

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

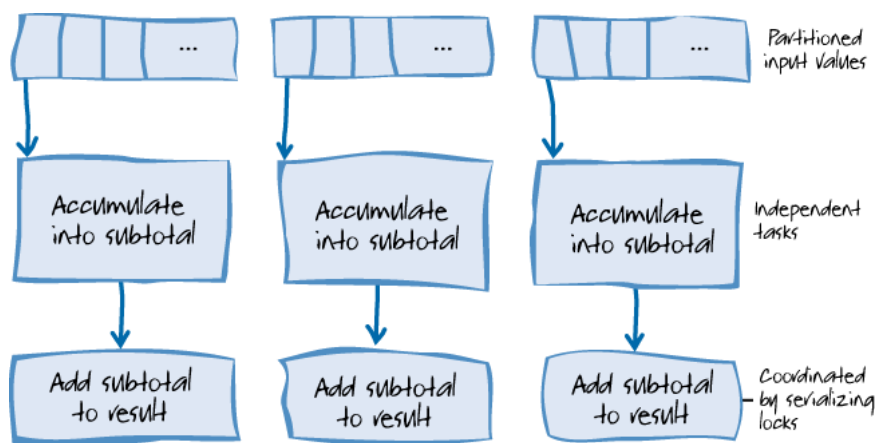
6. Parallel Aggregation + MapReduce

Parallel Aggregation

Aggregation 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.



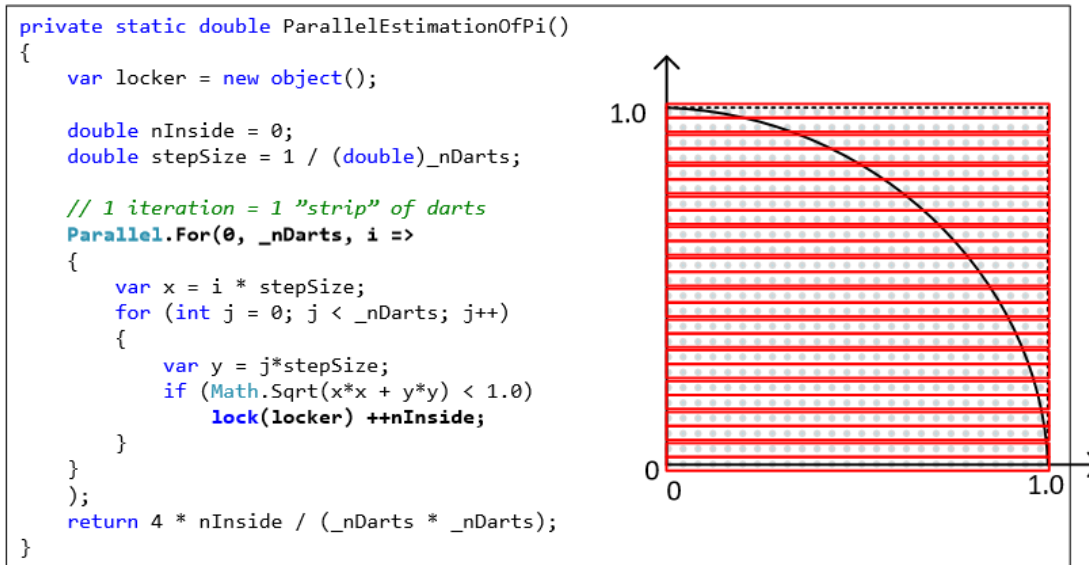
Det handler om at dele inputtet op i klumper og derefter udregne sub total for hver klump for til sidst at ende med at lægge summen sammen.

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 subtotaller samles i ét stort resultat (Sidste step). Dataen blev altså akkumuleret parallelt og samlet til sidst.

