Föreläsning 24

Tobias Wrigstad

Parallellprogrammering, trådar – en försmak

```
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*

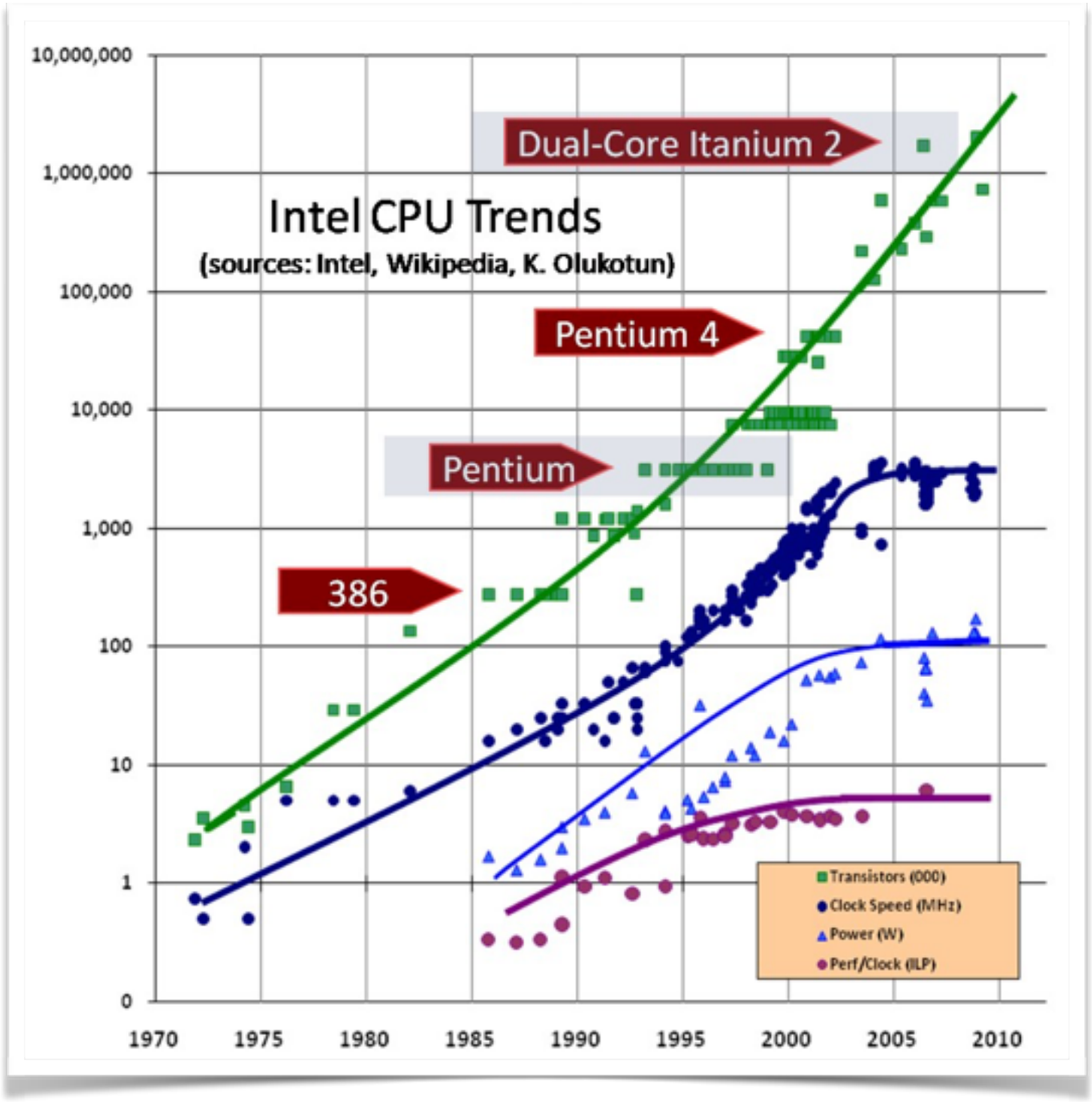
Concurrency

- 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.print(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 3 2 4 3 3 1 3 4 2 2 2 2 2 2 4 4 4 4 4 4 4 4 3 1 1 1 1 3 2 3 1 3 2 3 1 1 1 3 2
```

