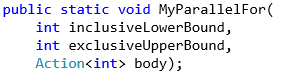
9. Patterns 7: Redegør for følgende concurrency mønstre

# Parallel Loops

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

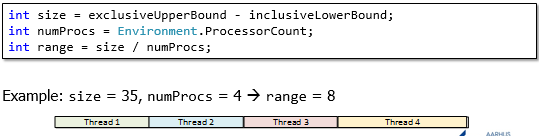
De bruges når mange operationer eller udregninger kan køre af sig selv. Det kan være tale om udregninger i en for løkke hvor udregningerne er uafhængige af iterationerne.

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.

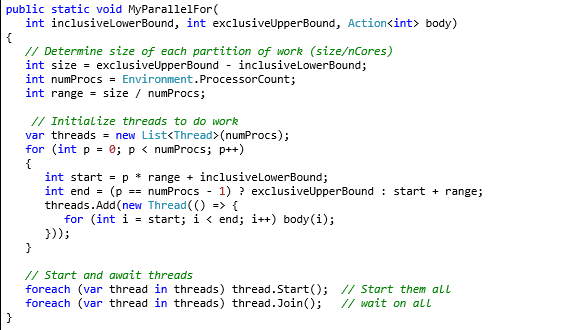


* Action<int> er en delegate som pakker en funktion ind
* Den tager en int og returnere void

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 CPUer vi har til rådighed.

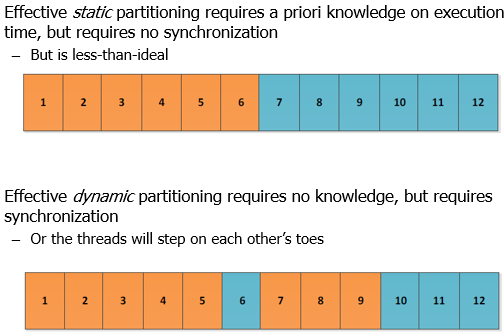


Når opdelingen er fundet kan der oprettes det antal tråde der skal bruges og derpå kan de startes.



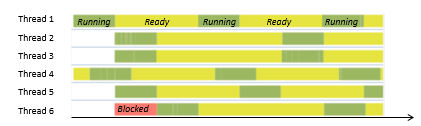
## Statisk eller dynamisk partitionering?

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



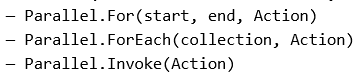
Hvis der derimod er lavet dynamisk partitionering skal vi synkronisere da trådene ellers vil komme op og slås over hvem der skal lave hvad.

Hvor er det så vigtigt? Fordi at 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 dot NET

Der findes en parallel implementering af For og ForEach.

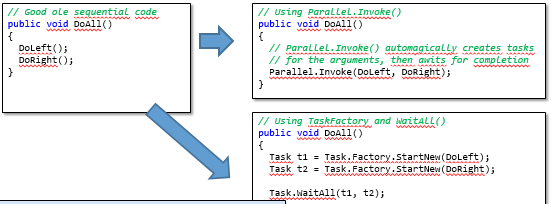


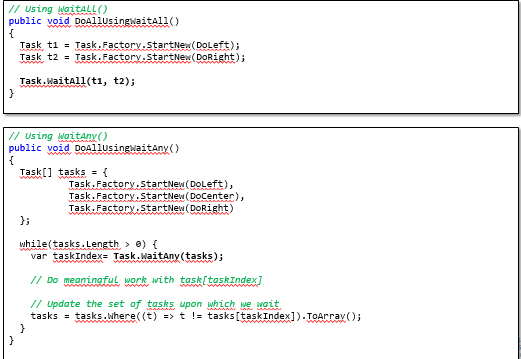
Disse er gode at bruge da de har exeption handling og generalt er optimerede. Er det så lykken?

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.



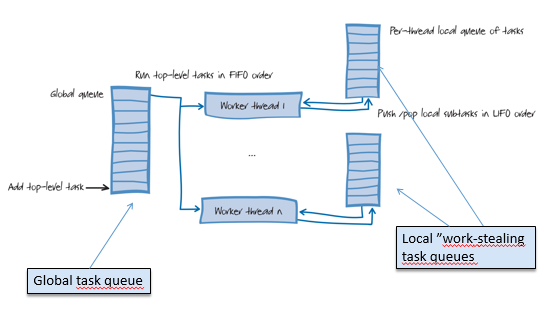
Tasks kan startes på mange måder og der kan selvfølgelige ventes på dem. Den der har kaldt wait vil få eventuelle exceptions!

## Task Scheduler

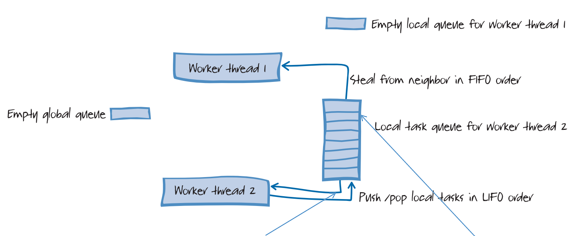
Da tasks er fin kornet arbejde er det vigtigt at sørge for at det er let at skifte tasks hvis det er gjort ineffektivt kan det være en flaskehals. TPL Bruger arbejder tråde til at eksekvere tasks.

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.

Task Scheduleren i .NET er implementere med en global task kø og hver tråd har deres egen lokale kø. Det betyder at tråde kan hente arbejde 3 steder i deres egen lokale kø (kræver ikke synkronisering) i den globale kø eller i en anden tråds kø.



En task der kan blive inlineed i en anden task hvis den er i arbejder trådens kø og den task der bliver afviklet på tråden venter på den task der er i køen.



## Parametre til tasks

En task kan ved hjælp af closures få parametre med. Eller ved hjælp af state objects.

