TLocal, TLocal> body,
    Action<TLocal> localFinally);
```

Aggregation strategies

- Parallel estimation of pi – 2nd attempt

```
private static double ParallelEstimationOfPi()
{
    var locker = new object();
    double nInsideCircle = 0;
    double stepSize = 1 / (double)_nDarts;
    Parallel.For(0, _nDarts,
        () => 0, // localInit: Initialize nInside (passed to first iteration)
        (i, dummyState, nInside) =>
        {
            var x = i * stepSize;
            for (int j = 0; j < _nDarts; j++)
            {
                var y = j * stepSize;
                if (Math.Sqrt(x * x + y * y) < 1.0) ++nInside;
            }
            return nInside; // Handed over to next task executing on thread
        },

        // LocalFinally: lock and aggregate local result to global result
        inside => { lock (locker) nInsideCircle += inside; });

    return 4 * nInsideCircle / (_nDarts * _nDarts);
}
```

```
public static ParallelLoopResult For<TLocal>(
    int fromInclusive, int toExclusive,
    Func<TLocal> localInit,
    Func<int, ParallelLoopState, TLocal, TLocal> body,
    Action<TLocal> localFinally);
```

Requires no locking – runs in same thread

"Final" operations – requires locking

Aggregation strategies

- Observe: While more efficient, the work done in the delegate is still very limited, and partitioning of the work is ”messy”

- We can use an similarly overloaded `Parallel.ForEach()` with a *partitioner* to create ”optimal chunks” of work for each task

Aggregation strategies

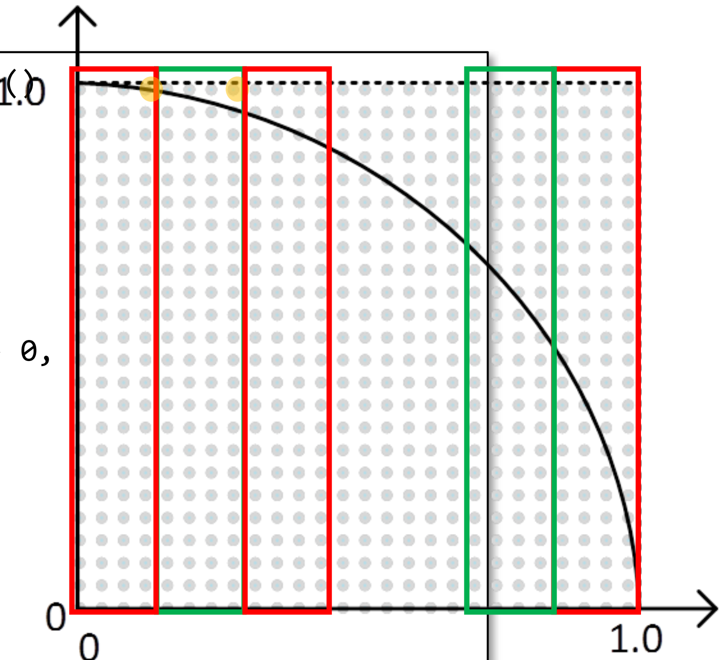
- Parallel estimation of pi – 3rd attempt

```
private static double ParallelEstimationOfiWithPartitioner()
{
    var locker = new object();

    double nInsideCircle = 0;
    double stepSize = 1 / (double)_nDarts;