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;
    }
}
```

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
```

Trådar är svåra att programmera korrekt

The Problem with Threads

Edward A. Lee
Professor, Chair of EE, Associate Chair of EECS
EECS Department
University of California at Berkeley
Berkeley, CA 94720, U.S.A.
eal@eecs.berkeley.edu

January 10, 2006

Abstract

Threads are a seemingly straightforward adaptation of the dominant sequential model of computation to concurrent systems. Languages require little or no syntactic changes to support threads, and operating systems and architectures have evolved to efficiently support them. Many technologists are pushing for increased use of multithreading in software in order to take advantage of the predicted increases in parallelism in computer architectures. In this paper, I argue that this is not a good idea. Although threads seem to be a small step from sequential computation, in fact, they represent a huge step. They discard the most essential and appealing properties of sequential computation: understandability, predictability, and determinism. Threads, as a model of computation, are wildly nondeterministic, and the job of the programmer becomes one of pruning that nondeterminism. Although many research techniques improve the model by offering more effective pruning, I argue that this is approaching the problem backwards. Rather than pruning nondeterminism, we should build from essentially deterministic, composable components. Nondeterminism should be explicitly and judiciously introduced where needed, rather than removed where not needed. The consequences of this principle are profound. I argue for the development of concurrent coordination languages based on sound, composable formalisms. I believe that such languages will yield much more reliable, and more concurrent programs.

1 Introduction

It is widely acknowledged that concurrent programming is difficult. Yet the imperative for concurrent programming is becoming more urgent. Many technologists predict that the end of Moore's Law will be answered with increasingly parallel computer architectures (multicore or chip multprocessors, CMPs) [15]. If we hope to continue to get performance gains in computing, programs must be able to exploit this parallelism.

[1]This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF award No. CCR-0225610), the State of California and the following companies: Agilent, DGIST, General Motors, Hewlett Packard, Infineon, Microsoft,

900+ citat enligt Google Scholar

Inte längre 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

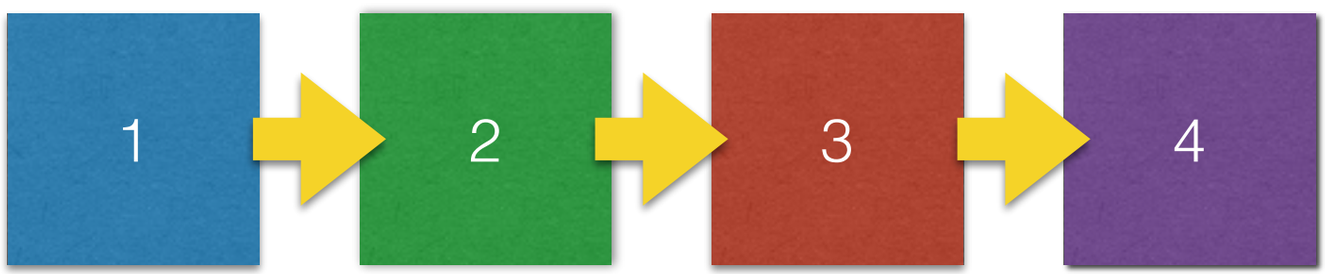
- Under huven finns ett antal trådar men målet här är att **undvika** att se dem

