Föreläsning 22

Tobias Wrigstad

Parallellprogrammering, trådar – en försmak

Visste du att…

- OS X använder sig av Emacs-style kommandon för att navigera i textrutor

  Ctrl-a går till början av en rad

  Ctrl-e går till slutet av raden

  Ctrl-k klipper ut en rad

  Ctrl-y klistrar in den igen

  Ctrl-p, Ctrl-n går till föregående rad, nästa rad, etc.

- *(Jag skriver detta i Keynote på OS X och använder ett tangentbord utan piltangenter)*

  Som någon som arbetar med text är mina händers ergonomi viktig!

```
x.f = 5;  y.f = 4;  System.out.println(x.f);
```

Aliaseringsproblemet

```
x.f = 5;  y.f = 4;  System.out.println(x.f);
```

- För att avgöra vad detta program gör måste vi veta något om variablerna `x` och `y`

  Om de avser samma objekt (dvs. `x == y`, de är alias, de aliaserar varandra) — 4

  Om de inte avser samma objekt — 5

- Att avgöra om `x == y` är möjligt kräver ofta ”non-local reasoning”, i värsta fall måste vi studera hela programmet noggrant!

Program som gör många saker samtidigt

- Concurrency (”samtidighet”)

  Hantera program med asynkrona händelser

  Ex. en webbserver behöver hantera många samtidiga *page requests*

  Ex. en filkopieringsdialog behöver kunna kopiera och lyssna på avbryt-knappen samtidigt

- Parallellism

  Effektivisera lösningen av problem genom att använda flera processorer samtidigt

  Nyckelord: *throughput (genomströmning)*, *latency (latens)*, *energieffektivitet*

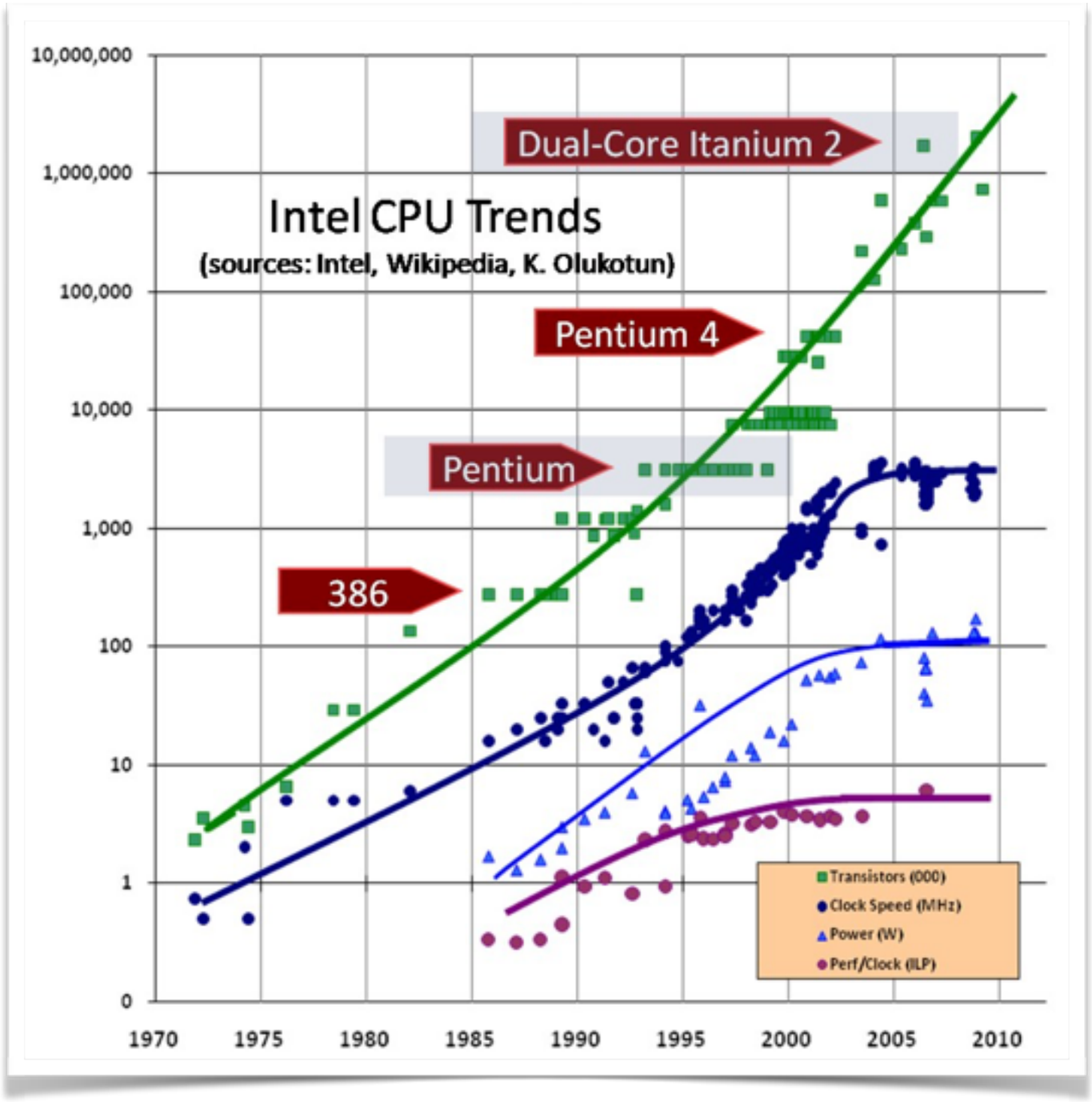
Concurrency

- Utanför denna kurs — se vidare ”operativsystemskursen” som kommer efter denna
- Handlar om hur program struktureras

  Ex. på serversidan finns ett objekt för varje uppkopplad användare som ”har eget liv” och som kan ta emot begäran när som helst

  Ex. dela upp programmet i två uppgifter som utförs samtidigt, en som kopierar och en som väntar på input från användaren — någon form av kommunikation är nödvändig mellan dem

Behovet av parallellprogrammering



http://www.extremetech.com/wp-content/uploads/2012/02/CPU-Scaling.jpg

Behovet av parallelprogrammering i ett nötskal

- Före ca 2005:

  ”Bästa sättet att optimera ett programs prestanda är att vänta ett år”

- Efter 2005:

  Program måste skrivas så att de kan dra nytta av framtida datorers parallella kapacitet

  Portabel prestanda…?

- Vi snuddar bara vid detta nu, men kommande kurser fördjupar

  Jag förklarar inte problemen med task-parallelism och Amdahls lag, etc.

  UPMARC-programmet i Concurrent- och parallellprogrammering (Master/IT-spår)

```
x.f = 5;  y.f = 4;  System.out.println(x.f);
```

CPU 1

CPU 2

```
x.f = 5;  System.out.println(x.f);        y.f = 4;
```

Aliaseringsproblemet + Concurrency = 

Task 1

```
x.f = 5;  System.out.println(x.f);
```

Task 2

```
y.f = 4;
```

- … måste vi veta något om variablerna x och y och schemaläggningen av task 1 & 2

  Om x != y är resultatet alltid — 5

  Om x == y beror resultatet *på om Task 1 utförs före, efter eller samtidigt med Task 2!*

- Att avgöra om x == y är möjligt kräver ofta ”non-local reasoning”
- *… och motsvarande för att avgöra ordningen mellan Task 1 och Task 2!*

Trådkonceptet

- Ett enkelt (och därför populärt) sätt att lägga till stöd för concurrency/parallelism i existerande programspråk är ”*trådar*”

  I Java kan man skapa ett objekt av klassen `Thread` och anropa metoden `start()`

  Då körs klassens metod `run()` ”parallellt” med det övriga programmet

- Kommunikation mellan trådar kan ske t.ex. genom att man delar objekt

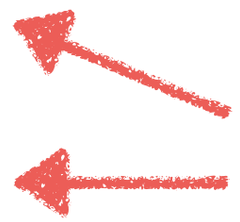
  Om två eller fler trådar kan läsa/skriva ett objekt krävs någon form av *synkronisering* — så att man inte skriver över varandras värden av misstag

  (OS-kursen går in på detta i mer detalj)

Demo.java

```
public class Demo extends Thread {
    private final int id;
    Demo(final int id) {
        this.id = id;
    }
    public void run() {
        for (int i = 0; i < 10; ++i) {
            System.out.println(this.id);
        }
    }
    public static void main(String[] args) {
        new Demo(1).start();
        new Demo(2).start();
        new Demo(3).start();
        new Demo(4).start();
    }
}
```

```
$ javac Demo.java
  $ java Demo
      1
      1
      1
      1
      1
      1
      1
      1
      1
      1
      4
      4
      4
      4
      4
      4
      4
      4
      4
      3
      3
      3
      3
      3
      3
      3
      3
      3
      3
      2
      2
      2
      2
      2
      2
      2
      2
      2
      2
```



Kunde lika gärna ha kommit 1, 2, 3 och 4 om vartannat

DataRace.java

```
public class DataRace extends Thread {
    private final Cell c;
    public boolean done = false;
    DataRace(final Cell c) {
        this.c = c;
    }
    public void run() {
        for (int i = 0; i < 10000; ++i) {
            c.inc();
        }
        this.done = true;
    }
    public static void main(String[] args) {
        Cell c = new Cell();
        DataRace d1 = new DataRace(c); d1.start();
        DataRace d2 = new DataRace(c); d2.start();
        DataRace d3 = new DataRace(c); d3.start();
        DataRace d4 = new DataRace(c); d4.start();
        while (d1.done == false);
        while (d2.done == false);
        while (d3.done == false);
        while (d4.done == false);
        System.out.println(c.value);
    }
}
```

```
class Cell {
    int value = 0;
    public void inc() {
        this.value = this.value + 1;
    }
}
```

```
$ javac DataRace.java
$ java DataRace
24936
$ java DataRace
24261
$ java DataRace
23881
$ java DataRace
23013
$ java DataRace
24892
```

På denna kurs ”Task Parallelism”

- Parallellisera (del av) ett program genom att identifiera *uppgifter* som kan utföras parallellt

  Förtingliga uppgifterna genom att göra dem till objekt

  Målet: berätta för en *schemaläggare* vilka uppgifter som skall göras och deras beroende mellan varandra

  Schemaläggaren bestämmer i vilken utsträckning uppgifterna faktiskt utförs parallellt — detta beror på datorns resurser och hur de används vid varje aktuellt tillfälle

- Under huven finns ett antal trådar men målet här är att **undvika** att se dem

  Trådar och lås är en extremt komplicerad programmeringsmodell (dålig!)