    Parallel.ForEach(Partitioner.Create(0, _nDarts), () => 0,
        (range, state, inside) =>
    {
        for (int i = range.Item1; i < range.Item2; i++)
        {
            var x = i * stepSize;
            for (int j = 0; j < _nDarts; j++)
            {
                var y = j*stepSize;
                if (Math.Sqrt(x*x + y*y) < 1.0)  ++ inside;
            }
        }
        return inside;
    },
    inside => { lock (locker) nInsideCircle += inside; })

    return 4 * nInsideCircle / (_nDarts * _nDarts);
}
```



range tuple contains start/end of this partition's calculations

MapReduce

Concurrency Patterns

A gentle introduction: http://ksat.me/map-reduce-a-really-simple-introduction-kloudo/

MapReduce

- Often, we require "simple" answers to questions that require investigation of large data sets - routinely petabytes (10^{15} bytes).
  - 1 PB = 1.5 km high stack of CD-ROMs!
- *MapReduce* is a pattern that allows parallel computations on such data sets

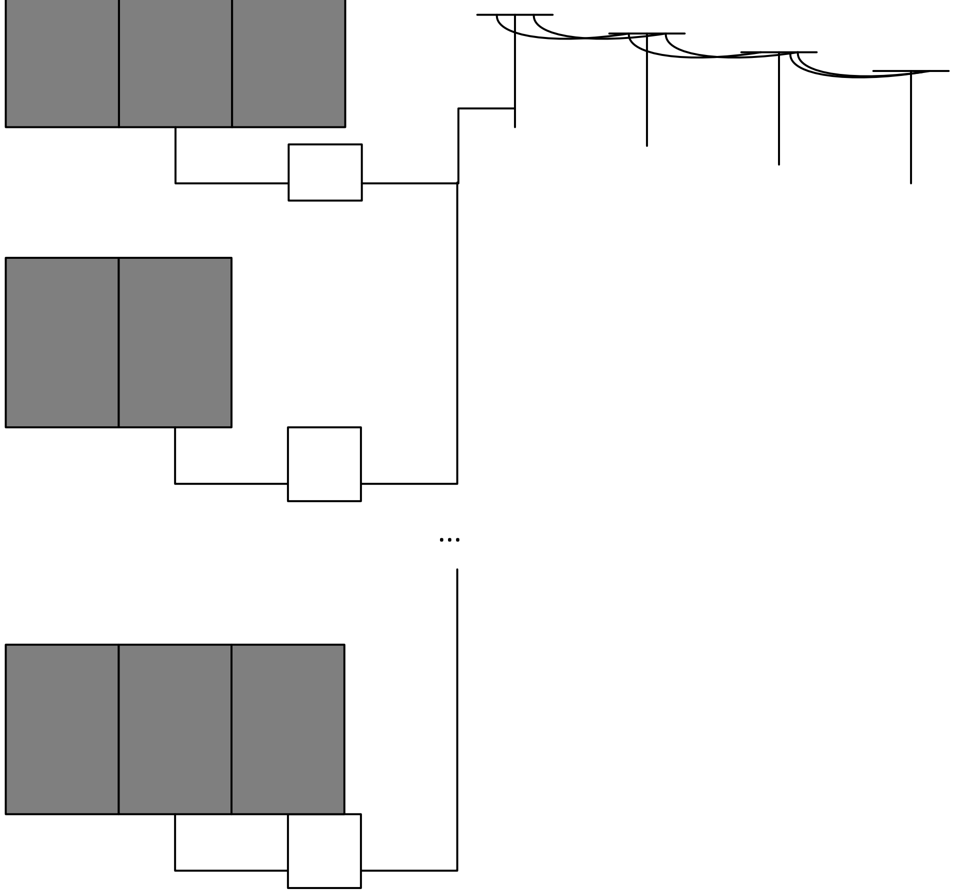
> MapReduce takes a set of input key/value pairs, and produces a set of output key/value pairs. The user of the *MapReduce* library expresses the computation as two functions: *Map* and *Reduce*.

- The key to the strategy is to parallelize data processing on many *nodes* to allow speed-up and thus answer queries quickly