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

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

Task-parallellism

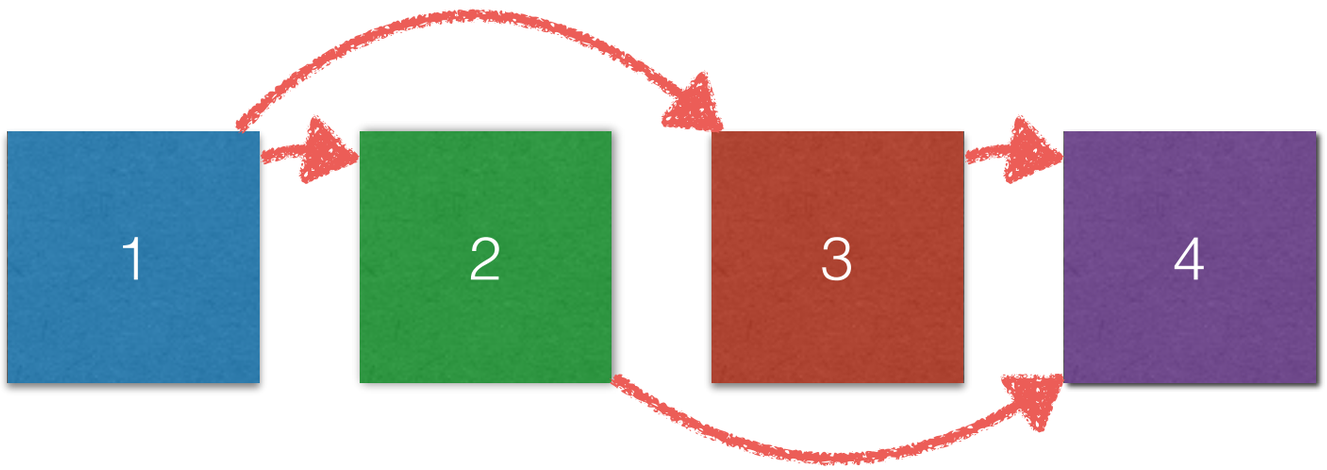
```
my_function(…) {
1  x, y = frob(z);
2  foo₁ = bar(x);
3  foo₂ = bar(y);
4  foo₃ = quux(foo₁, foo₂);
}
```



