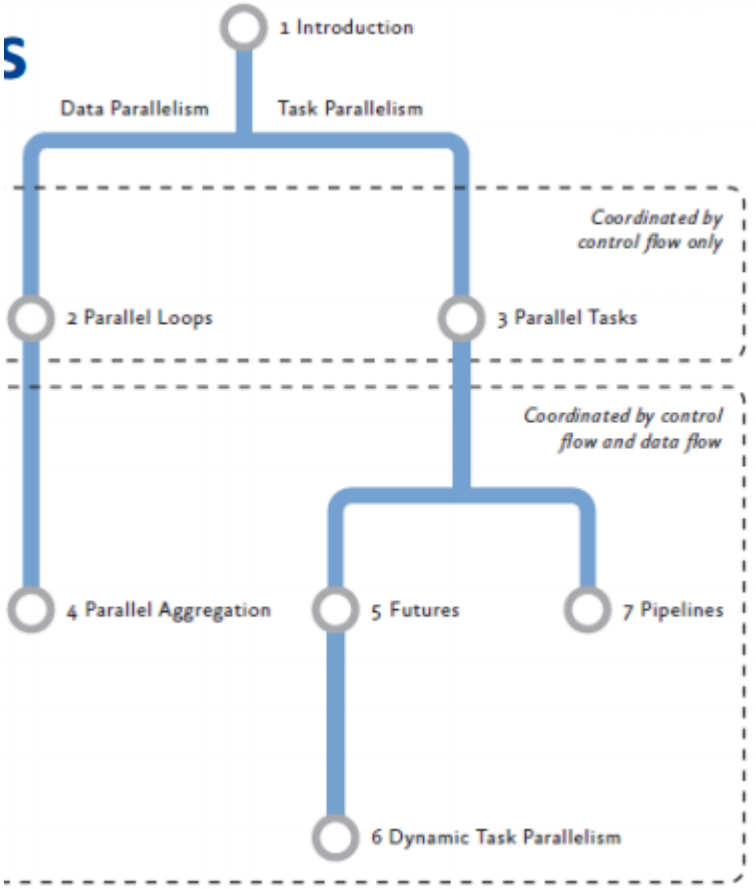
11. Patterns 9: Redegør for følgende concurrency mønstre

- Dynamic Task Parallelism
- Pipelines

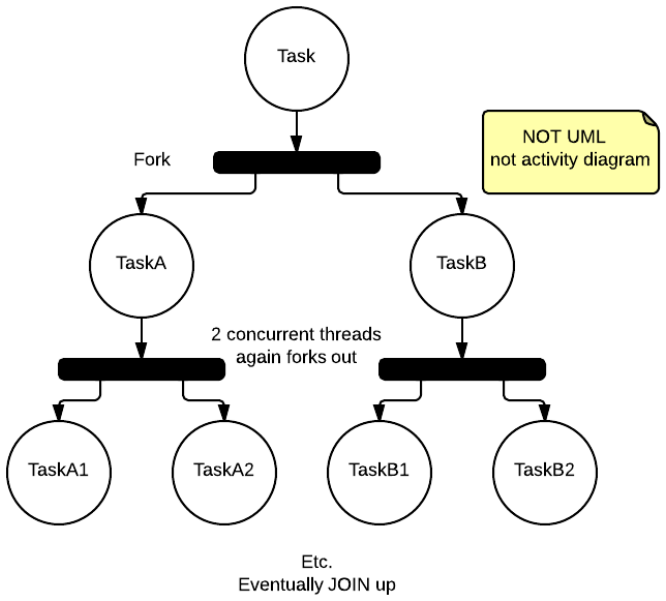


Dynamic Task Parallelism:

- Dynamisk task parallelism er hvor tasks selv dynamisk laver flere tasks ”on the fly” i forbindelse med en algoritme
  - Task i .NET er en funktion som der kan køre parallelt med andre tasks, asynkron funktion
  - Minder om en tråd men gør brug af thread-pool til kørsel, og indeholder mere funktionalitet end tråde (Exception handling, Start/stop af task)

```
Task t1 = Task.Factory.StartNew(() =>
{
      Console.WriteLine(“Hello”);
      task t1a = Task.Factory.StartNew(() =>
      {
            ...
      }
});
```

- Dvs. task forker ud og laver selv nye task som evt. kan forke



- Dynamic task parallelism kaldes også "divide and conquer" og "rekursiv decomposition"
    - Dvs. problemet løses ved at dele problemer ned i mindre bidder