  Som programmerare vet vi i regel inte bättre än systemets schemaläggare!

Om lektionen på tisdag

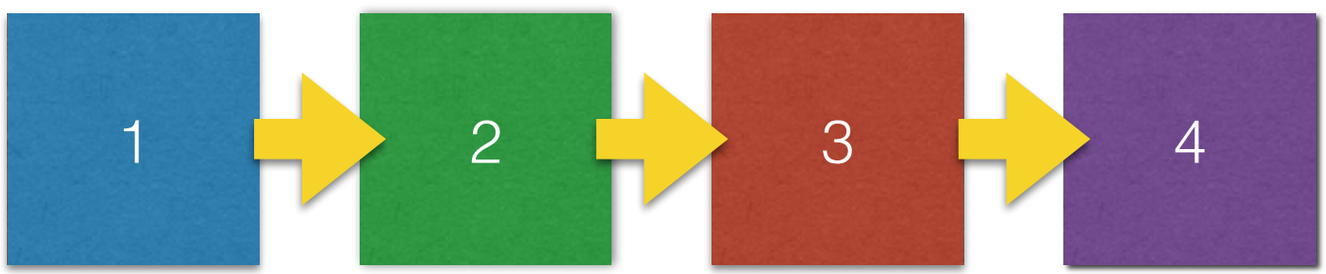
Vi delar ut ett sekventiellt program

Målet är att parallellisera det med task-parallellism som visas denna timme

Mål på nivå 3–5 går att redovisa på detta program

Task-parallellism

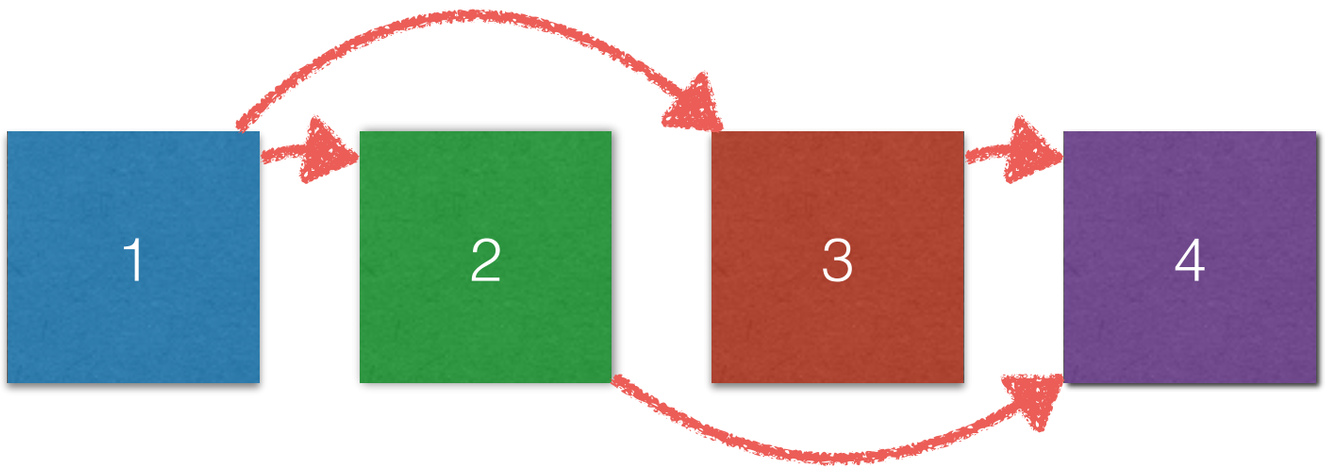
```
my_function(…) {
1  x, y = frob(z);
2  foo₁ = bar(x);
3  foo₂ = bar(y);
4  foo₃ = quux(foo₁, foo₂);
}
```



Den sekventiella hjärnan ser…

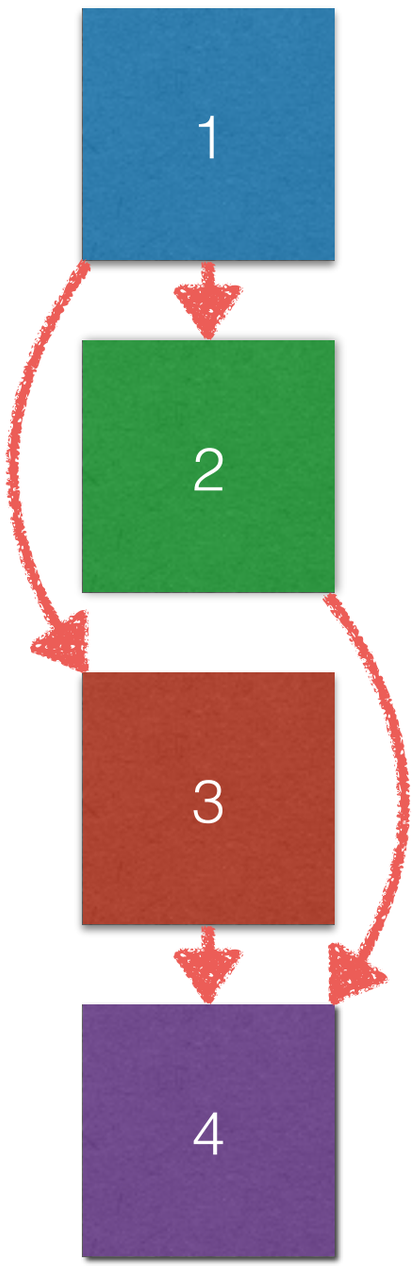
Task-parallellism

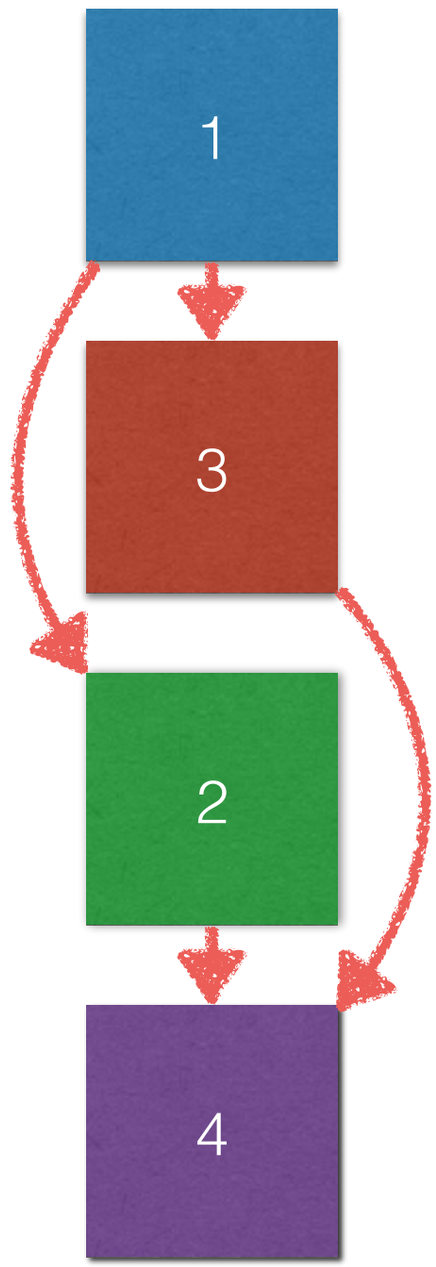
```
my_function(…) {
1     x, y = frob(z);
2     foo1 = bar(x);
3     foo2 = bar(y);
4     foo3 = quux(foo1, foo2);
}
```

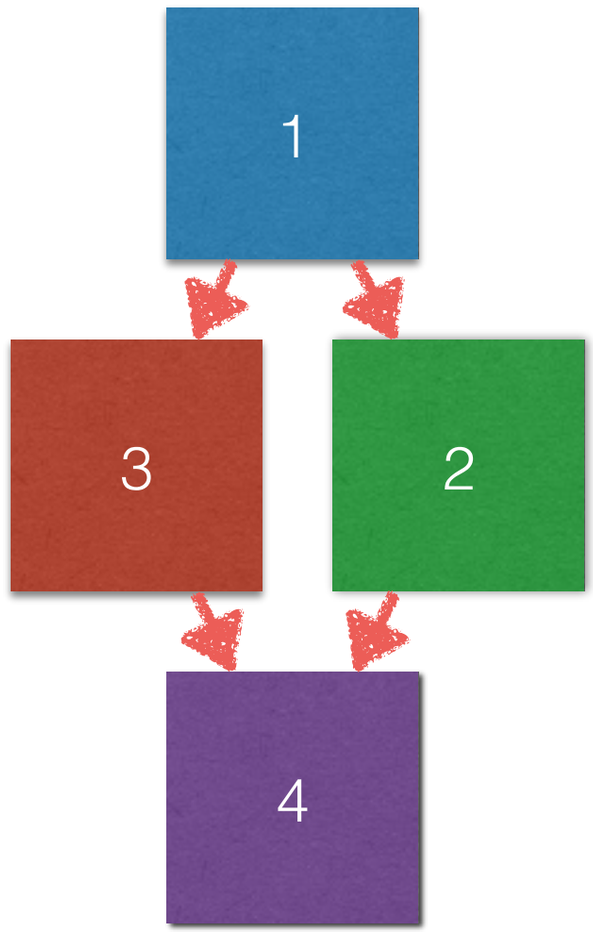


De faktiska beroendena

Hur kan vi schemalägga dem?