Den sekventiella hjärnan ser…

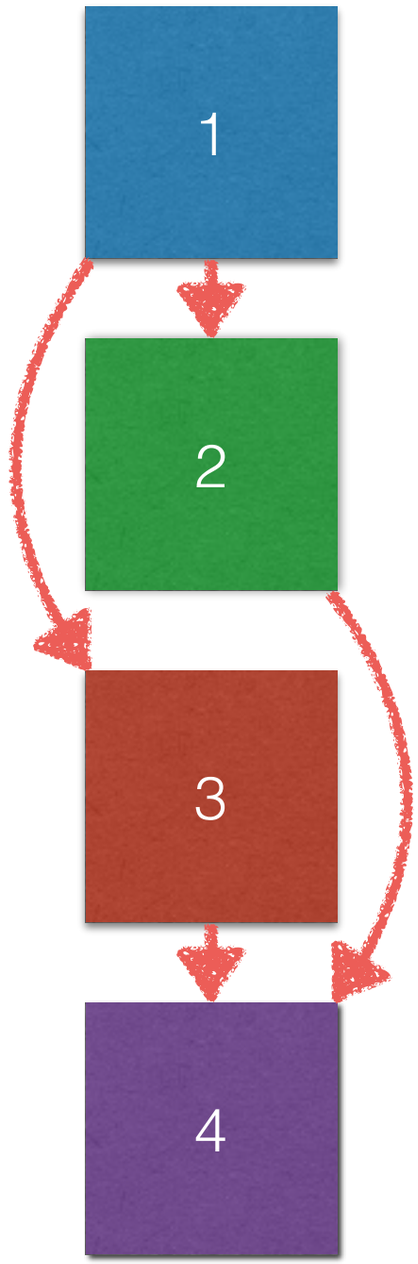
Task-parallellism

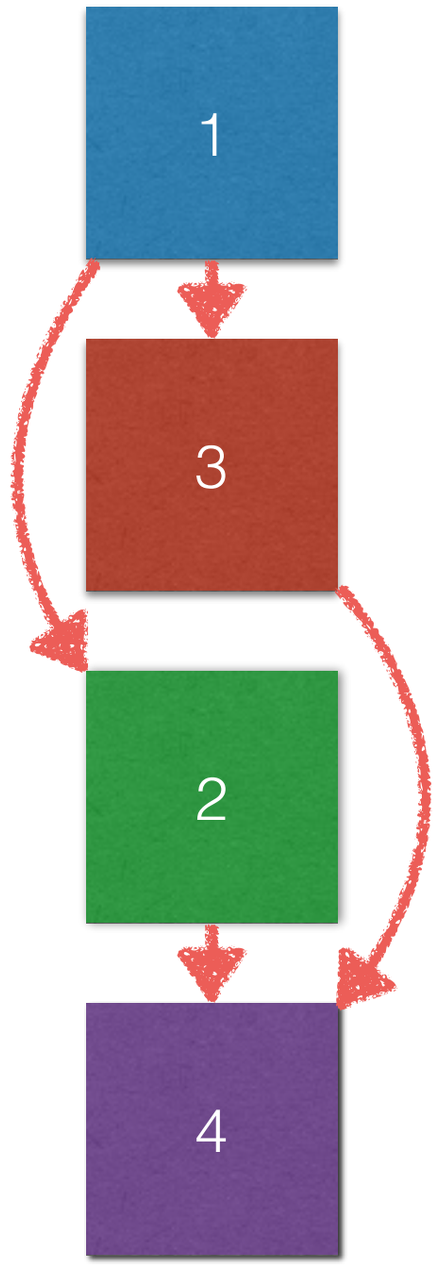
```
my_function(…) {
1     x, y = frob(z);
2     foo1 = bar(x);
3     foo2 = bar(y);
4     foo3 = quux(foo1, foo2);
}
```

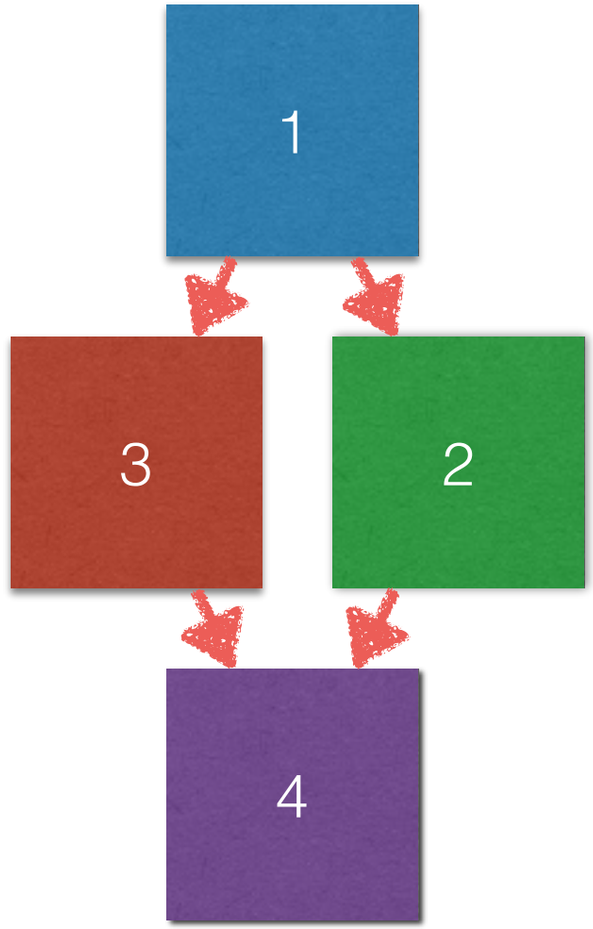


De faktiska beroendena

Hur kan vi schemalägga dem?







Task-parallellism

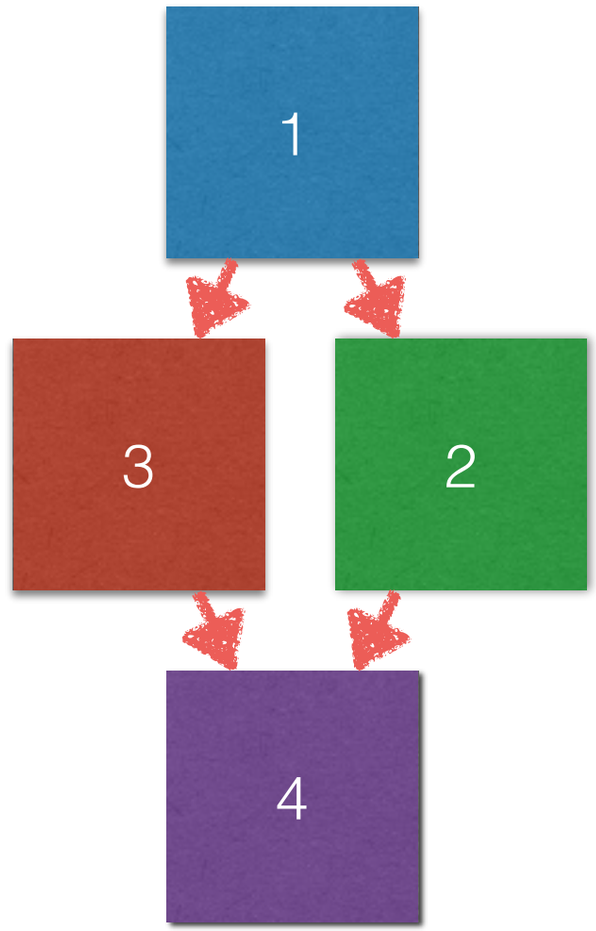
- Utför operationen på 3 tidsenheter istället för 4

