6. Parallel Aggregation + MapReduce

1. Redegør for hvad et design pattern er
2. Redegør for strukturen af Parallel Aggregation
    a. Handling der samler data til større helhed
    b. Parallel programmering -> aggregation der samler subresultater til et samlet resultat
    c. Tidskrævende arbejde kan deles op i mindre bidder
        i. Hver bid bearbejdes af tråde
        ii. De enkelte bidder er ikke afhængige af hinanden
        iii. Samles igen til et samlet resultat
    d. Den variable der opsamles data på skal synkroniseres til alle tråde
    e. Eksempel, En summering deles op: (Tegn eksempel fra disposition)
        i. Inputtet deles op i ”klumper”
        ii. Subtotal udregnes for hver ”klump”
        iii. ”klumper” samles sammen til en samlet total
    f. Dette er smart hvis man har mulighed for at dele arbejdet ud på flere tråde og/eller flere computere
3. Redegør for strukturen i MapReduce
    a. Tillader udregninger på store datasæt (xByte > 1 PetaByte)
    b. Følger 4 steps:
        i. Distribute – data deles ud til forskellige noder der laver udregninger samtidigt
        ii. Map – Laver Key Value Pairs på datasæt
        iii. Group – Samler Key Value Pairs, hvis Key er ens i to values samles disse I en Key Value Pair, hvor alle value værdier sættes til det, det var I det første Key Value Pair
        iv. Reduce – aggregere dataene til det ønskede fra den originale forespørgsel
    c. Eksempel, Energiproduktion:
        i. Gennemgå steps på eksemplet, tegn evt. op.
4. Parallel Loops (kom kun ind på denne hvis tid er)
    a. Hvis tråde arbejder med forskellige algoritmer kan der være behov for at summere sammen til slut -> Aggregation
    b. For mere se disposition

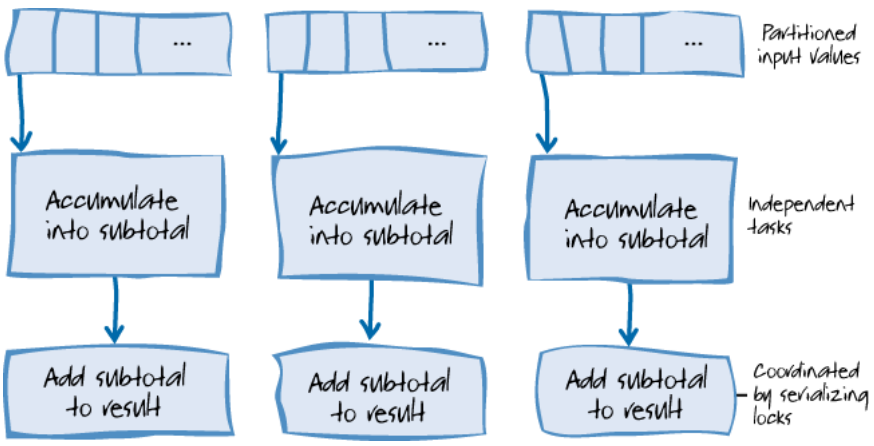
6. Parallel Aggregation + MapReduce

Parallel Aggregation

Aggregration er handling at opsamle ting/data til at forme en helhed. Når man snakker om parallel programmering, så kan man tænke på Aggregation som samler ”sub-resultater” og samler det til et samlet resultat. Hvis man har en meget tidskrævende beregning, som ønskes eksekveres, så kan man splitte beregningerne op i mindre bider vha. Parallel aggregation, som benytter sig af tråde på de enkelte bider. De enkelte dele er ikke afhængig af hinanden, men kan arbejde parallelt, og når alle tråde er færdig, så bliver resultaterne "aggregated" sammen igen.

Når der arbejdes parallelt er der ofte behov for at opsamle data, og når det skal ske på tværs af tråde så er vi nødt til at synkronisere den variable som vi opsamler data på.

Det kunne være et loop der udregner en sum. Dette har ikke uafhængige steps. Men der er stadig en måde at løse det på via parallel Aggregation pattern.