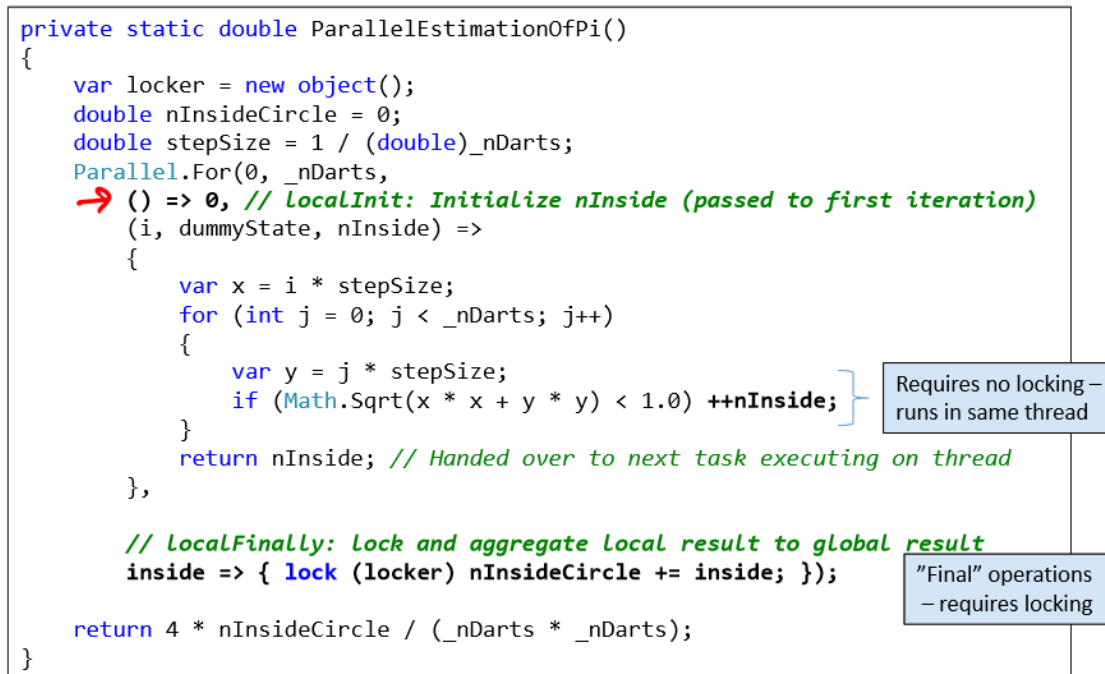
I software verdenen bruges dette til at opdele opsamlingen af data så det kan gøres flertrådet eller på flere computere.

- Parallel estimation of pi – 1st attempt



Der bliver lavet en ny tråd for hver af de beregninger der skal laves. Inde i selve tråden bliver der lavet en for-løkke, hvor der bliver lavet en beregning på om den pågældende data er inden i for "range". Hvis der er 50 data punkter i eksemplet ovenover, så bliver der brugt en "lock" 250 gange, på en statisk variabel. Dette betyder at paralleliseringen ikke er helt så hurtigt, da låsningen spærrer for udførelsen af andre tråde.

- Parallel estimation of pi – 2nd attempt



I denne parallelisering bliver der også brugt en statisk variabel, dog bliver denne ikke brugt hver gang, da delresultatet bliver lagt ind i en lokal variabel. Dette betyder at man kan bruge samme opbygning som i forrige "attempt". Ved at bruge en lokal variabel er man i stand til kun at låse det antal gange man laver en ny tråd.

