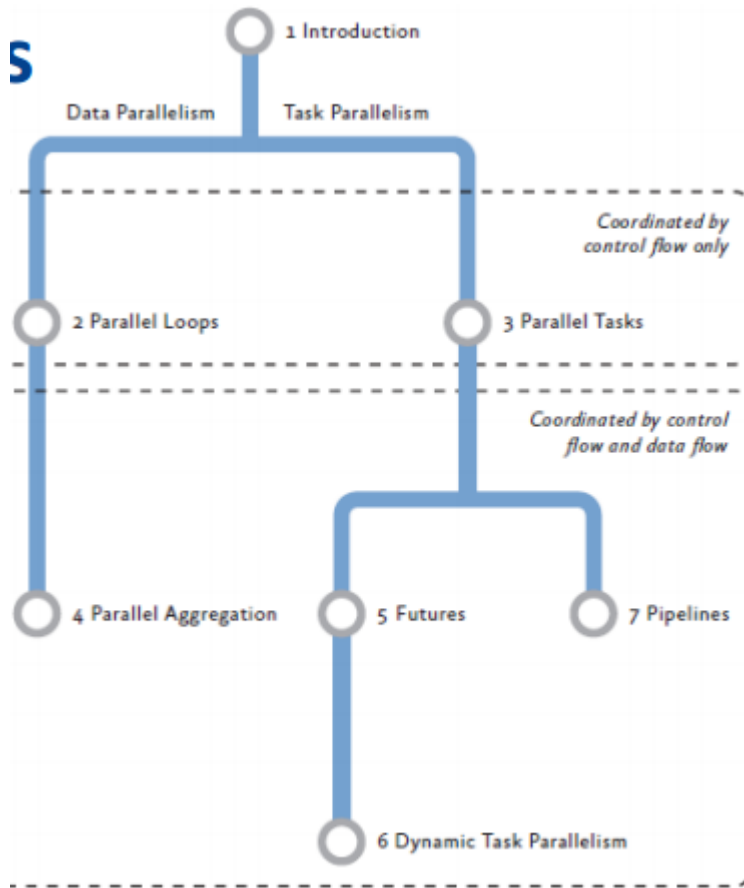


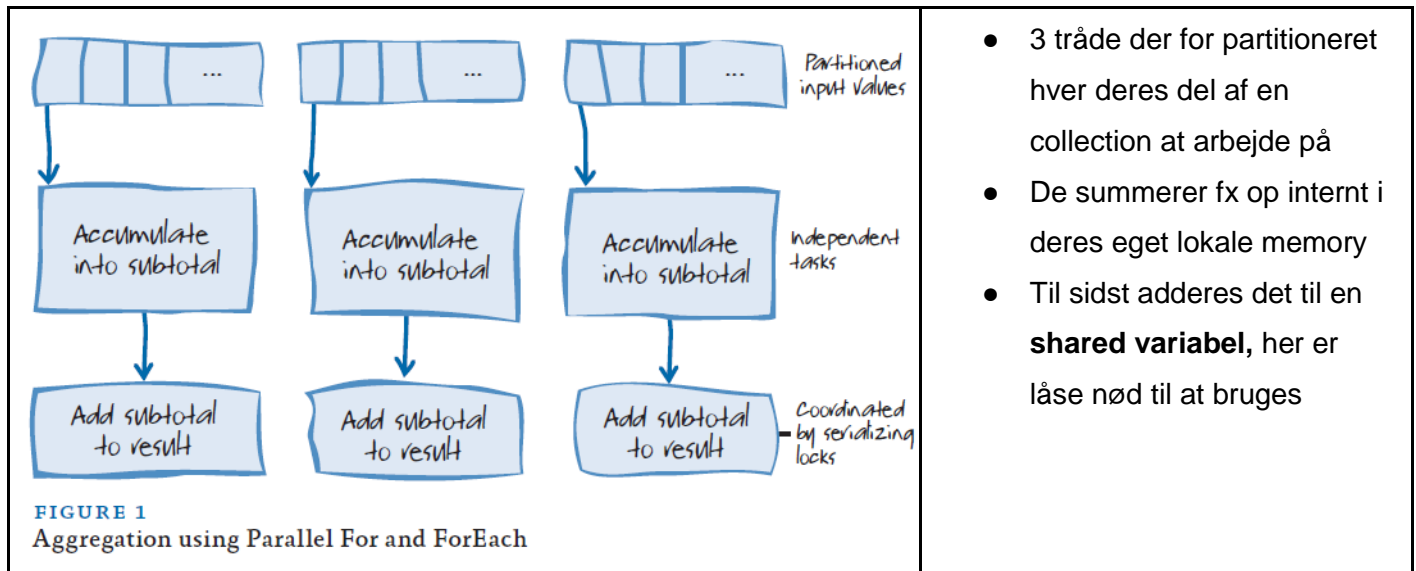
## 10. Patterns 8: Redegør for følgende concurrency mønstre

- Parallel Aggregation
- Futures (task graph)



### Parallel aggregation

- Parallel aggregation er når fx
  - I et parallel loop arbejder tråde på forskellige steder i en collection
  - Hvor tråden fx skal udføre en algoritme på dataen
  - Dernæst ønsker at fx summere deres del sammen med de andres trådes dele
  - Der er denne sum en aggregation
- Skal bruges når noget arbejde kan køres uafhængig af hinanden, men hvor der til sidst er noget samlearbejde
  - Typisk matematiske operatorer såsom plus og gange dvs. de skal være:
  - **Associative** dvs. grupperingen af elementer er ligegyldig  $a+(b+c) = (a+b)+c$
  - **Commutative** dvs. rækkefølgen af operanderne ændrer ikke output  $a+b = b+a$