Det handler om at dele inputtet op i klumper og derefter udregne sub total for hver klump for til sidst at ende med at lægge summen sammen.

Et tænkt eksempel:

Der skal måles vandforbrug for et helt lejlighedskompleks. Målingerne opdeles i klumper bestående af én opgang (Step 1 på figuren). Alle lejlighederne i én opgangs målinger samles i en subtotal (Step 2), hvorefter alle subtotaler samles i ét stort resultat (Sidste step). Dataen blev altså akkumuleret parallelt og samlet til sidst.

I software verdenen bruges dette til at opdele opsamlingen af data så det kan gøres flertrådet eller på flere computere.

Kendte længde på input:

If the size of the input collection is known in advance, this can be converted into an instance of the aforementioned example, where the results are stored directly into the corresponding slots in the output:

C# Sequential

```
var output = new TOutput[input.Count];
for (int i = 0; i < input.Count; i++)
{
    var result = Compute(input[i]);
    output[i] = result;
}
```

This then makes parallelization straightforward, at least as it pertains to aggregation of the results:

C# Parallel

```
var output = new TOutput[input.Count];
Parallel.For(0, input.Count, i =>
{
    var result = Compute(input[i]);
    output[i] = result;
});
```

Ikke altid muligt, for hvis længde af input er ukendt:

C# Sequential

```
var output = new List<TOutput>();
foreach (var item in input)
{
    var result = Compute(item);
    output.Add(result);
}
```

Kan gøres parallelt ved

C# Parallel

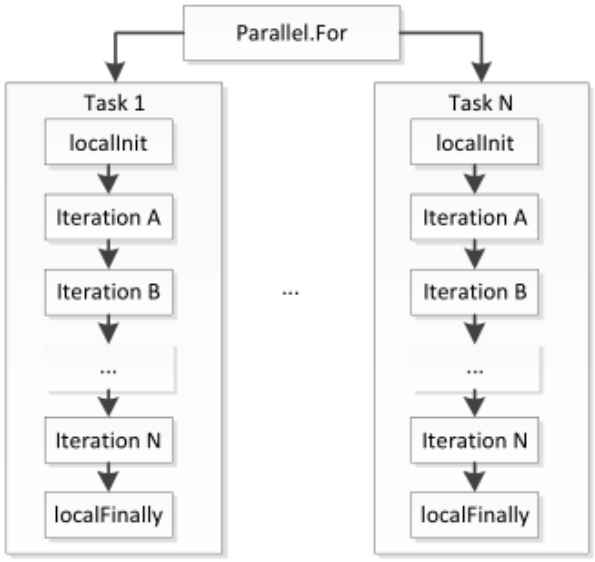
```
var output = new List<TOutput>();
Parallel.ForEach(input, item =>
{
    var result = Compute(item);
    lock (output) output.Add(result);
});
```

Men vi mister rækkefølgen i output i forhold til input. Da vi ikke ved rækkefølgen ”scheduleren” vælger. Det er også vigtigt at have ”explicit synchonization” for at sikre at output listen kun ændres af én tråd af gangen.

Sequential < parallel

Det er ikke altid bedre at lave parallelle tråde for en forløkke. Hvis beregningerne i den enkelte iteration er korte og simple, kan det koste flere ressourcer at oprette trådene samt at skulle tage og slippe låsen i hver tråd.

Der er det dog muligt at optimere ved at lave mindre forløkker som kan kører parallelt.



For at gøre det nemmere for programmøren er der LINQ OG PLINQ bibliotekerne.
Specielt PLINQ som har indbygget funktioner til at samle data fra tråde til f.eks en list. Hvor PLINQ tager sig af alt den nødvendige synkronisering.

C#

```
var output = new List<TOutput>();
foreach (var item in input)
{
    var result = Compute(item);
    output.Add(result);
}
```

This may be converted to a LINQ implementation as follows:

C#

```
var output = input
             .Select(item => Compute(item))
             .ToList();
```

And then it can be parallelized with PLINQ:

C#

```
var output = input.AsParallel()
             .Select(item => Compute(item))
             .ToList();
```

In fact, not only does PLINQ handle all of the synchronization necessary to do this aggregation safely, it can also be used to automatically regain the ordering we lost in our parallelized version when using **Parallel.ForEach**:

C#

```
var output = input.AsParallel().AsOrdered()
             .Select(item => Compute(item))
             .ToList();
```

MapReduce

Map Reduce er et pattern der tillader parallelle udregninger på store data sæt. Store sæt er ofte større end 1 PB (PetaByte).