- Parallel estimation of pi – 3rd attempt

```
private static double ParallelEstimationOfPiWithPartitioner()
{
    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

I modsætningen til de forrige eksempler, hvor der bliver oprettet en tråd til at lave en beregning, så bliver der her oprettet en tråd til at beregne på et mindre data sæt. Dette går hurtigere da man her er i stand til at have et færre antal tråde, som scheduleren skal skifte mellem, og derved får hver tråd mere tid. En anden fordel ved dette er at man er i stand til at lave det antal tråde som man har fysiske kerner til.

MapReduce

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

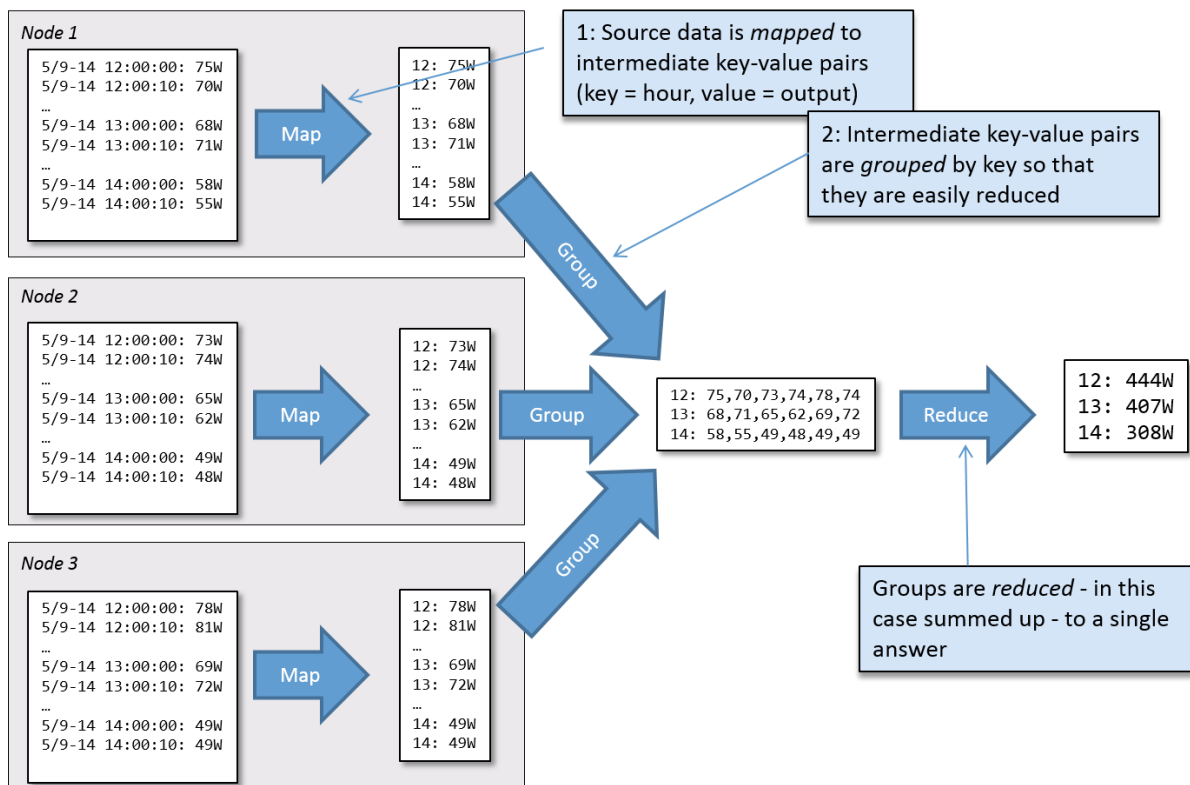
- Distribute* and *Group* do not vary – usually provided by a framework
- Map* and *Reduce* vary – provided by user

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 – 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.

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]

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 opdelt arbejdet vi ønskede at få udført i mindre bidder for herefter at starte en tråd til hver af klumperne.

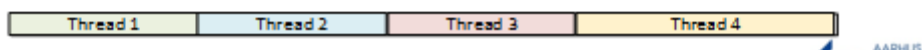
- Action<int> er en delegate som pakker en funktion ind
- Den tager en int og returnerer void

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

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.

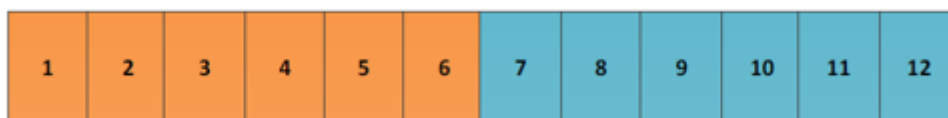
```
public static void MyParallelFor(  
    int inclusiveLowerBound, int exclusiveUpperBound, Action<int> body)  
{  
    // Determine size of each partition of work (size/nCores)  
    int size = exclusiveUpperBound - inclusiveLowerBound;  
    int numProcs = Environment.ProcessorCount;  
    int range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

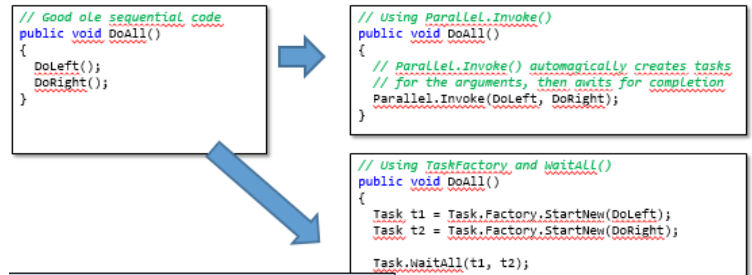
Disse er gode at bruge da de har exception handling og generalt er optimerede. Er det så lykken?

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 parallelismen 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 selvfølgelig 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.