Task-parallellism

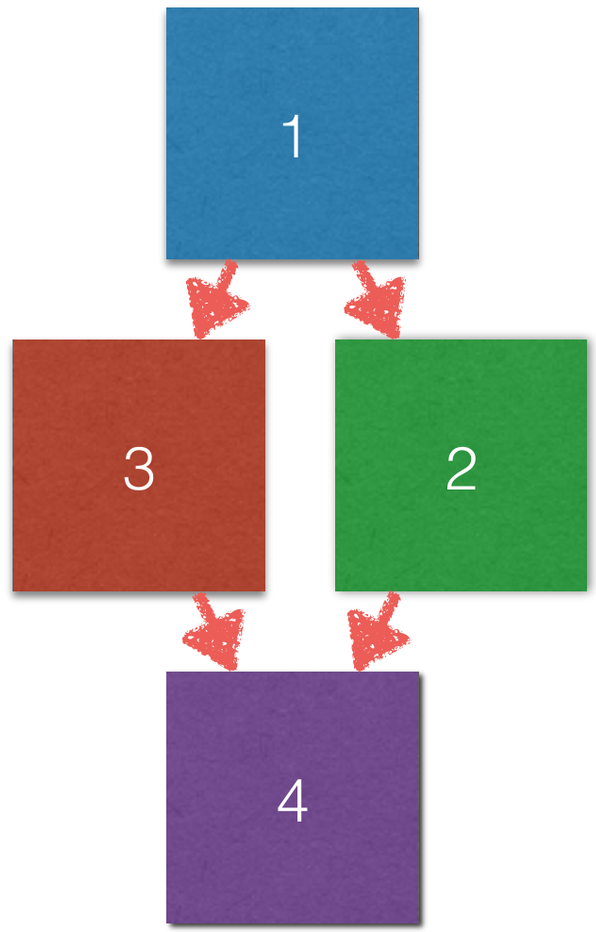
- Utför operationen på 3 tidsenheter istället för 4

  Minskad **latency**

- Om vi har 4 processorer kan vi utföra 3 operationer på 3 tidsenheter var, med latency 4

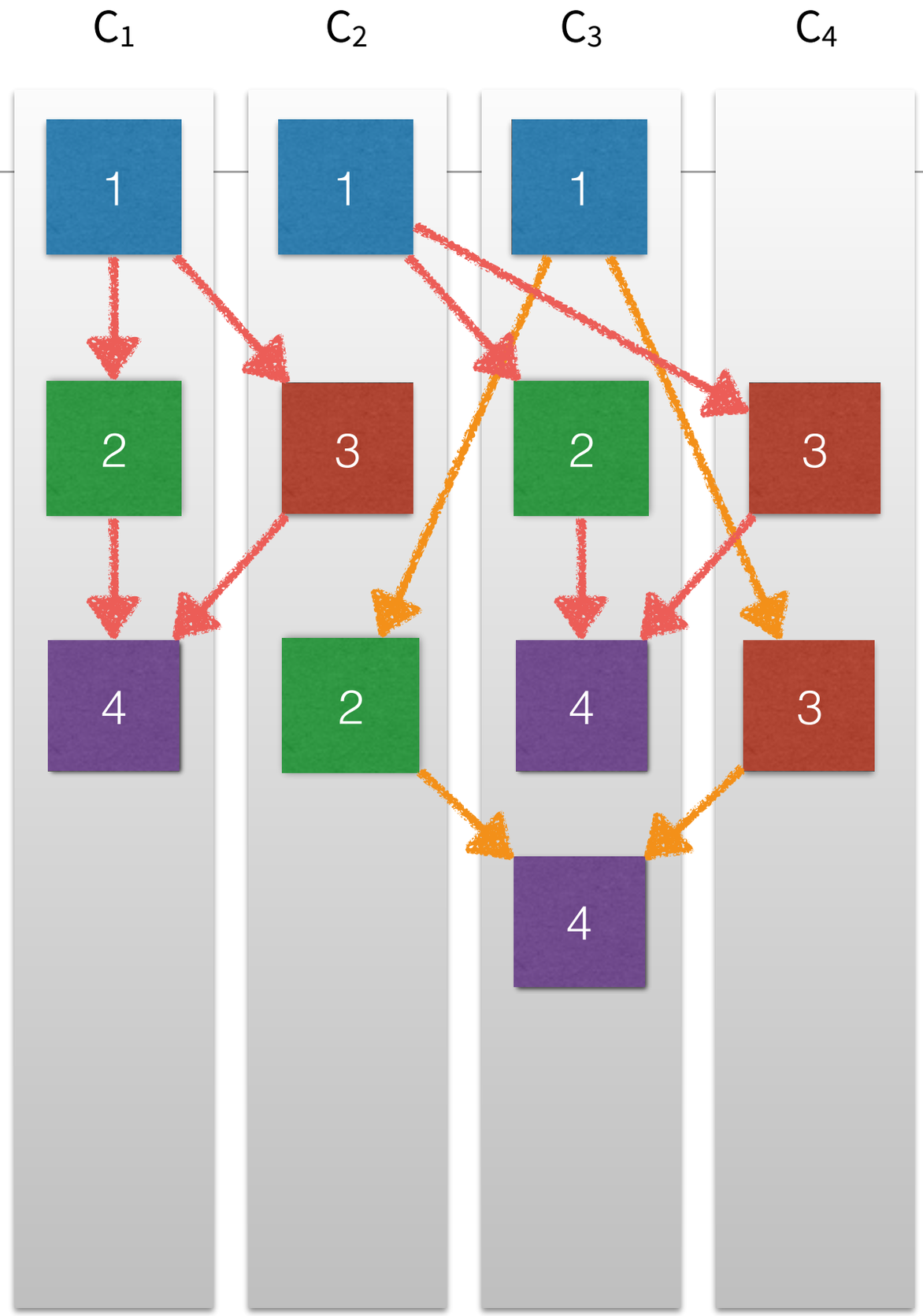
  Ökad **throughput**

Hur då?

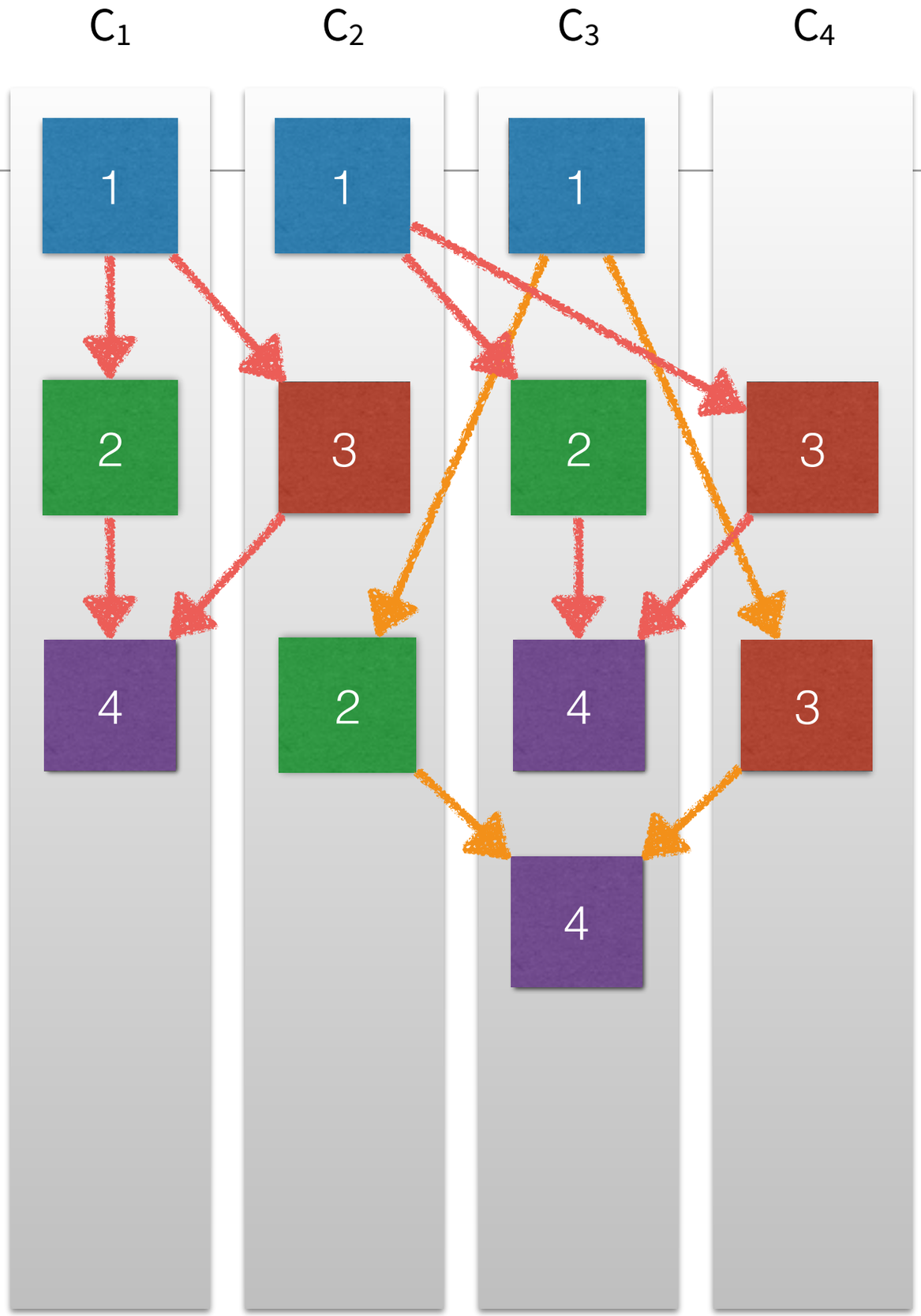


Svar

3 tidsenheter behövs per CPU, men vi kan inte utföra arbetet på kortare tid än 4 tidsenheter av ”wall clock time”, på grund av beroenden mellan uppgifterna