- Typisk sker den rekursive forkning i at man man bruger rekursive funktioner
    - Dvs. funktioner som kalder sig selv
- Dynamic Task Parallism giver mening at bruge når den algoritme man vil køre enten:
    - Skal køres på hvert element i en træ-struktur
    - Algoritmen er rekursiv i natur fx QuickSort algoritme (forklar evt. hurtig)
        - Hvor det er ligemeget i hvilken rækkefølgen dataen i strukturen
        - Man kan ikke umiddelbart styre hvilken rækkefølge hvad sker i

Eksempel med træstruktur man vil løbe igennem:

```
public class Tree<T>
{
  public T       Data  { get; set; }
  public Tree<T> Left  { get; set; }
  public Tree<T> Right { get; set; }
}
```

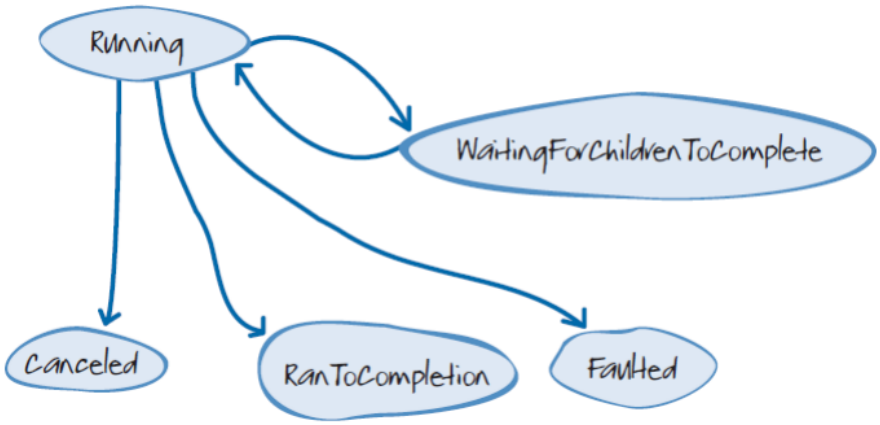
- Hver "gren" indeholder
    - noget data
    - yderligere to instanser af den selv, højre og venstre
- Vi ønsker at rende igennem alt data i træet
- Så kan Dynamic task parallelism pattern bruges

```
static void ParallelWalk<T>(Tree<T> tree, Action<T> action)
{
  if (tree == null) return;
  var t1 = Task.Factory.StartNew(
            () => action(tree.Data));
  var t2 = Task.Factory.StartNew(
            () => ParallelWalk(tree.Left, action));
  var t3 = Task.Factory.StartNew(
            () => ParallelWalk(tree.Right, action));

  Task.WaitAll(t1, t2, t3);
}
```

- ParallelWalk funktionen går hele træstrukturen igennem
- Hvis tree er null så stopper træstrukturen
- Ellers laves en task for både den action
- Og en task som kalder sig selv (rekursiv) på hver sin græn i tret

- I dette eksempel bliver antallet af tasks 3 gange størrelsen af træet.
- Kunne give oversubscription problemer hvis vi lavede manuelt tråde
- Men da vi bruger tasks som bruger threadpool og som run-time scheduleren bedre kan håndtere, så kand en tage højde for sådan nogle ting
- **Man kan bruge Option AttachedToParent når man laver en task**
    - På denne måde vil jeg parent task stå i en WaitingForChildrenToComplete stadie, inden det går i "RanToCompletion" som på figuren på næste side.



Pipelines:

- Pipelines pattern bruges når man har flere forskellige stadier af kode eller algoritme som skal køres på fx collection
    - Hvert stadie kan være én Task i .NET
- MEN det er vigtigt at styre sekvensen af koden
    - For at styre sekvensen indsættes buffers mellem hver task
    - I bufferen bliver der sat noget ind fra det tidligere stadie, og den næste stadie tager noget ud af bufferen og udfører dens opgave
    - Rækkefølgen bevares, men hvert stadie kan køres concurrent

Forklares bedst med et eksempel:

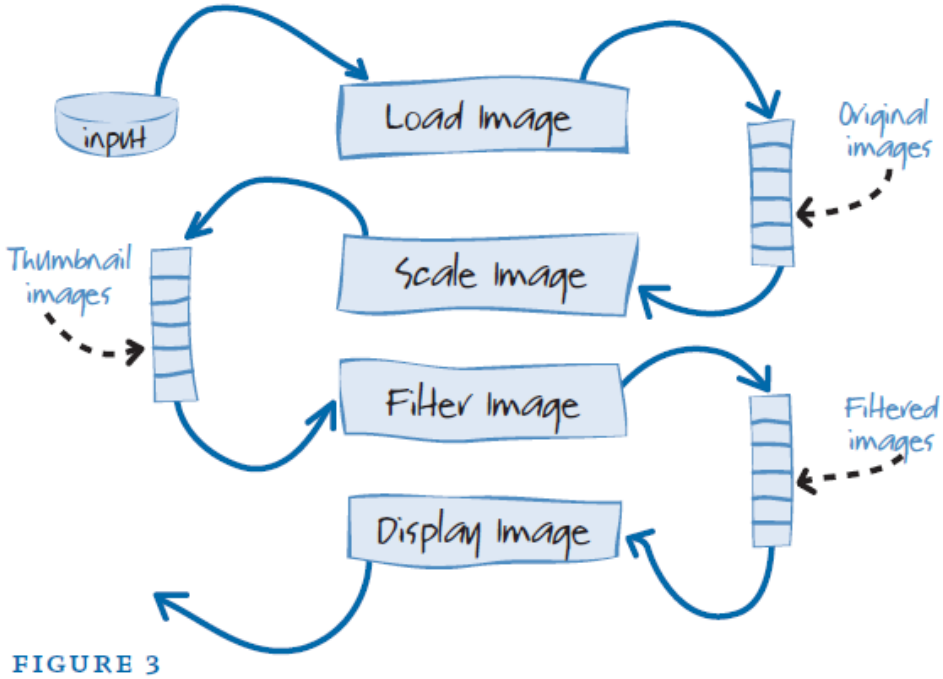


FIGURE 3
Image pipeline

- Et billede skal loades skaleres, dereefter filtreres og så vises
- Rækkefølgen er vigtig

Sekventielt:

1 billede af gangen, først efter Display Image er færdig kan et nyt billede loades

Pipelining:

- Nyt billede kan indlæses så snart det første billede er i “scale image” task’en

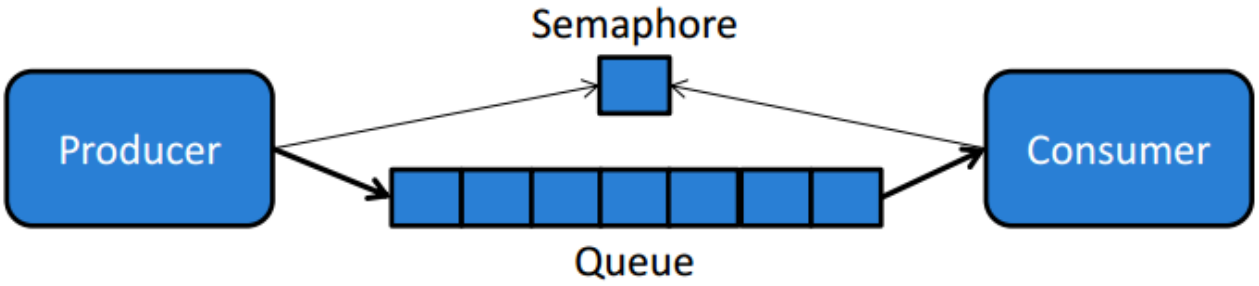
```
  var f = new TaskFactory(TaskCreationOptions.LongRunning,
                          TaskContinuationOptions.None);
  // ...

  var loadTask = f.StartNew(() =>
                   LoadPipelinedImages(fileNames, sourceDir,
                                       originalImages, ...));
  var scaleTask = f.StartNew(() =>
                   ScalePipelinedImages(originalImages,
                                        thumbnailImages, ...));
  var filterTask = f.StartNew(() =>
                   FilterPipelinedImages(thumbnailImages,
                                         filteredImages, ...));
  var displayTask = f.StartNew(() =>
                   DisplayPipelinedImages(
                       filteredImages.GetConsumingEnumerable(),
                       displayFn, ...));

  Task.WaitAll(loadTask, scaleTask, filterTask, displayTask);
}
finally
{
  // ... release handles to unmanaged resources ...
}
```

- Her vises hvordan hvert task dvs. stadie oprettes
- LongRunning option er et hint til scheduleren at den måske bør lave nye tråde i stedet for at bruge af thread pool
- Hver task får defineret en source collection og en destination collection

- Vores collection skal sørge for at blokere en consumer tråd hvis den er tom
- Den skal sørge for at blokere en producer tråd hvis den er fyldt
- Det gøres med låsning, som vi kender det fra ISU 3.semester med semaphore.
- Kan kodes manuelt eller BlockingCollection<t> i .Net.



- Denne queue kan håndtere flere producers, eller flere consumers
- I .Net findes en collection ved navn BlockingCollection<t> som har samme funktionalitet, bare bedre og hurtigere

- Problemer med Pipelines:
- Nogle tasks bliver hurtigere færdig, så bliver køen fyldt nogle steder, og begrænses af denne