9. Patterns 7: Redegør for følgende concurrency mønstre

- Parrallel Loops
- Parrallel Task

Der findes overordnet to typer af concurrency

- Data parallelism
  - Samtidigheden kommer af at udføre de samme algoritmer/udregninger på forskellig data, på samme tid!
  - Det tager vores Parallel Loops af
- Functional parallelism eller task parallelism
  - Samtidigheden kommer af at udføre/håndtere forskellige funktioner på samme tid
  - Det tager Parallel Task sig af

Parallels Loops:

- Parallel Loops skal bruges når:
- Mange operationer/udregninger kan køre forholdsvist uafhængig af hinanden
  - Typisk hvis man arbejder på en collection, som der skal gøres noget ved i en for-løkke, hvor hver udregningerne er uafhængig

- Det implementeres ved at starte et antal tråde til at arbejde forskellige steder på en collection.

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

- Action<int> body er en delegate som pakker en funktion ind
- Funktionen må tage en int som parameter og returnere void
- Parameteren er den iteration vi er nået til

- Men hvor mange tråde skal vi oprette? Tommelfinger regel siger 1 tråd per fysisk kerne.
- For mange tråde kan give for mange context switching, mere om det..

- Dernæst skal vi have delt vores opgave ud, også kaldt **Partitioning**
    - Hvor mange tråde vi opretter
    - Hvor mange iterationer der skal køre pr. tråd
    - Hvilke iterationer skal køre på hvilke tråde

```
// Determine the number of iterations to be processed, the number of
// cores to use, and the approximate number of iterations to process
// in each thread.
int size = exclusiveUpperBound - inclusiveLowerBound;
int numProcs = Environment.ProcessorCount;
int range = size / numProcs;
```

- Her laves en tråd pr. core. Range er antal iterationer pr. tråd, delt ligeligt lidt.

```
// Use a thread for each partition.
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);
   }));
}
foreach (var thread in threads) thread.Start();  // Start them all
foreach (var thread in threads) thread.Join();   // wait on all
```

Problemer med designet:

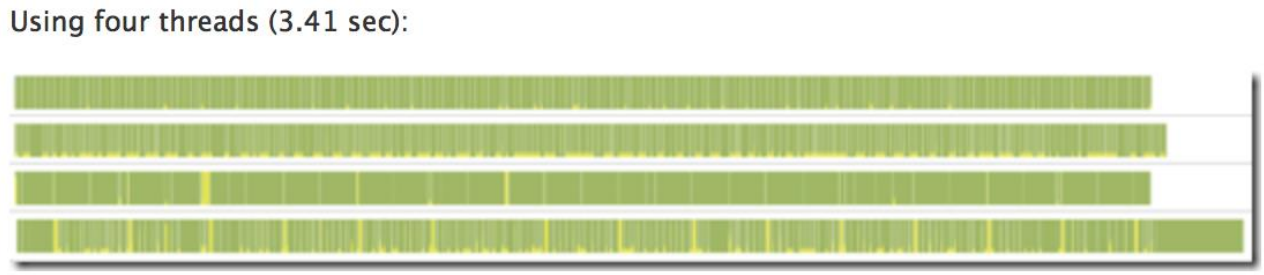
- Oprettelse af tråde er forholdsvist dyrt, fx i hukommelse (1 megabyte pr. tråd)
- Men tråd oprettelse tager også i CPU power, så hvis iterationen og udregningerne er forholdsvis små, så er der ikke idé i at bruge dette pattern
- **Load balance** kan være skæv, fordi det er ikke sikkert hver tråd har lige meget arbejde
    - Således bliver en tråd færdig før en anden, og man starter ikke en ny og ompartitionere arbejdet, således bruges alle ens cores ikke
    - Hvis man skal kunne ompartitionere skal trådene snakke sammen om “hvad skal jeg gøre næste gang”
        - tæller en nextIteration counter op vha. Interlocked.Increment -> Overhead!
    - Ideel hvis man kunne forudsige arbejdsbyrden på forhånd, og tildele tråde ud fra det