  Minskad **latency**

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

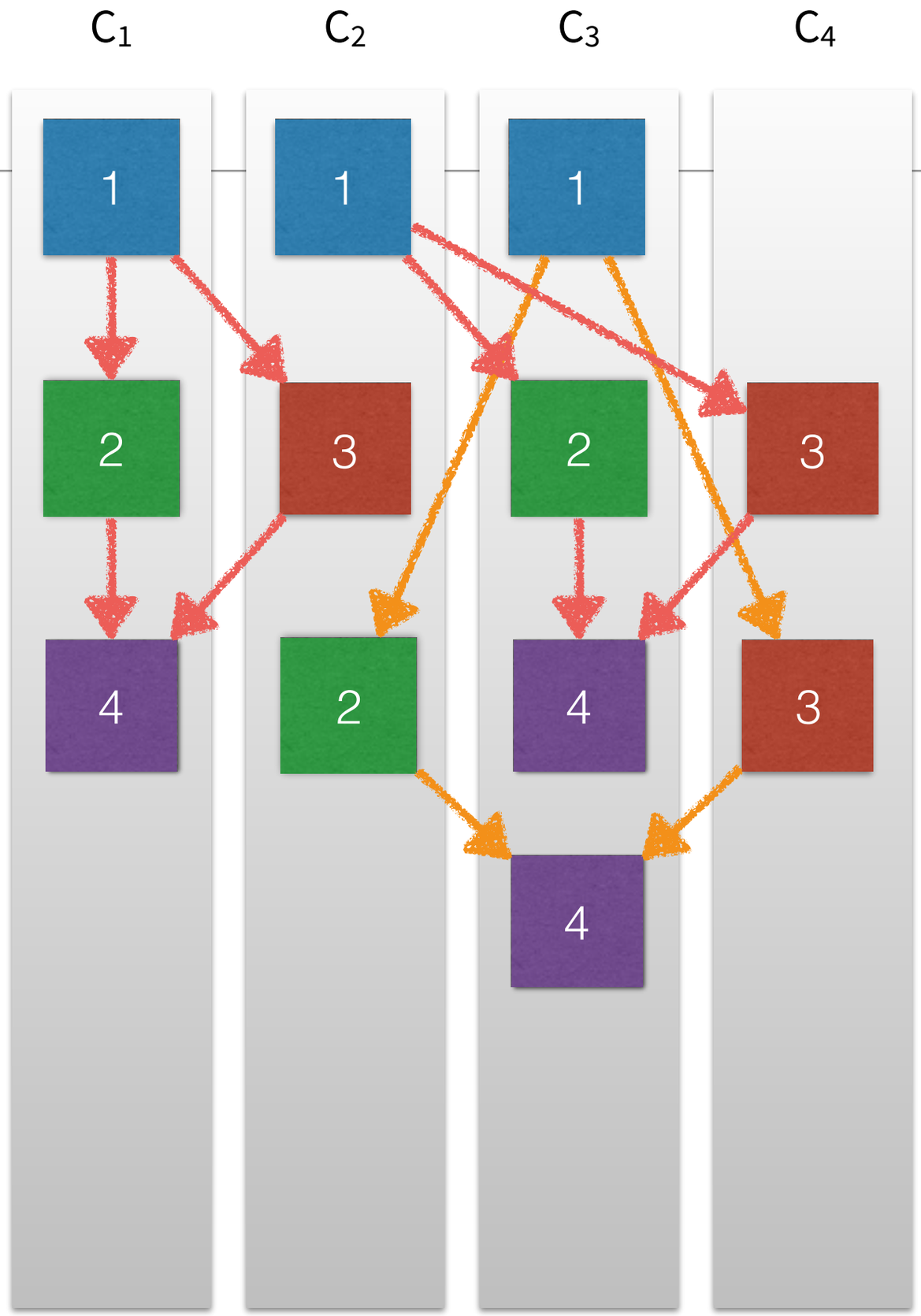
  Ökad **throughput**

Hur då?