Prestanda



Work: 12

Span: 4

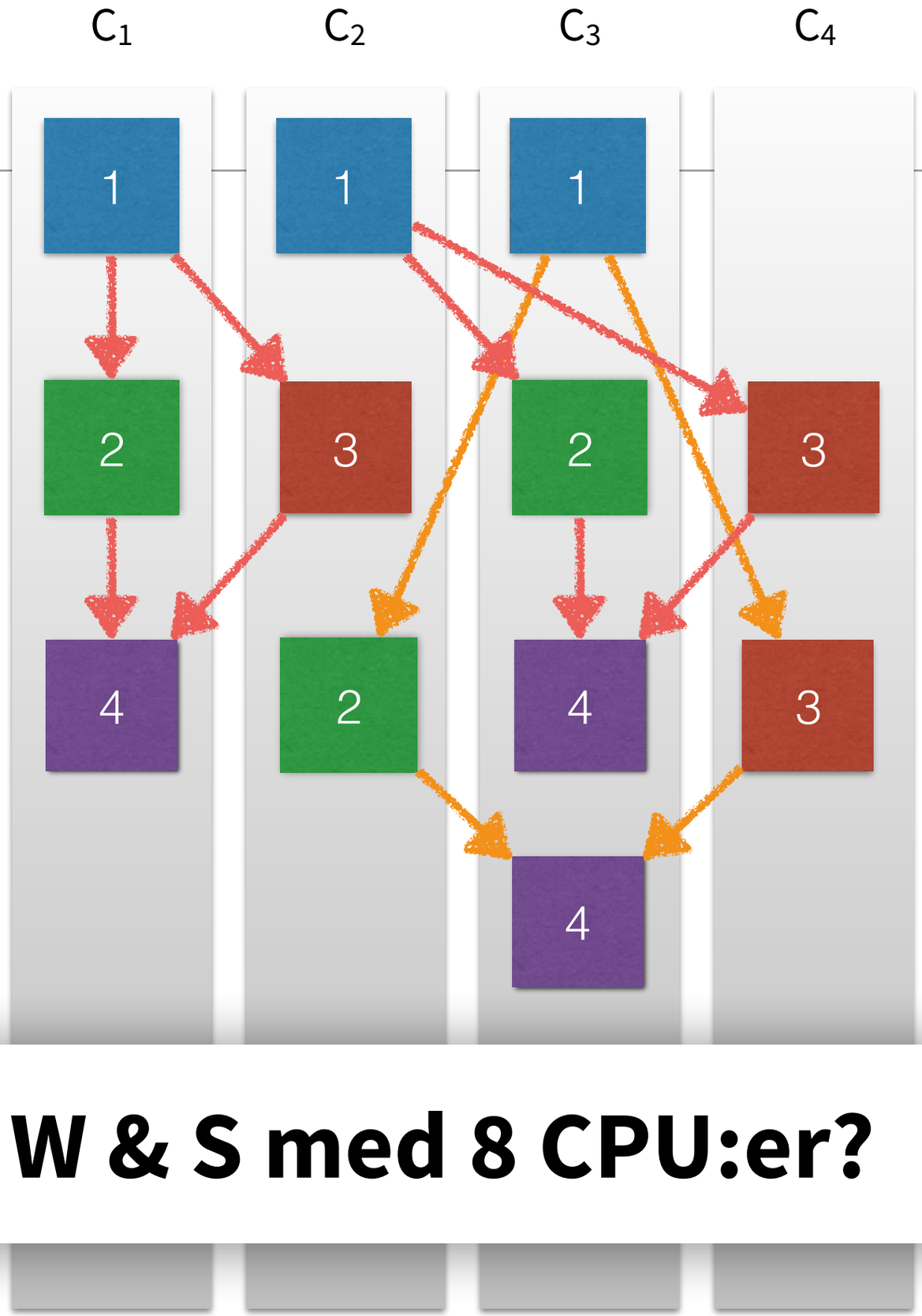
Work

T_1 — tiden det skulle ta med bara en CPU

Span

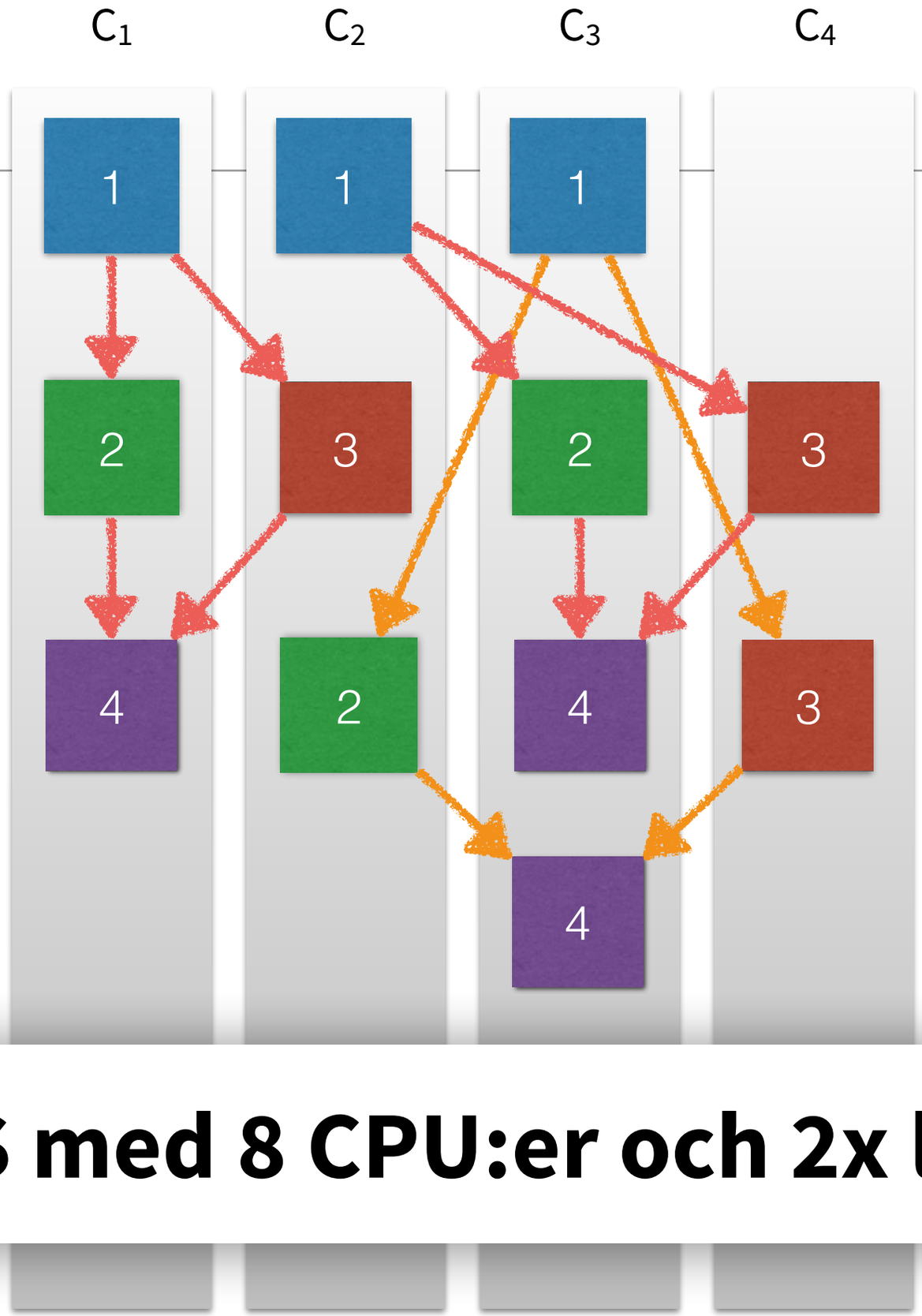
T_∞ — tiden det skulle ta med oändligt många CPU:er