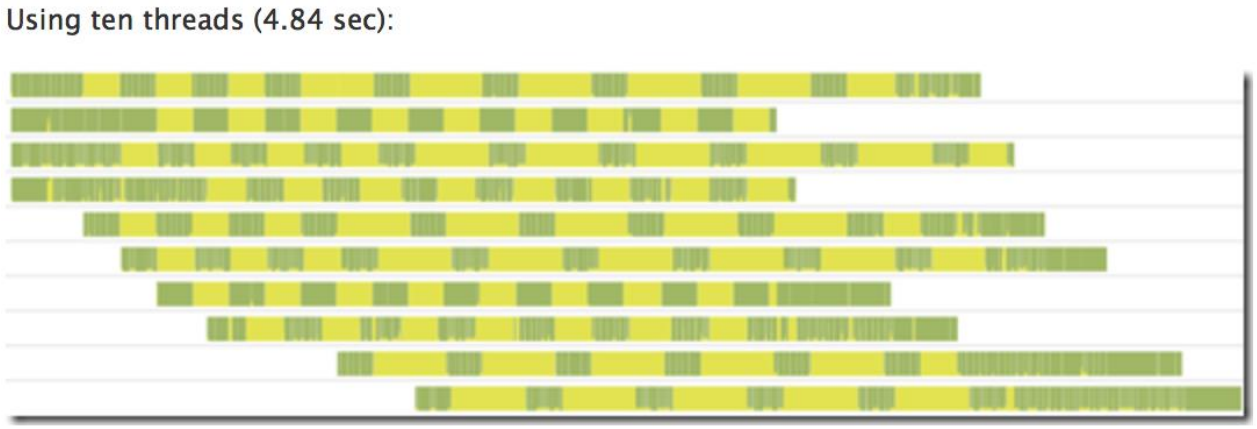
Oversubscription:

- Et andet problem kan det lidt ala det omvendte nemlig **oversubscription**
    - Dvs. vi har lavet for mange tråde

I et quad core setup (eksempel fra MSDN)

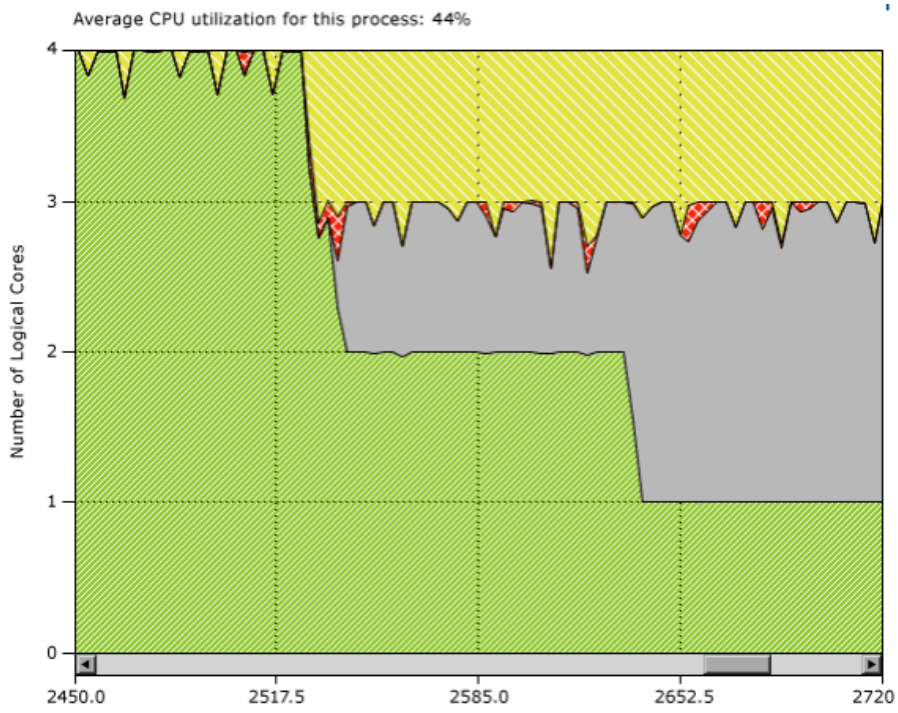
hvor 





Preemption: Vente på at en ressource bliver frigivet, context switch ol.

Load imbalance:



Load imbalance er når man har en ulige fordeling af arbejdet på kernerne.
Det går ikke en 1 kerne bliver dobbelt så hurtig færdig med mine opgaver end de andre. Så har man ikke fordelt arbejdet godt nok.
Kender man opgavernes størrelse i tid bør man fordele arbejdet på kernerne så alle kerne bliver færdig på samme tid -> det vil resultere i at arbejdet bliver udført hurtigst muligt.