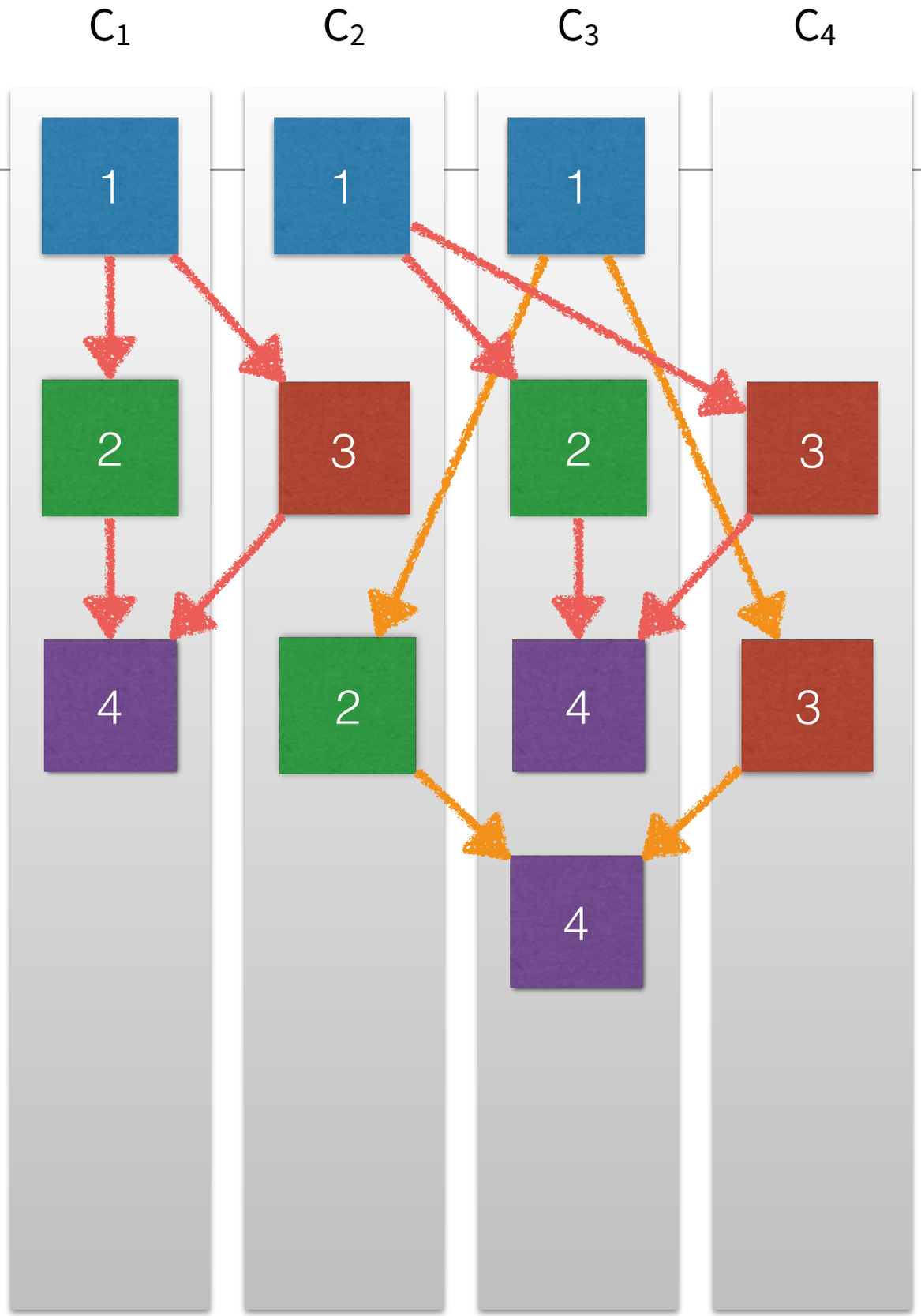


Svar [1 av flera]