MapReduce – four steps

- The four steps in MapReduce
  1. *Distribute* partitions and distributes source data to different nodes to allow parallel work
  2. *Map* transforms source data representation on each node to (a large number of simple) intermediate key-value pairs
  3. *Group* groups the intermediate key-value by keys for ease of reduction (a "group-by"-operation)
  4. *Reduce* merges/aggregates/interprets reduced data into an answer to the original query
- *Distribute* and *Group* _do not_ vary – usually provided by a framework
- *Map* and *Reduce* vary – provided by user

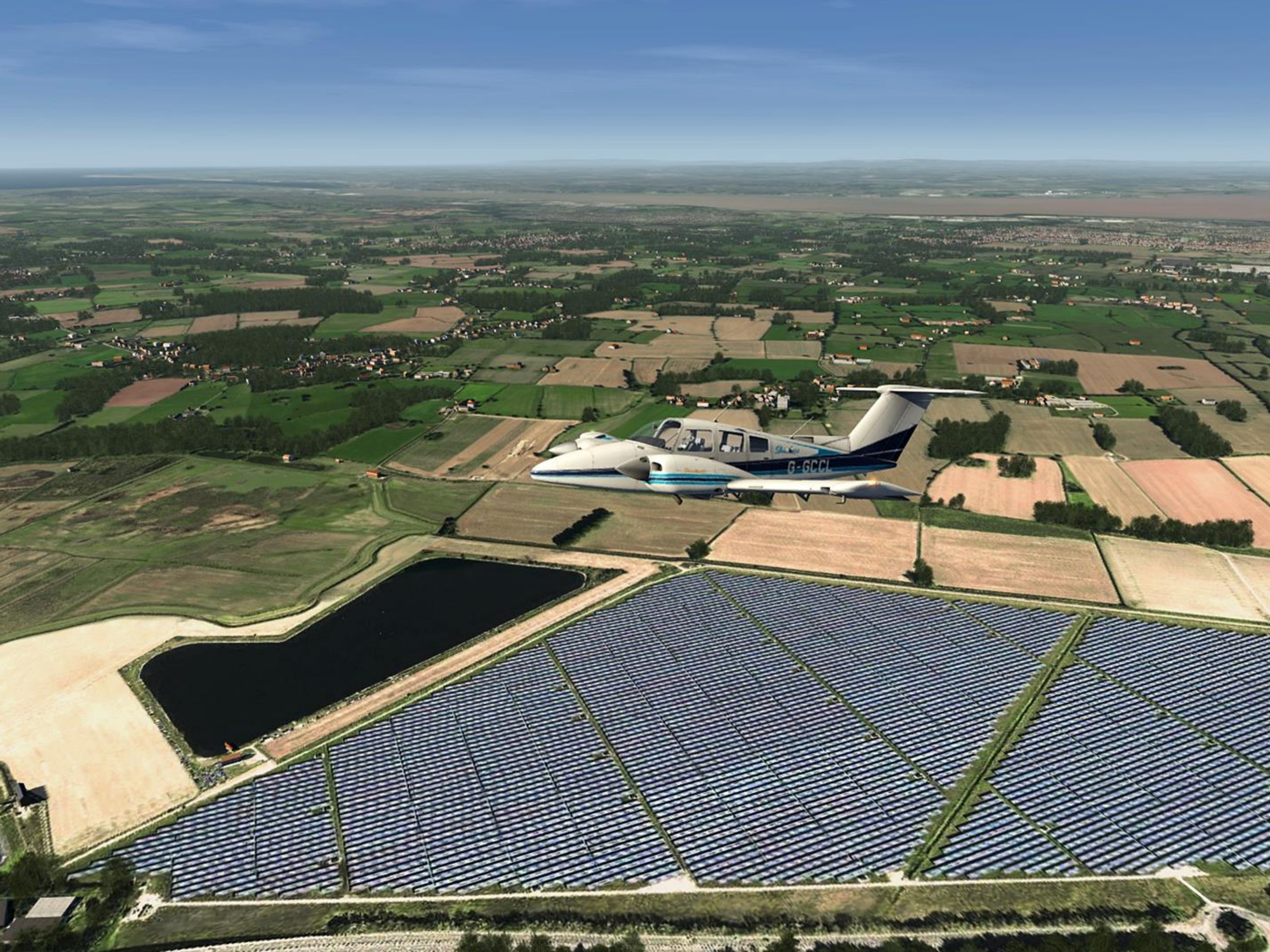
Example: Solar panels