Work & Span



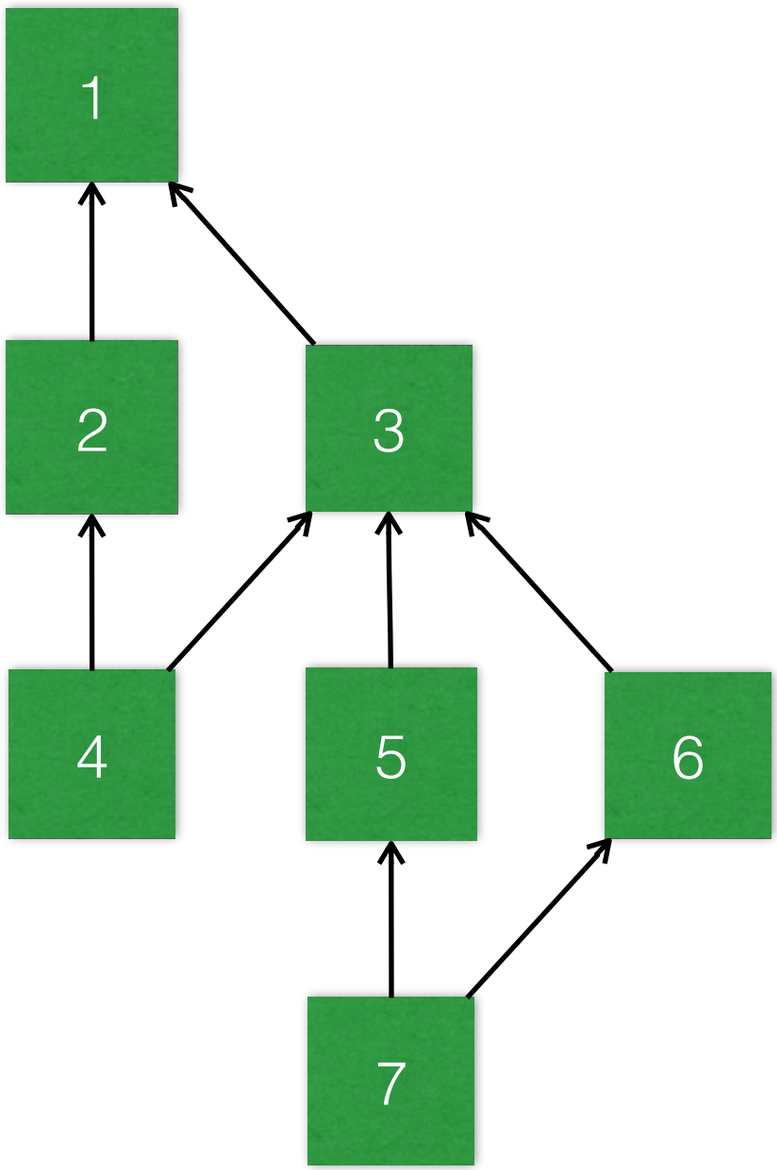
W & S med 8 CPU:er?

Work & Span



W & S med 8 CPU:er och 2x last?

Tasks formar en riktad acyklisk graf



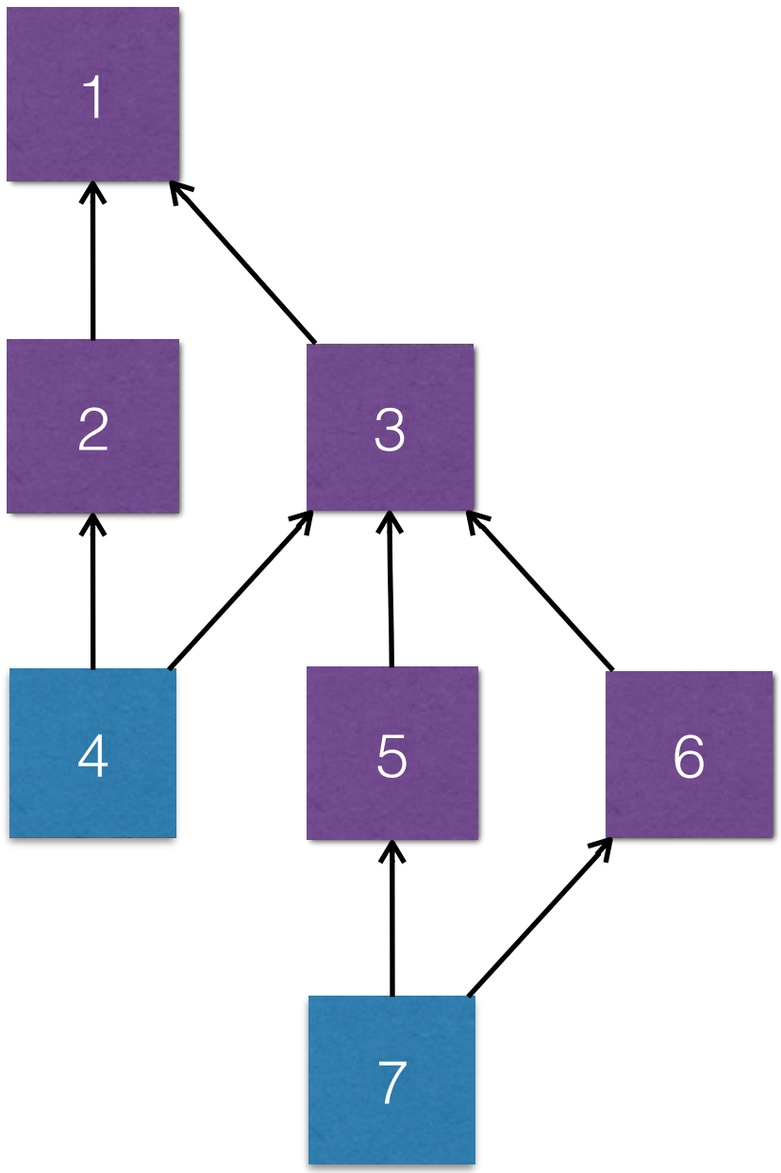
Noder är tasks

Kanter är beroenden (1 beror på 2 och 3)

Vi kan enbart utföra task som inte har beroenden

Vilka tasks kan utföras?