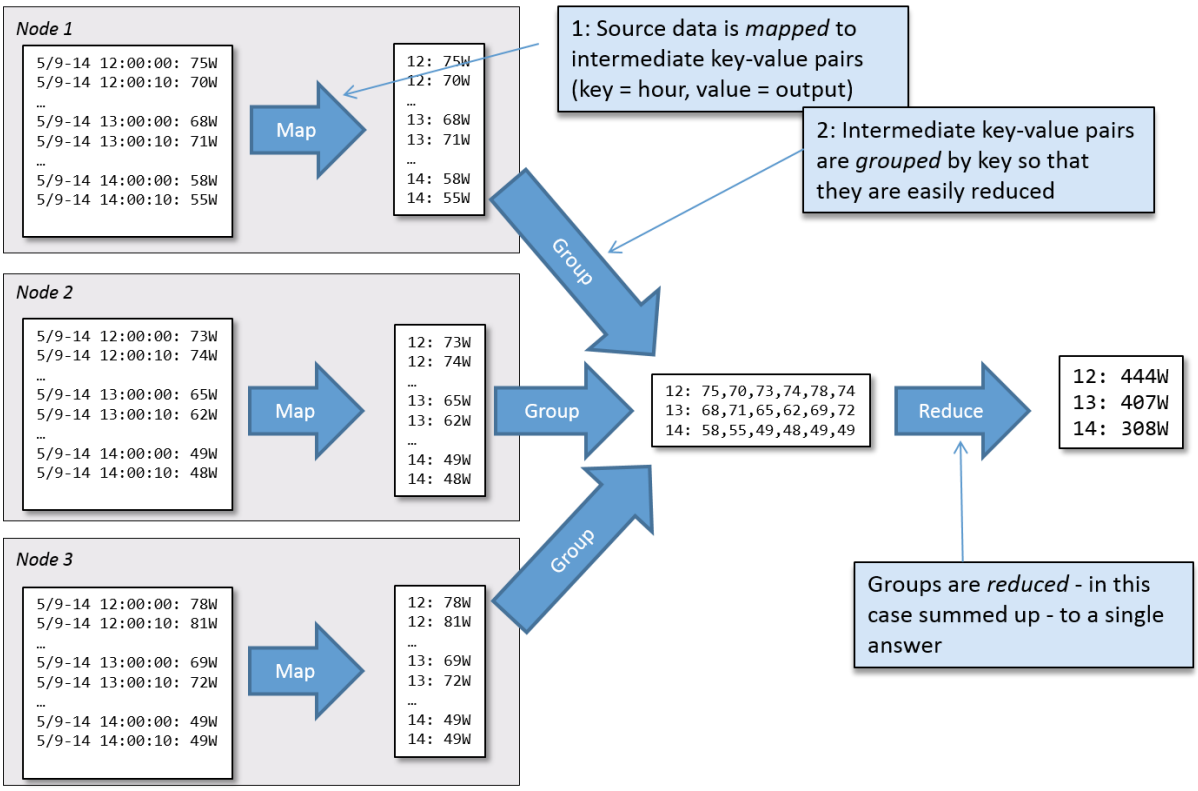
- *Distribute* and *Group* do not vary – usually provided by a framework
- *Map* and *Reduce* vary – provided by user

Der er 4 steps i Map Reduce

- 1. Distribute – Deler data ud til de forskellige noder der skal lave beregninger for at kunne arbejde parallelt (Forskellige PC'er)
- 2. Map – Laver key value pairs på datasættet.
- 3. Group – Samler key value pairs så alle values der har samme key bliver samlet i et key value pair hvor value har alle de value der var i de første key value pairs.
- 4. Reduce – Aggregere data til det svar vi ønskede fra den oprindelige forespørgsel.