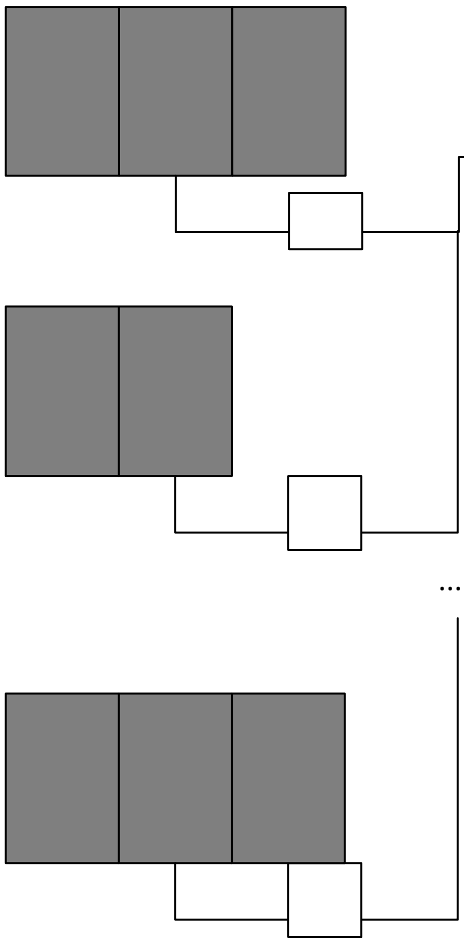
Log file (1 panel)

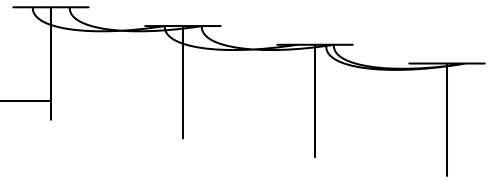
```
5/9-14 12:00:00: 75W
5/9-14 12:00:10: 79W
5/9-14 12:00:20: 81W
…
…
```

3.153.600 data points per panel per year


G-GCCL

Example: Solar panels





Log file (1 panel)

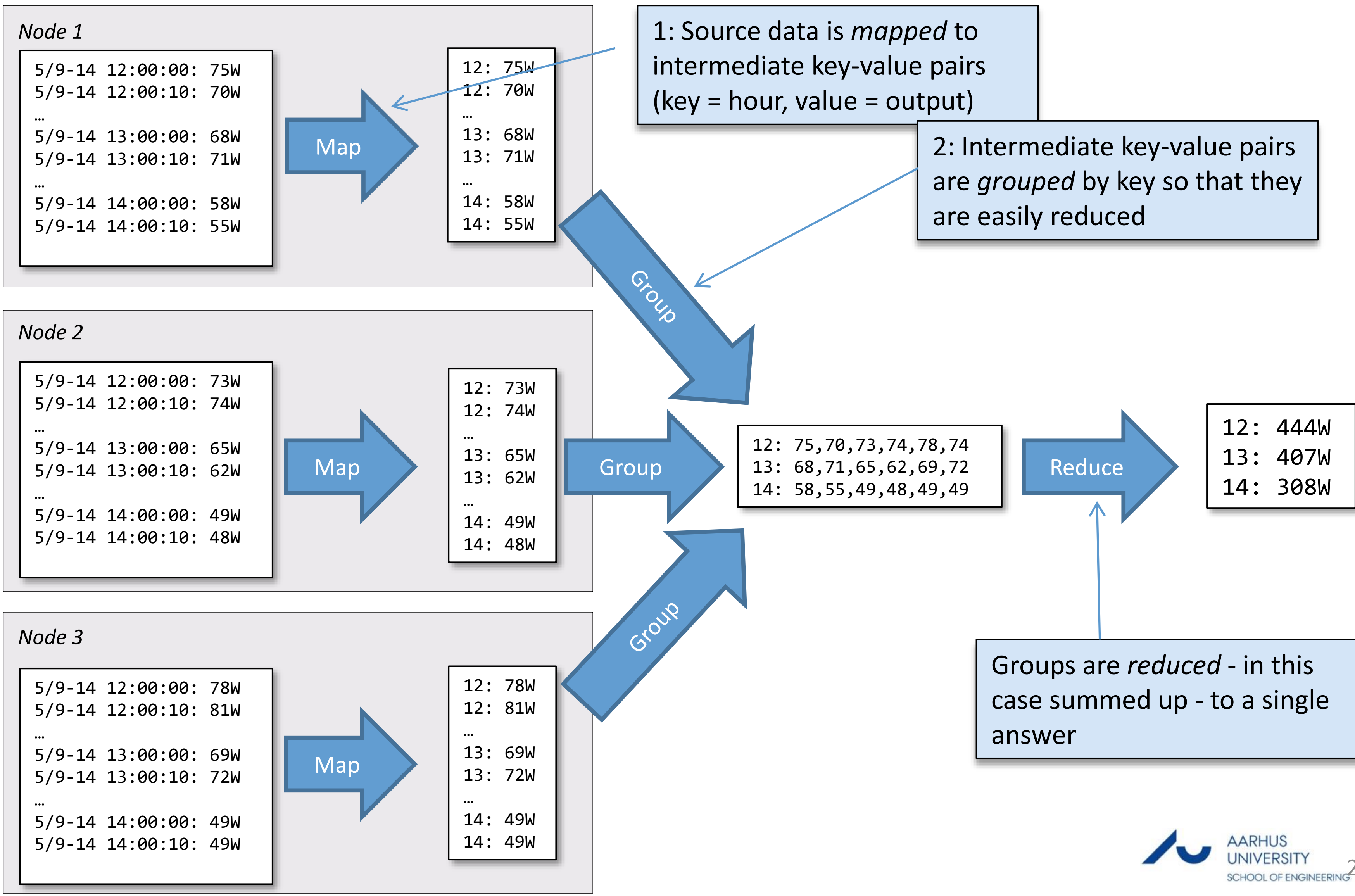
```
5/9-14 12:00:00: 75W
5/9-14 12:00:10: 79W
5/9-14 12:00:20: 81W
…
…
```

3.153.600 data points per panel per year

Query:
”What is the total per-hour production from all panels?”

Answer form:
”…
From 12:00 to 13:00, 745MW is produced.
From 13:00 to 14:00, 812MW is produced.
…”

Example: Solar panel



Our “own” MapReduce( ) implementation using PLINQ