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

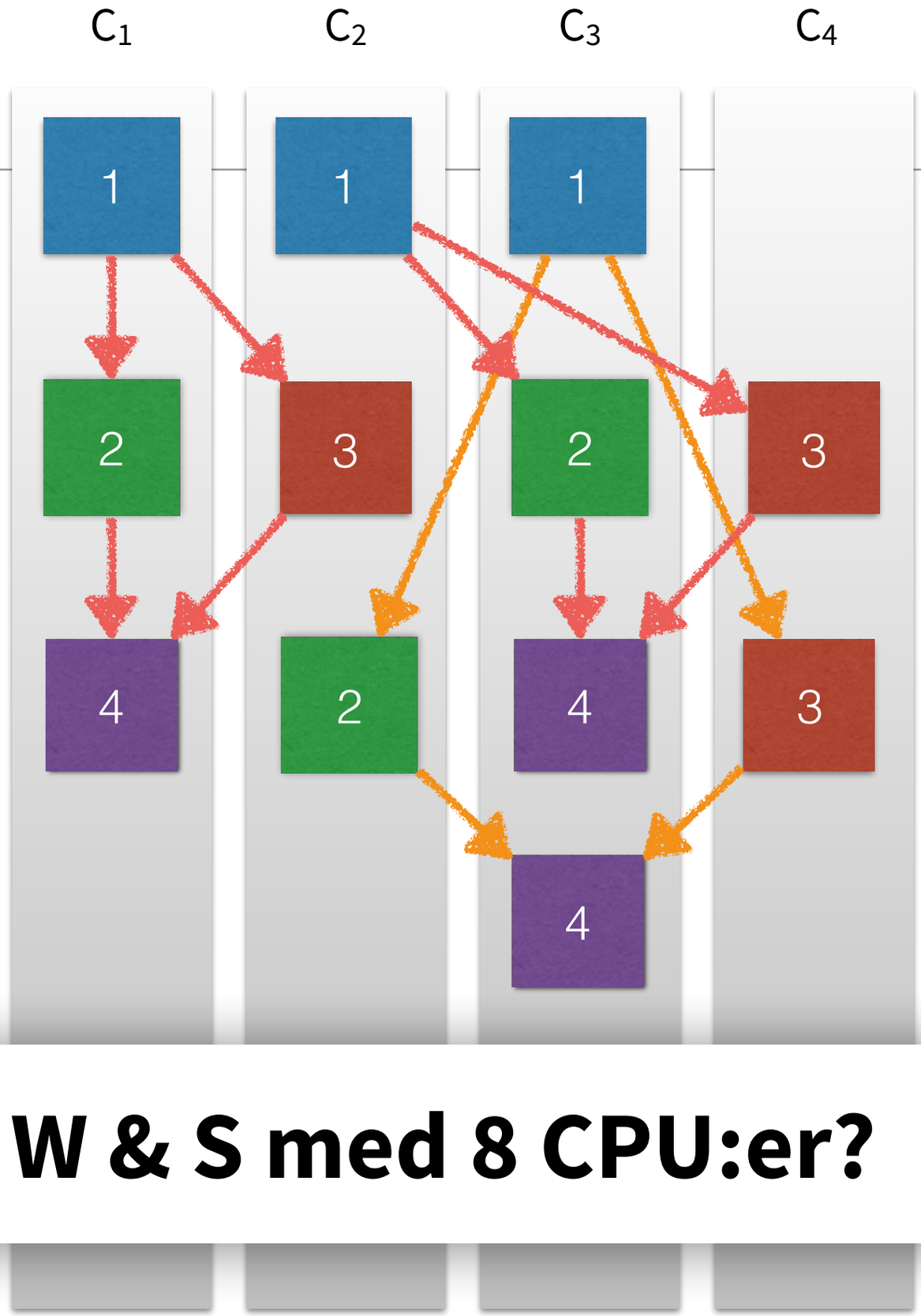
Work

T_1 — tiden det skulle ta med bara en CPU