Som eksempel kunne man se på energi produktionen for sol celle anlæg her kunne vi spørge hvor meget EL er der blevet produceret for hver time i døgnet.

1. Distribute (master)– vil her været at dele logfilerne ud til de forskellige noder
2. Map – Kunne være at tage alle entries for energi ved timen 12 og så lave det til et key value pair.
3. Group – Her samles alle key values så key 12 vil indeholde alle de aflæste kW
4. Reduce – Her ville vi så summere alle kW for f.eks. Key 12.




```
// Patterns of Parallel Programming p. 75
public ParallelQuery<TResult> MapReduce<TSource, TMapped, TKey, TResult>(
    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);
}
```

Prototype for MapReduce funktion.

The end

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"]]

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

AARHUS UNIVERSITY 20

De 4 steps vist sammen, med enkelte eksempler undervejs

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

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


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

Source() provides the source data

Map() maps each source data item to a key

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

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"]

```
// Source() returns the source data on which the MapReduce query shall run.
static IEnumerable<string> Source(string path)
{
    return File
        .ReadLines(path)                                  // Read all lines in the path
        .SelectMany(line => line.ToLower().Split(' ')); // Project words into an IEnumerable
}
```

2. *Map*: Map each word to key-value pair, where key = length of word

"The" → [3: "The"],
"fox" → [3: fox],
"hound" → [5, "hound"],
…

```
// Map() returns the key which the word fits
static int Map(string word)
{
    return word.Length;          // The key is the length of the word
}
```

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())
    };
}
```

Ved ikke om følgende skal bruges….

Parallel Loops

Bruges til at udføre de samme udregninger/algoritmer på forskellige data på samme tid.

De bruges når mange operationer eller udregninger kan køre af sig selv. Det kan være tale om udregninger i en for løkke hvor udregningerne er uafhængige af iterationerne.

Vi kunne lave vores egen funktion der opdelte arbejdet vi ønskede at få udført i mindre bidder for herefter at starte en tråd til hver af klumperne.

```
public static void MyParallelFor(
    int inclusiveLowerBound,
    int exclusiveUpperBound,
    Action<int> body);
```

- Action<int> er en delegate som pakker en funktion ind
- Den tager en int og returnerer void

Hvor mange tråde skal der så bruges? Ikke for mange da det er dyrt at skifte mellem tråde dvs hvis skal lave så det passer med de CPU'er vi har til rådighed.

```
int size = exclusiveUpperBound - inclusiveLowerBound;
int numProcs = Environment.ProcessorCount;
int range = size / numProcs;
```

Example: size = 35, numProcs = 4 → range = 8

Når opdelingen er fundet kan der oprettes det antal tråde der skal bruges og derpå kan de startes.

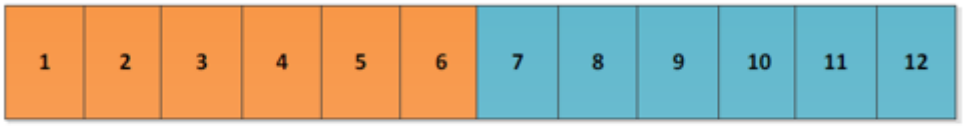
```
public static void MyParallelFor(
   int inclusiveLowerBound, int exclusiveUpperBound, Action<int> body)
{
   // Determine size of each partition of work (size/nCores)
   int size = exclusiveUpperBound - inclusiveLowerBound;
   int numProcs = Environment.ProcessorCount;
   int range = size / numProcs;

    // Initialize threads to do work
   var threads = new List<Thread>(numProcs);
   for (int p = 0; p < numProcs; p++)
   {
      int start = p * range + inclusiveLowerBound;
      int end = (p == numProcs - 1) ? exclusiveUpperBound : start + range;
      threads.Add(new Thread(() => {
         for (int i = start; i < end; i++) body(i);
      }));
   }