En tasks tillstånd



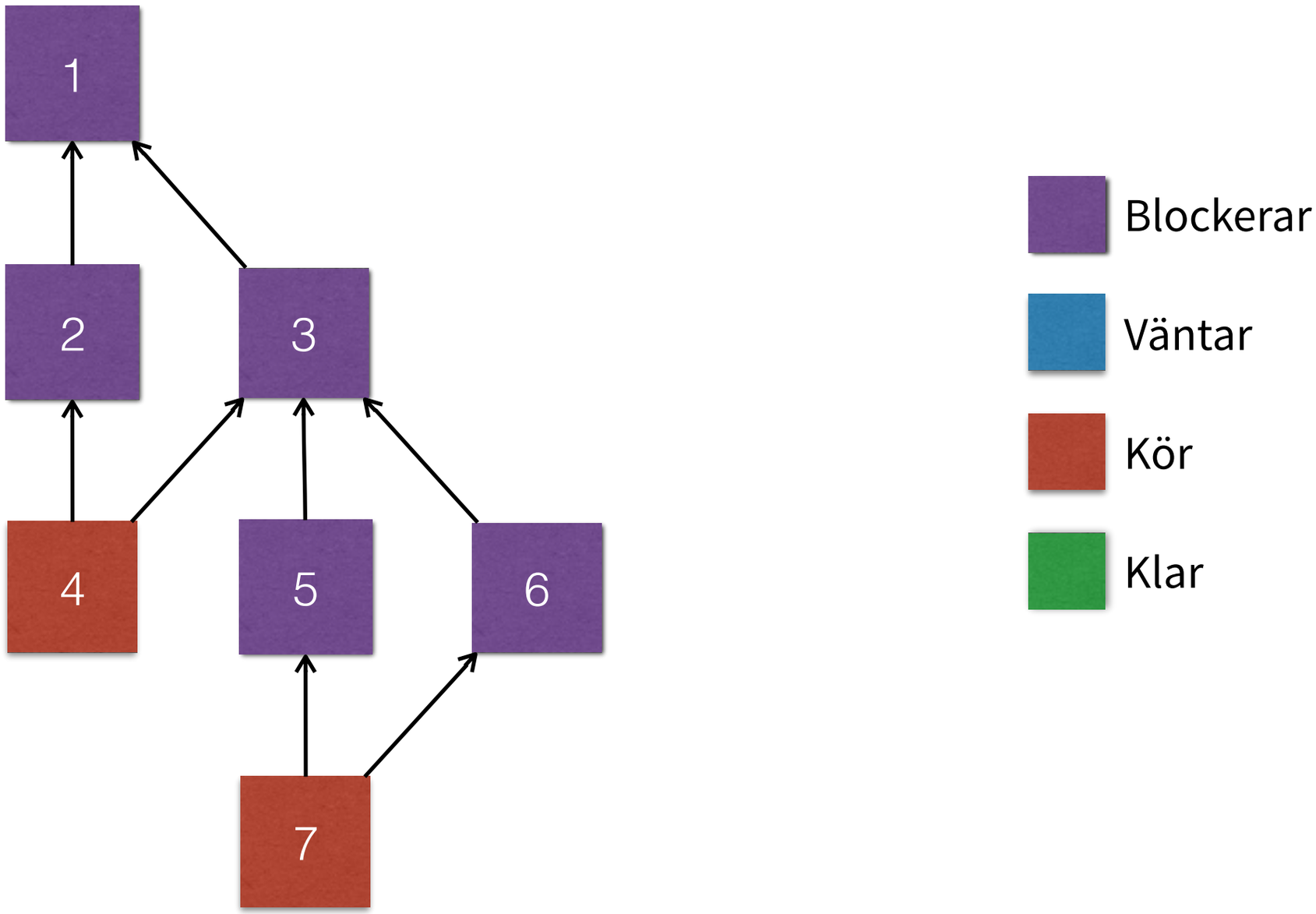
Blockerar

Väntar

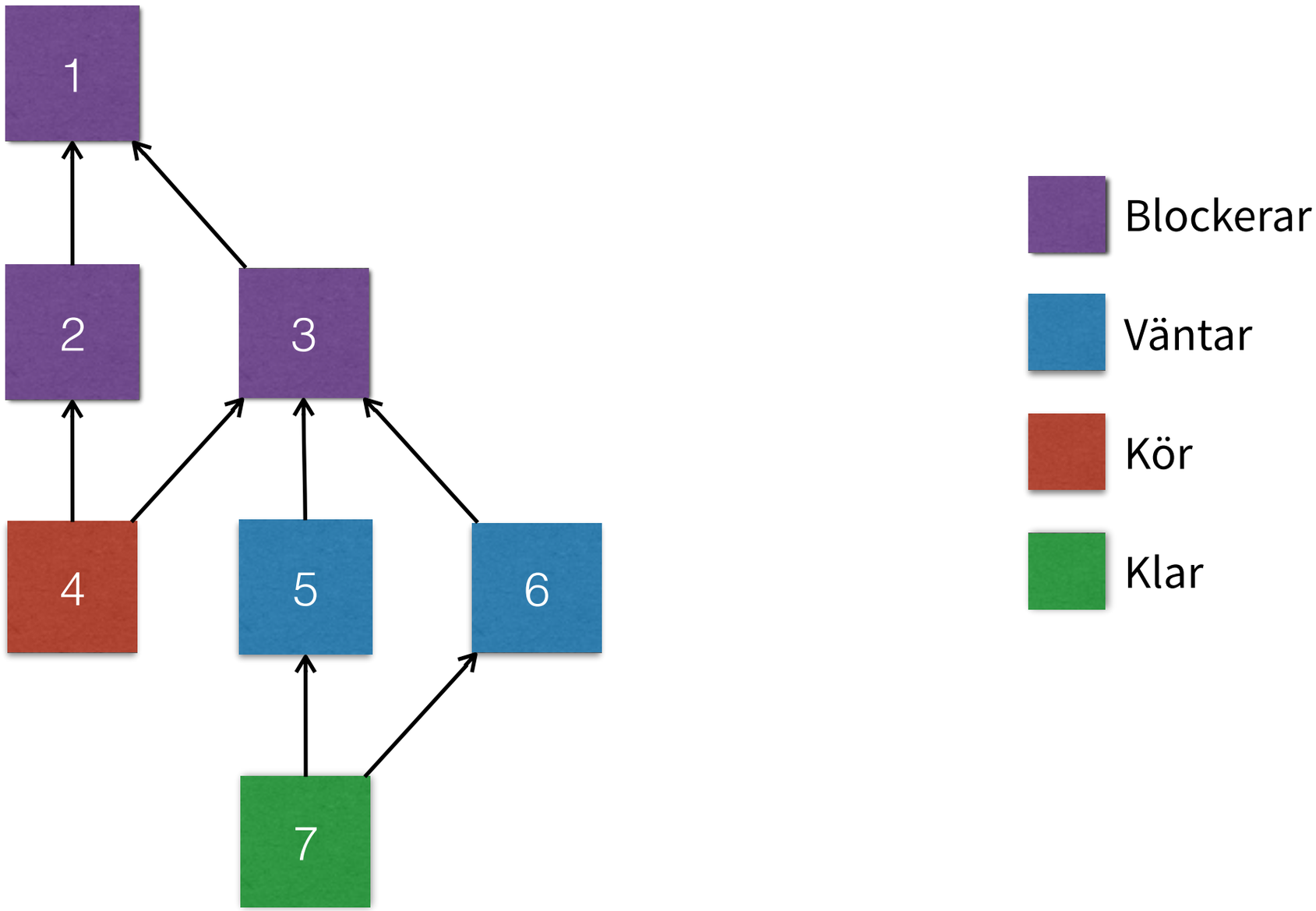
Kör

Klar

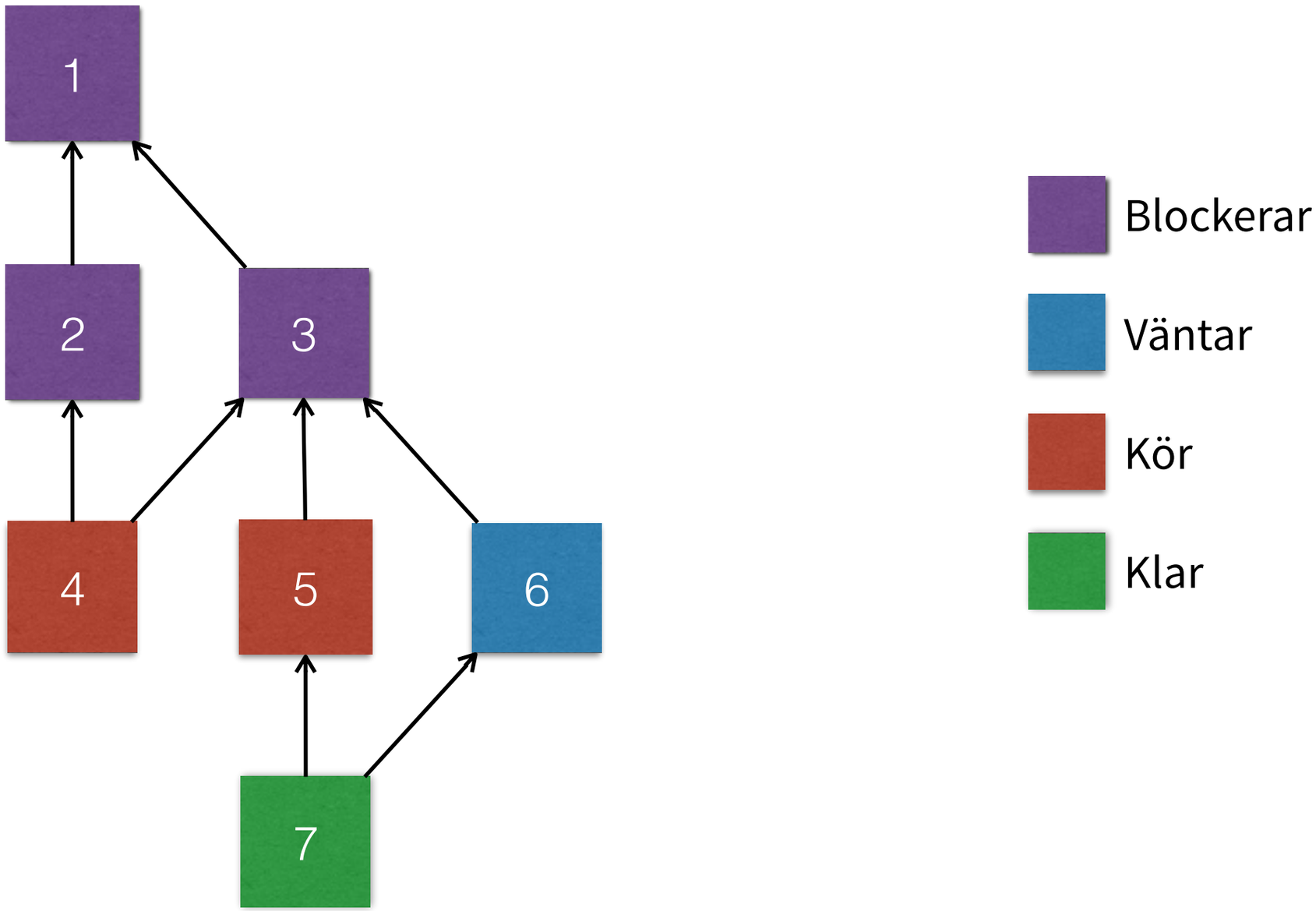
Vi kör!



Vi kör!



Vi kör!



Exempel: parallellisera en map

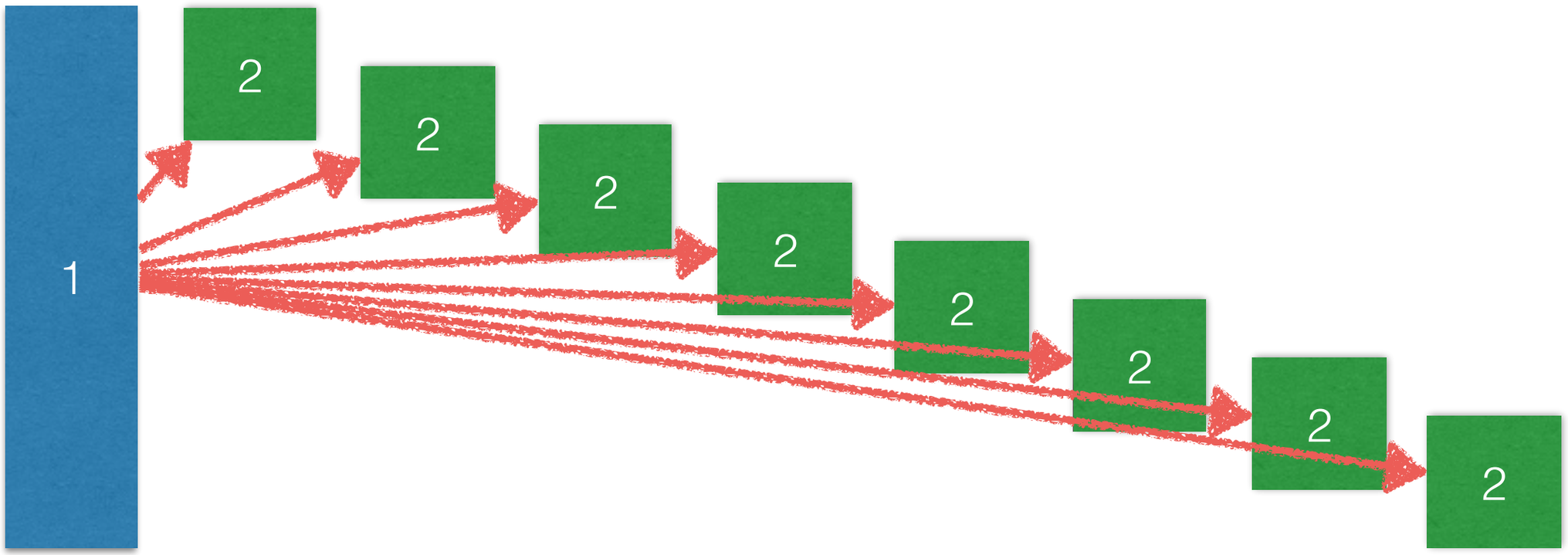
- Var finns våra tasks?

```
[1,2,3,4,5,6,7,8].collect { |e| e * e }
```