.NET frameworket har allerede implementeret

```
//.NET PARALLEL
Console.WriteLine("Starts .NET Parallel.For:");
sw.Reset();
sw.Start();
Parallel.For(0, n, i =>
{
    c2[i] = a[i]*b[i];
});
sw.Stop();
Console.WriteLine(".NET Parallel.For time {0} ms", sw.ElapsedMilliseconds);
```

- Tager højre for oversubscription og load imbalance vha. avancerede algoritmer
- Supporterer at bryde ud af loop, cancellation token
- Supporterer brug af thread-local storage
- Exception handling (hvilken tråd kastede exception)
- Meget mere (nested parallel.for loops arbejder)

Parallel Task:

Kaldes også fork/join pattern og master/worker pattern

- Skal bruges når:
    - Der er flere tydelig adskilte operationer som kan køres på samme tid
    - Man kan forke sit arbejde ud og arbejde parallelt, og senere joine dem sammen for at sikre at alle er kommet hvor de skal være.

- Dette eksempel viser hvordan man nemt kan specificere 3 statements som skal køre i parallel.

```
Parallel.Invoke(
    () => ComputeMean(),
    () => ComputeMedian(),
    () => ComputeMode());
```

- På et dataset vil vi gerne udregne median, gennemsnit og mode
- Det er adskilte operationer som ligeså godt kan køre på samme tid

- Implementere det selv

```
static void MyParallelInvoke(params Action[] actions)
{
  var tasks = new Task[actions.Length];
  for (int i = 0; i < actions.Length; i++)
    {
      tasks[i] = Task.Factory.StartNew(actions[i]);
    }
  Task.WaitAll(tasks);
}
```

- params keyword betyder at vi kan tage flere argumenter af typen Action
- Task repræsenterer en asynkron action
- Når StartNew kaldes, startes den Task.
- **Task er lidt ligesom en tråd men tager for sig af ThreadPool som C# stiller til rådighed.**
- Task.Run()

- Vores implementering kunne også bare kalde vores MyParallelFor funktion og så delegere opgaverne ud i en tråd til hver opgave
- Parallel.Invoke vil ud fra antallet af actions bestemme om det er værd at oprette en tråd for hver funktion
- Først når alle funktionerne dvs. ComputeMean, Meadian og Mode er færdige, anses hele handling som færdig og handlingerne JOINES.

```
static int ParallelTaskImageProcessing(Bitmap source1,
   Bitmap source2, Bitmap layer1, Bitmap layer2,
   Graphics blender)
{
  Task toGray = Task.Factory.StartNew(() =>
     SetToGray(source1, layer1));
  Task rotate = Task.Factory.StartNew(() =>
     Rotate(source2, layer2));
  Task.WaitAll(toGray, rotate);
  Blend(layer1, layer2, blender);
  return source1.Width;
}
```

- I dette eksempel er SetGrayScale uafhængig af Rotate
- De kan derfor køres som tasks
- Men de skal dog joines inden de skal Blendes sammen
- Brug fork/join når rækkefølgen er vigtig

Tasks er at fortrække over blot en tråd da de

- Placeres i en work queue, som håndteres af systemet
- Man kan ikke forvente at det kører parallelt, det kommer an på hvad task scheduleren mener er bedst
- Supporterer
    - At blive stoppet og startet igen

    - Har exception handling
    - Detaljeret status for task
    - Deres prioritet/scheduling kan konfigureres mm.

- Til sidst kan dernæst kalde Wait på en task, for at joine flere tråde sammen. Fx for at synkronisere noget data mellem dem til sidst.
- Hvis task har returværdi, overvej at bruge futures i stedet for

Improving our parallel.for design

C#

```
public static void MyParallelFor(
    int inclusiveLowerBound, int exclusiveUpperBound, Action<int> body)
{
    // Get the number of processors, initialize the number of remaining
    // threads, and set the starting point for the iteration.
    int numProcs = Environment.ProcessorCount;
    int remainingWorkItems = numProcs;
    int nextIteration = inclusiveLowerBound;

    using (ManualResetEvent mre = new ManualResetEvent(false))
    {
        // Create each of the work items.
        for (int p = 0; p < numProcs; p++)
        {
            ThreadPool.QueueUserWorkItem(delegate
            {
                int index;
                while ((index = Interlocked.Increment(
                    ref nextIteration) - 1) < exclusiveUpperBound)
                {
                    body(index);
                }
                if (Interlocked.Decrement(ref remainingWorkItems) == 0)
                    mre.Set();
            });
        }

        // Wait for all threads to complete.
        mre.WaitOne();
    }
}
```

- Problematikken ved parallel.foreach er at man skal have fat i enumatoren for collection, og den skal deles mellem tråde