   // Start and await threads
   foreach (var thread in threads) thread.Start();  // Start them all
   foreach (var thread in threads) thread.Join();   // wait on all
}
```

Statisk eller dynamisk partitionering?

Hvis vi har lavet statisk partitionering her vi ingen synkroniserings problemer men hvad hvis nogle af opgaverne tager længere tid end de andre? Ja så må vi vente på den sidste bliver færdig.

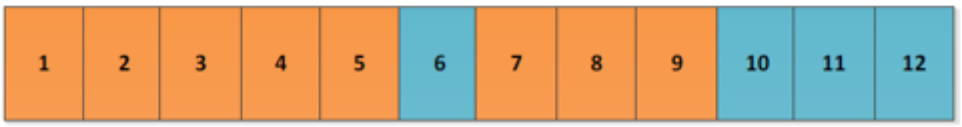
Effective *static* partitioning requires a priori knowledge on execution time, but requires no synchronization

- But is less-than-ideal



Effective *dynamic* partitioning requires no knowledge, but requires synchronization

- Or the threads will step on each other's toes



Hvis der derimod er lavet dynamisk partitionering skal vi synkronisere da trådene ellers vil komme op og slås over hvem der skal lave hvad.

Hvor er det så vigtigt? For hvis vi har tråde der står og venter så laver vi ikke ret meget på CPUen. Dette kan ske hvis vi har for mange tråde eller vores partitionering er forkert.

I .NET

Der findes en parallel implementering af For og ForEach.

- `Parallel.For(start, end, Action)`
- `Parallel.ForEach(collection, Action)`
- `Parallel.Invoke(Action)`

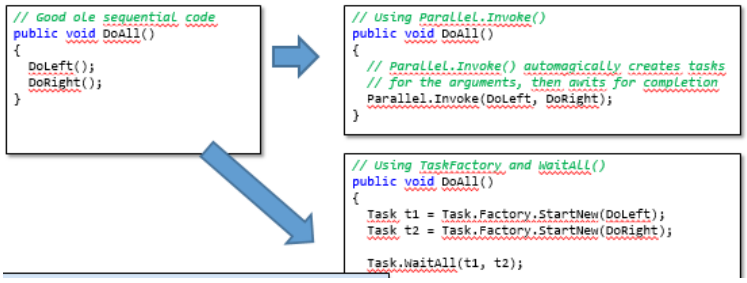
Disse er gode at bruge da de har exeption handling og generalt er optimerede. Er det så lykken?

Ikke altid hvis ikke iterationerne er uafhængige eller hvis det er et meget lille loop.

Parallel Task

En task kan ses som et isoleret logisk stykke arbejde – En sekventiel operation der kan paralleliseres.

Det er ikke en tråd, det er en sekventiel operation! Kan bruges når man har flere adskilte asynkrone operationer. De bor i Task Parallel Library de bruger dynamisk skalering af paraellelismen for best muligt at udnytte alle de tilgængelige processorer.




```
// Using WaitAll()
public void DoAllUsingWaitAll()
{
  Task t1 = Task.Factory.StartNew(DoLeft);
  Task t2 = Task.Factory.StartNew(DoRight);

  Task.WaitAll(t1, t2);
}
```

```
// Using WaitAny()
public void DoAllUsingWaitAny()
{
  Task[] tasks = {
          Task.Factory.StartNew(DoLeft),
          Task.Factory.StartNew(DoCenter),
          Task.Factory.StartNew(DoRight)
  };

  while(tasks.Length > 0) {
    var taskIndex= Task.WaitAny(tasks);

    // Do meaningful work with task[taskIndex]

    // Update the set of tasks upon which we wait
    tasks = tasks.Where((t) => t != tasks[taskIndex]).ToArray();
  }
}
```

Tasks kan startes på mange måder og der kan selvfølgelige ventes på dem. Den der har kaldt wait vil få eventuelle exceptions!

Task Scheduler

Da tasks er fint kornet arbejde er det vigtigt at sørge for at det er let at skifte tasks hvis det er gjort ineffektivt kan det være en flaskehals. TPL Bruger arbejder tråde til at eksekvere tasks.

De bliver styret af .NET ThreadPool klassen der er mindst en tråd pr cpu kerne. Task sættes i kø på trådene der kan være mange task til få tråde.