Exempel: parallellisera en map

```
[1,2,3,4,5,6,7,8].collect { |e| e * e }
```

- Var finns våra uppgifter — varje e^2, plus att **skapa** alla dessa uppgiftsobjekt vid körning

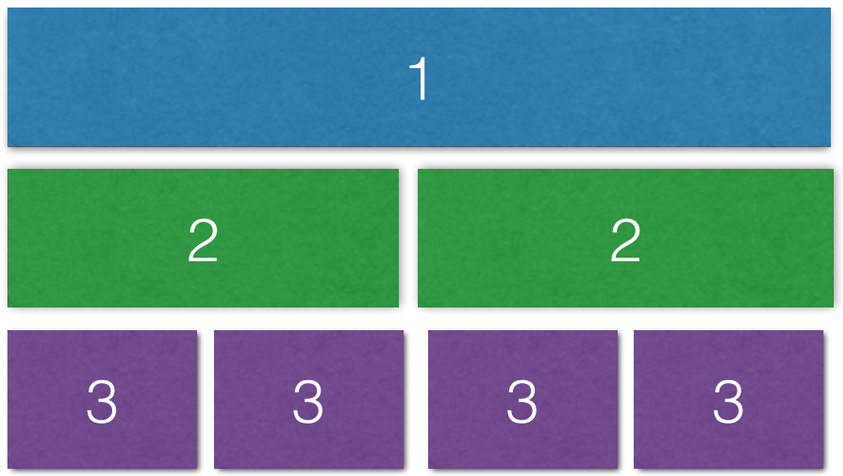


Exempel: parallellisera en map

```
[1,2,3,4,5,6,7,8].collect { |e| e * e }
```

- Var finns våra uppgifter — varje e^2, plus att **skapa** alla dessa uppgiftsobjekt vid körning

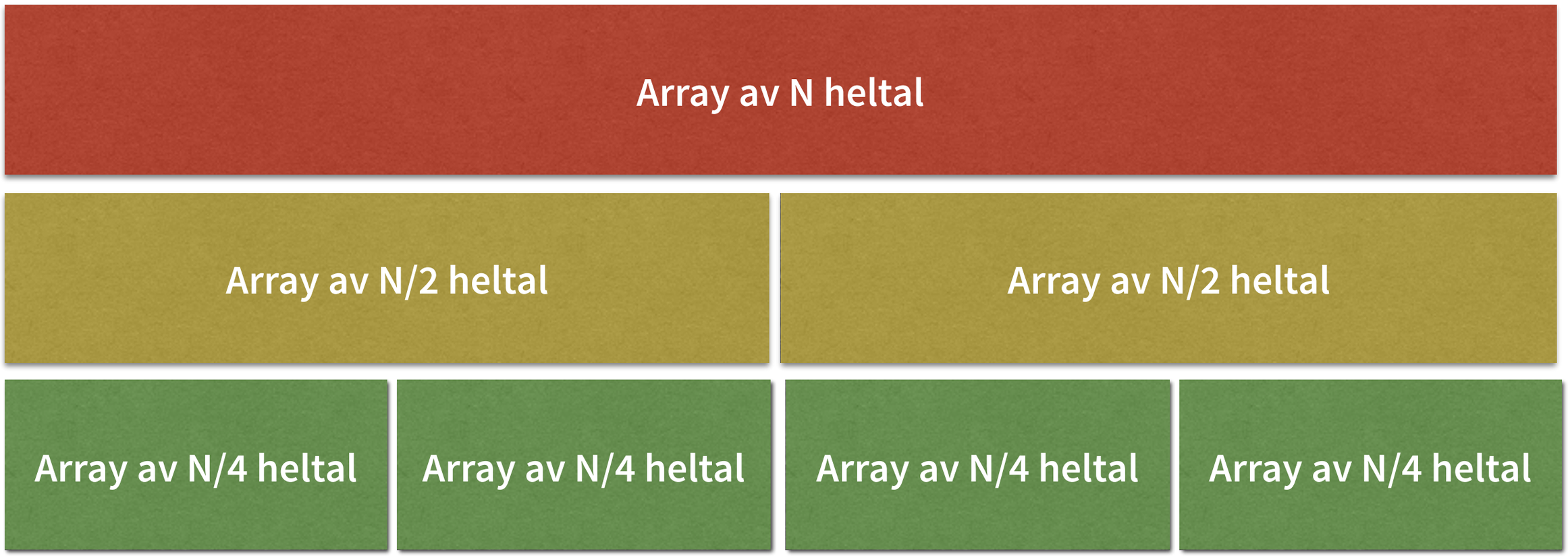
```
def map(list:[Int]) : [Int] {
  if list.size() == 1
  then [list.first * list.first]
  else {
     fst, snd = list.split();
     a = async map(fst);
     b = async map(snd);
     a ++ b;
  }
}
```



Summera tal i en array

- **Input**: en lång array av heltal
- **Output**: summan av alla heltal
- Hur beräknar vi output parallellt?

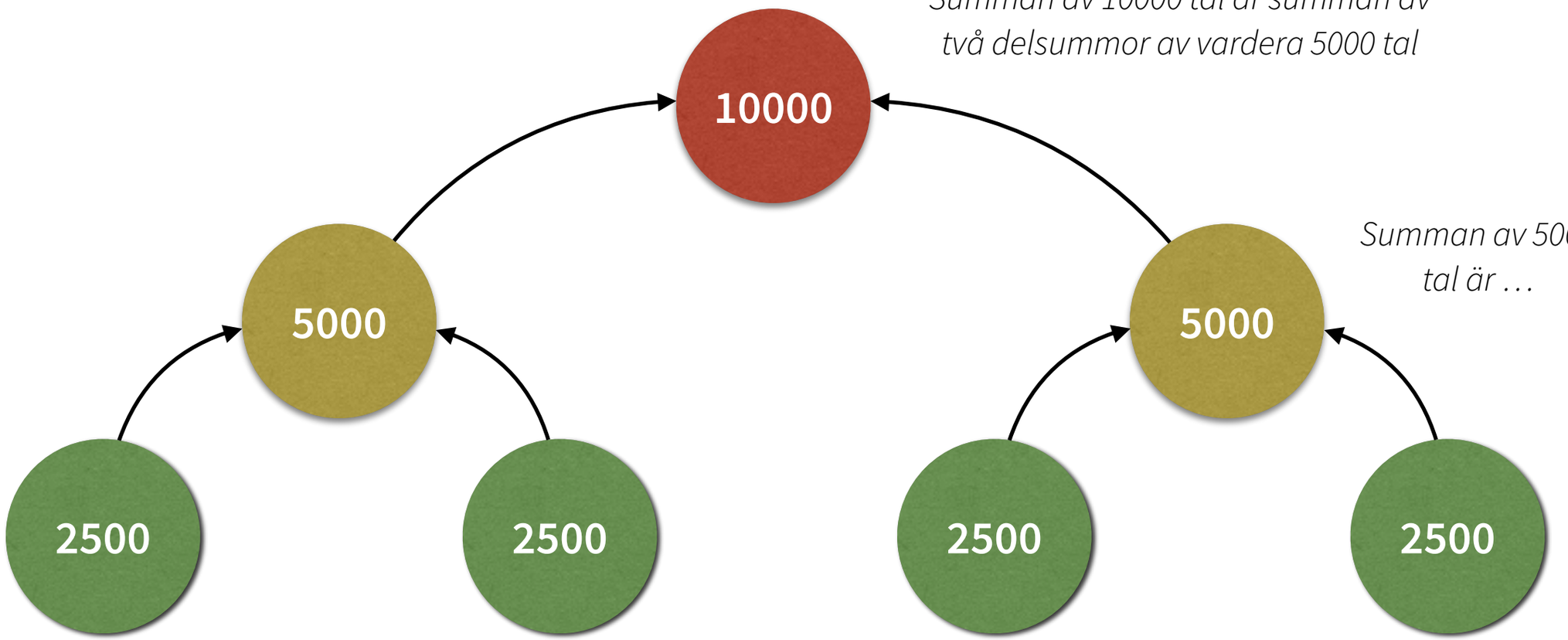
Summera med divide-and-conquer



. . .

Bryt ned tills vi har ett lagom antal tasks! (Vad är lagom?)