```
// Patterns of Parallel Programming p. 75
public ParallelQuery<TResult> MapReduce<TSource, TMapped, TKey, TResult>(
    this ParallelQuery<TSource> source,
    Func<TSource, IEnumerable<TMapped>> map,
    Func<TMapped, TKey> keySelector,
    Func<IGrouping<TKey, TMapped>, IEnumerable<TResult>> reduce)
{
  return source
    .SelectMany(map)
    .GroupBy(keySelector)
    .SelectMany(reduce);
}
```

C# Extensions methods

- Static methods called used instance method syntax

```
namespace ExtensionMethods
{
    public static class MyExtensions
    {
        public static int WordCount(this String str)
        {
            return str.Split(new char[] { ' ', '.', '?' },
                             StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}
```

```
using ExtensionMethods;

….
           string s = "Hello Extension Methods";
           int i = s.WordCount();
```

- In IL the compiler translate ‘WordCount’ call to a static method call.
- Extensions methods cannot access private var.

MapReduce() implementation – the word-count-by-length example

1. *Distribute*: Read the books into memory, separate them word-by-word
```
“The fox and the hound are mortal fiends” →
[“The”, “fox”, “and”, “the”, “hound”, “are”, “mortal”, “fiends”]
```

2. *Map*: Map each word to key-value pair, where key = length of word
```
“The”   →   [3: “The”],
“fox”   →   [3: “fox”],
“hound” →   [5, “hound”],
…
```

3. *Group*: Let `GroupBy()` create groups of key-value pairs with same key:
```
[3: “The”], [3: “fox”], [3: “and”], [3: “the”], [5, “hound”], … →
[3: [“The”, “fox”, “and”, “the”, “are”]],
[5: [“hound”]],
[6: [“mortal”, “fiends”]]
```

1. *Reduce*: Reduce the number of words in each grouping to a count
```
[3: 5],
[5: 1],
[6: 2]
```

MapReduce() invocation – word-count-by-length example

```
static void Main(string[] args)
{
    var files = Directory.EnumerateFiles(@"C:\(…)\Books", "*.txt").AsParallel();

    var wordCounts = files.MapReduce(
        path => Map(path),
        map => ExtractKey(map),
        group => Reduce(group));


    foreach (var pair in wordCounts)
    {
        Console.WriteLine("{0}: {1}",
          pair.Key, pair.Value);
    }
}
```

Map() transforms files into w

ExtractKey() returns the key for a word (its length)

Reduce() performs calculations on sets of data that belong to the same key

```
// Patterns of Parallel Programming p. 75
public ParallelQuery<TResult> MapReduce<TSource, TMapped, TKey, TResult>(
    this ParallelQuery<TSource> source,
    Func<TSource, IEnumerable<TMapped>> map,
    Func<TMapped, TKey> keySelector,
    Func<IGrouping<TKey, TMapped>, IEnumerable<TResult>> reduce)
{
  return source
    .SelectMany(map)
    .GroupBy(keySelector)
    .SelectMany(reduce);
}
```

MapReduce() invocation – word-count-by-length example

1. *Map:* Read the books into memory, separate them word-by-word
   “The fox and the hound are mortal fiends” →
   [“The”, “fox”, “and”, “the”, “hound”, “are”, “mortal”, “fiends”]

```
// Map() provides the source data on which the MapReduce query shall run.
static IEnumerable<string> Map(string path)
{
    return File.ReadLines(path) // Read all lines in the path
        .SelectMany(line => line.ToLower().Split(new char[] { ' ', ',', '.', '-', '!', '?', ';' }
    // Project the words into a single enumerable
}
```

2. *keySelector*: Each word’s key is length.
   “The” → 3,
   “fox” → 3,
   “hound” → 5
   …

```
// ExtractKey() returns the key which the word fits
static int ExtractKey(string word)
{
    return word.Length;
}
```

MapReduce() invocation – word-count-by-length example

3. *Group:* Let `GroupBy()` create groups of key-value pairs with same key:
   `[3: “The”], [3: “fox”], [3: “and”], [3: “the”], [5, “hound”], … →`
   `[3: [“The”, “fox”, “and”, “the”, “are”]]`
   `[5: [“hound”]]`
   `[6: [“mortal”, “fiends”]]`

4. *Reduce:* Reduce the number of words in each grouping to a count
   `[3: 5],`
   `[5: 1],`
   `[6: 2]`

```
// Reduce() returns a list of key/value pairs representing the results
// Note: IGrouping<> represents a set of values that have the same key
// e.g. [int: [str1, str2, str3, …, strn]]
static IEnumerable<KeyValuePair<int, int>> Reduce(IGrouping<int, string> group)
{
    return new KeyValuePair<int, int>[]
    {
        new KeyValuePair<int, int>(group.Key, group.Count())
    };
}
```

Your turn

Solve the exercises