Parallel\_Programming\_with\_Microsoft\_dotNET

Det kan gøres i Dot-Net vha. en specialiseret udgave af ForEach:

```
//MSDN Example
// The sum of these elements is 40.
int[] input = { 4, 1, 6, 2, 9, 5, 10, 3 };
int sum = 0;

Parallel.ForEach(
    input,                                // source collection
    () => 0,                             // thread local initializer
    (n, loopState, localSum) =>          // body loop for each int in input
    {
        localSum += n;
        return localSum;
    },
    (localSum) => Interlocked.Add(ref sum, localSum) // thread local aggregator
);
```

- Specialiseret udgave af ForEach løkke der supporterer en thread local value, et loop body og endnu et action til sidst hvor man kan sammensætte data.
- Supporterer altså direkte vores Parallel Aggregation pattern
- Her der det gjort vha. closure?

### Futures (task graph)

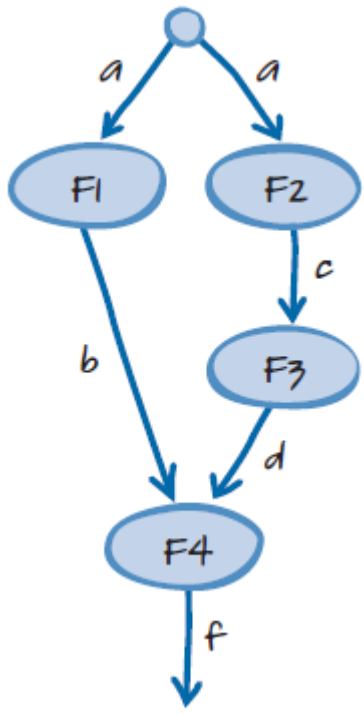
- En future er en variabel som endnu ikke er udregnet, men som i fremtiden vil blive udregnet
  - Måske parallelt med noget andet
  - Måske sekventielt
  - Styret af underliggende logik i .NET scheduler

- Skal bruges når man vil have mulighed for at køre funktioner med returværdier asynkront
  - Hvis man ved man i fremtiden har brug for dens returværdi, og at vi ligeså godt kan starte med at udregne det nu
- I .NET er en future en Task med en returværdi vha. en Task<TResult> hvor TResult er typen af resultatet fx int, float ol.

Eksempel fra Parallel Programming with DotNet

<pre>var b = F1(a); var c = F2(a); var d = F3(c); var f = F4(b, d); return f;</pre>	<ul style="list-style-type: none"> <li>• Funktioner F1-4 kunne være meget CPU intensive</li> <li>• Kunne derfor være en fordel af køre nogle parallel</li> <li>• Men hvilke kan køres parallelt?</li> <li>• Vi kan fx se at F3 har brug for at F2 er kørt</li> </ul>
---	--

- Til det kan man tegne **task graph**

 <pre> graph TD     Root(( )) -- a --&gt; F1((F1))     Root -- a --&gt; F2((F2))     F1 -- b --&gt; F4((F4))     F2 -- c --&gt; F3((F3))     F3 -- d --&gt; F4     F4 -- f --&gt; Exit(( ))   </pre>	<ul style="list-style-type: none"> <li>• Cirklerne er tasks/funktioner</li> <li>• linjerne er værdier som bliver givet videre gennem argumenter og returværdier</li> <li>• Var a skal processeres af F1 og F2, det ses tydeligt at dette kan gøres parallelt</li> <li>• Hvorimod det kan ses at F3 afhænger af at F2 er færdig</li> <li>• Og at F4 afhænger af at F1 er færdig</li> <li>• <b>Overblik over hvad kan paralleliseres?</b></li> </ul>
--	--

Løsning med futures:

```
Task<int> futureB = Task.Factory.StartNew<int>(() => F1(a));
int c = F2(a);
int d = F3(c);
int f = F4(futureB.Result, d);
return f;
```

- Vi laver en future som returnerer en int: Task<int>
- Kalder den futureB og starter task'en, fortæller den skal starte en funktion som tager int som argument
- **Vi opnår at F1 og F2 kan køre på sammen tid!**
- F3 kører efter F2
- Så snart det data er klar som F4 skal bruge, få kører dens udregning
- Data kunne være klar efter F3, eller det kunne være F1 tog meget længere tid

### Continuation task:

- En continuation task er en funktion der automatisk startes så snart andre tasks kaldet antecedents (forgængere) er blevet kørt
- Gør man det vha. continuation task gør man det muligt for run-time environment at bedre schedule disse tasks sammen

```
TextBox myTextBox = ...;

var futureB = Task.Factory.StartNew<int>(() => F1(a));
var futureD = Task.Factory.StartNew<int>(() => F3(F2(a)));

var futureF = Task.Factory.ContinueWhenAll<int, int>(
    new[] { futureB, futureD },
    (tasks) => F4(futureB.Result, futureD.Result));

futureF.ContinueWith((t) =>
    myTextBox.Dispatcher.Invoke(
        (Action)(() => { myTextBox.Text = t.Result.ToString(); })))
    );
```

- F1 og F3 kan køre i parallel så vi starter en ny future for hver
- futureF kan først køre når begge dens forgængere er færdig
- Viser ved at bruge ContinueWhenAll
  - Den kører F4 på resultatet af de to startede futures
- ContinueWith bruges da den kun skal vente på at F4 bliver kørt.
- Her opdateres GUI ved at finde GUI tråden vha. dispatcheren.

### Gode ting ved futures:

- Undgår at skulle arbejde med shared variable
  - Miste performance på låse
  - Korrupt data