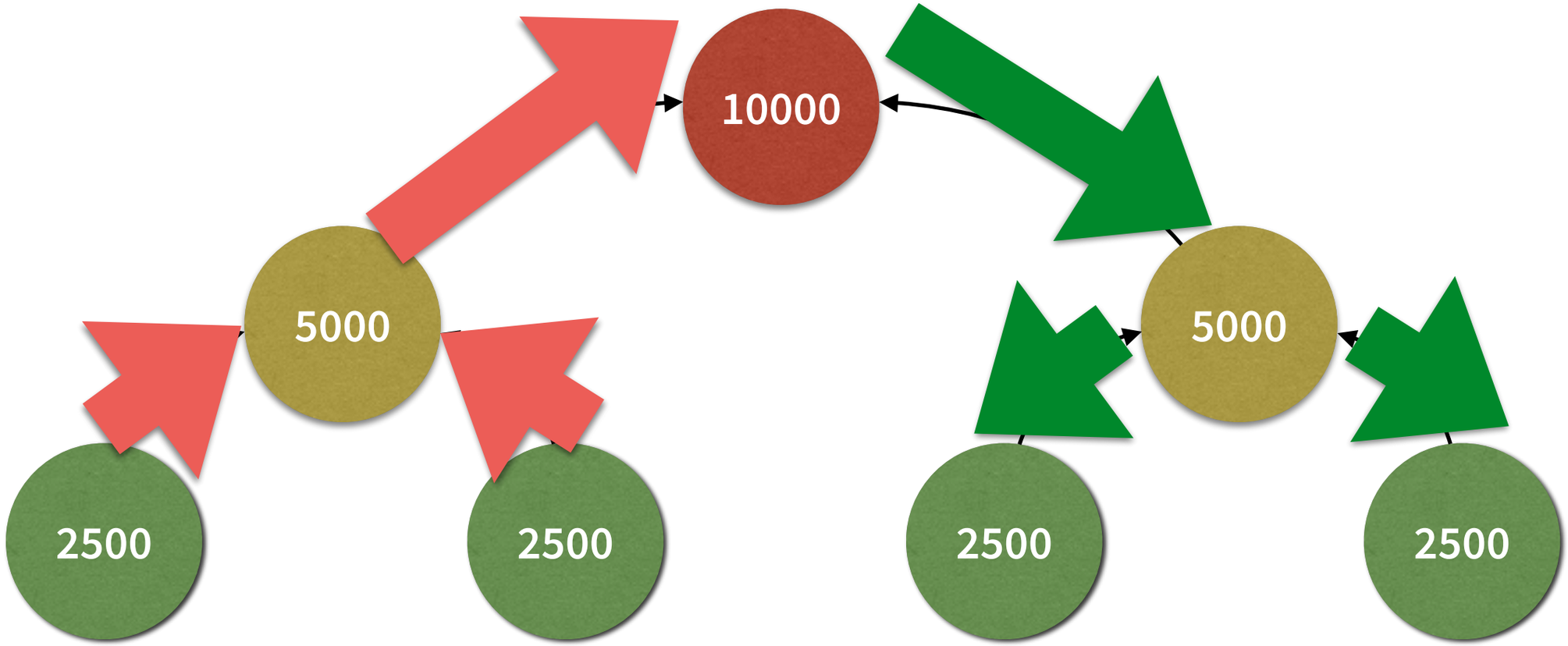
Acyklisk graf av beroenden



. . .

Bryt ned tills vi har ett lagom antal tasks! (Vad är lagom?)

Divide and Conquer — Fork/Join



. . .

Fork until we have a suitable number of tasks, perform them and join to ”unblock” waiting tasks

Summera en array i Java

Fork/Join

Recursive Task

```java
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

public class Sum extends RecursiveTask<Long> {
    static final int SEQUENTIAL_THRESHOLD = 1; // No cut-of
    int low;
    int high;
    int[] array;

    Sum(int[] arr, int lo, int hi) {
        this.array = arr;
        this.low   = lo;
        this.high  = hi;
    }

    protected Long compute() {
        if (high - low <= SEQUENTIAL_THRESHOLD) {
            long sum = 0;
            for(int i=low; i < high; ++i) sum += array[i];
            return sum;
        } else {
            int mid = low + (high - low) / 2;
            Sum left  = new Sum(array, low, mid);
            Sum right = new Sum(array, mid, high);
            left.fork();
            right.fork();
            long rightAns = right.join();
            long leftAns  = left.join();
            return leftAns + rightAns;
        }
    }

    static long sumArray(int[] array) {
        return Globals.fjPool.invoke(new Sum(array,0,array.length));
    }
}
```

```java
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

public class FJ {
    public static void main(String[] args) {
        System.out.println(Sum.sumArray(new int[] { … }));
    }
}

public class Globals {
    static ForkJoinPool fjPool = new ForkJoinPool();
}
```

```
$ javac FJ.java
$ java FJ
<large number>
```

Notera att detta program inte är speciellt objektorienterat!

```java
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

public class Sum extends RecursiveTask<Long> {
    static final int SEQUENTIAL_THRESHOLD = 1; // No cut-of
    int low;
    int high;
    int[] array;

    Sum(int[] arr, int lo, int hi) {
        this.array = arr;
        this.low   = lo;
        this.high  = hi;
    }

    protected Long compute() {
        if (high - low <= SEQUENTIAL_THRESHOLD) {
            long sum = 0;
            for(int i=low; i < high; ++i) sum += array[i];
            return sum;
        } else {
            int mid = low + (high - low) / 2;
            Sum left  = new Sum(array, low, mid);
            Sum right = new Sum(array, mid, high);
            left.fork();
            long rightAns = right.compute();
            long leftAns  = left.join();
            return leftAns + rightAns;
        }
    }

    static long sumArray(int[] array) {
        return Globals.fjPool.invoke(new Sum(array,0,array.length));
    }
}
```

```java
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

public class FJ {
    public static void main(String[] args) {
        System.out.println(Sum.sumArray(new int[] { … }));
    }
}

public class Globals {
    static ForkJoinPool fjPool = new ForkJoinPool();
}
```

```
$ javac FJ.java
$ java FJ
<large number>
```

Notera att detta program inte är speciellt objektorienterat!

Undantagshantering

Undantagshantering

- När något går fel i Java *genereras* och *kastas* ett *undantag*
- Ett undantagsobjekt beskriver

  Typen av fel

  Var i programmet felet uppstod

  Vägen programmet tagit för att komma till felet i form av en stacktrace
- När ett undantag kastas avbryts exekveringen

  Kontrollen flyttas till *närmast omslutande undantagshanterare för denna typ av fel*

  Om det inte finns en sådan termineras programmet
- Untantagshanterare implementeras med hjälp av den s.k. *try-catch-satsen*

Att generera och kasta ett undantag

- En mängd fördefinierade undantag finns i Javas standardbibliotek:

  RuntimeException, IllegalArgumentException, ArrayIndexOutOfBoundsException, ArithmeticException, ClassCastException, …

- Dessa är vanliga klasser – att generera ett undantag = instantiering

  new RuntimeException();

  new IllegalArgumentException(); etc.

- Att kasta ett undantag görs med nyckelordet **throw** – normalt bakas generering och kastande av undantag ihop:

  throw new RuntimeException();

- …men även throw x; om x är en variabel som innehåller ett untantag

Att definiera egna undantag

- Lämpligt att göra i klasser för att förenkla felhantering

  Koden nedan definierar ett nytt undantag genom att ärva av undantagsklassen `Exception` som sedan kan kastas som vanligt

```
/**
 * @author Tobias Wrigstad (tobias.wrigstad@it.uu.se)
 * @date 2013-10-20
 */
public class DuplicateElementException extends Exception {
    public DuplicateElementException(String msg) { super(msg); }
}
```

```
throw new DuplicateElementException("Element "
            + e.toString() + " already in the set");
```

Felhanterare: try–catch–(finally)

- Följande kod installerar en felhanterare för all kod som körs i ... – även sådan som är i anropade metoder

```
try {
    ... // code that could fail
} catch(ExceptionType1 e) {
    // code to handle this type of failure
} catch(ExceptionType2 e) {
    // code to handle this type of failure
} finally {
    // code to always run -- fail or nofail
}
```

Vad skrivs ut när `example()` körs?

```
void bar() { int stupid = 1/0; }

void foo() { bar(); System.out.println("!!!"); }

void example() {
    try {
        foo();
    } catch(ArithmeticException e) {
        System.err.println("Do something");
    } catch(Exception e) {
        System.err.println("Do something else");
    } finally {
        System.err.println("Grr arrrgh");
    }
}
```

!!! *skrivs aldrig ut!*

Vad skrivs ut när `example()` körs?

```
void bar() { int stupid = 1/0; }

void foo() { bar(); }

void example() {
    try {
        foo();
    } catch(ArithmeticException e) {
        System.err.println("Do something");
    } catch(Exception e) {
        System.err.println("Do something else");
    } finally {
        System.err.println("Grr arrrgh");
    }
}
```

Varför går följande kod inte att kompilera?

```
Test.java:15: exception java.lang.ArithmeticException has already been caught
    } catch(ArithmeticException e) {
            ^
```

```
void foo() { bar(); }

void example() {
    try {
        foo();
    } catch(Exception e) {
        System.err.println("Do something else");
    } catch(ArithmeticException e) {
        System.err.println("Do something");
    } finally {
        System.err.println("Grr arrrgh");
    }
}
```

Titta också på…

- Screencasten om undantagshantering och programmet ExceptionDemo i kursrepot
- Vilka typer av fördefinierade undantag som finns i Javas standardbibliotek
- Skillnaden mellan kontrollerade och okontrollerade undantag (tas bl.a. upp i screencasten ovan; på eng. checked/unchecked exceptions)
- Varför det är problematiskt att använda inre klasser (i motsats till nästlande klasser) för att definiera undantag