Span

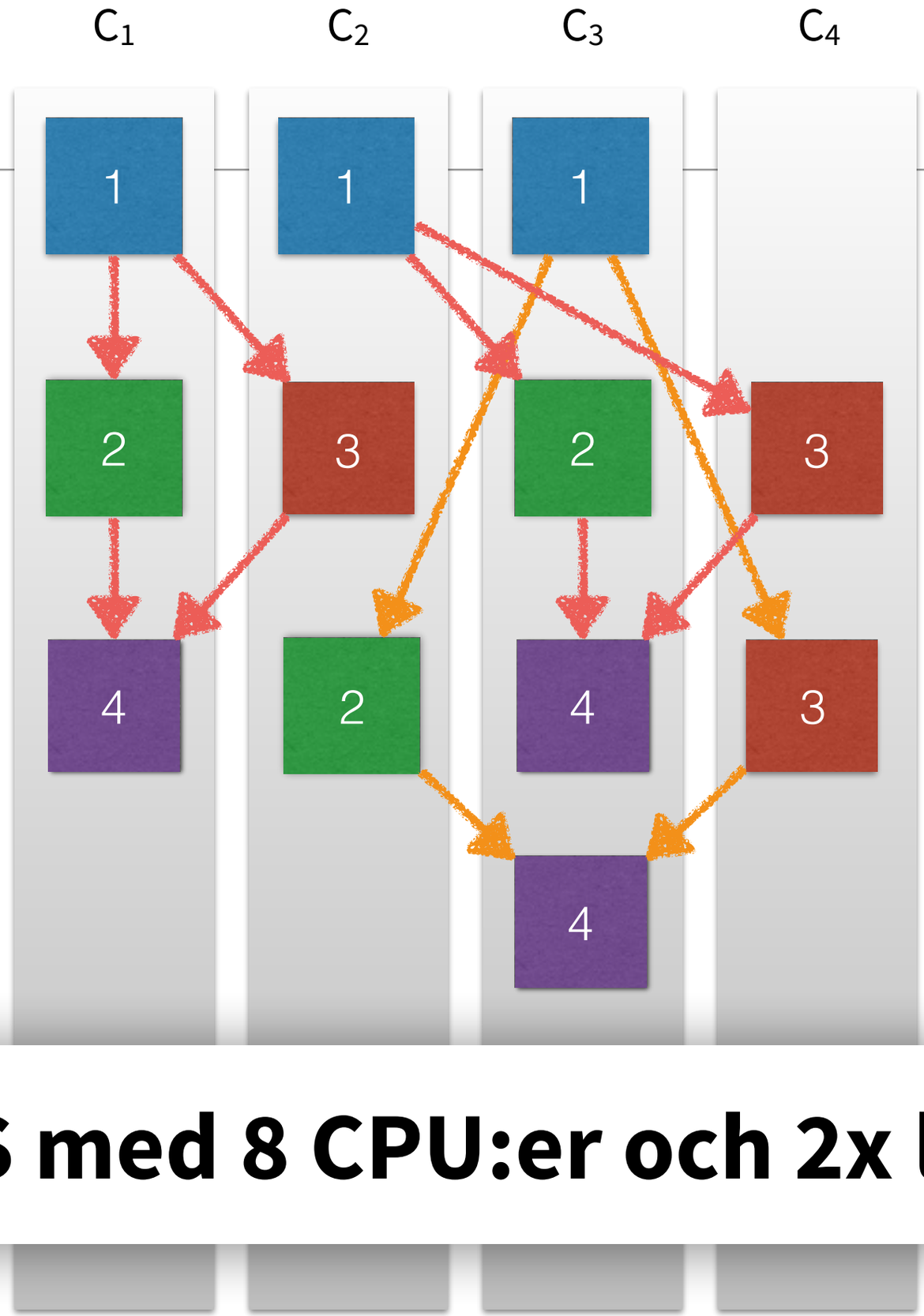
T_∞ — tiden det skulle ta med oändligt många CPU:er

Work & Span



W & S med 8 CPU:er?

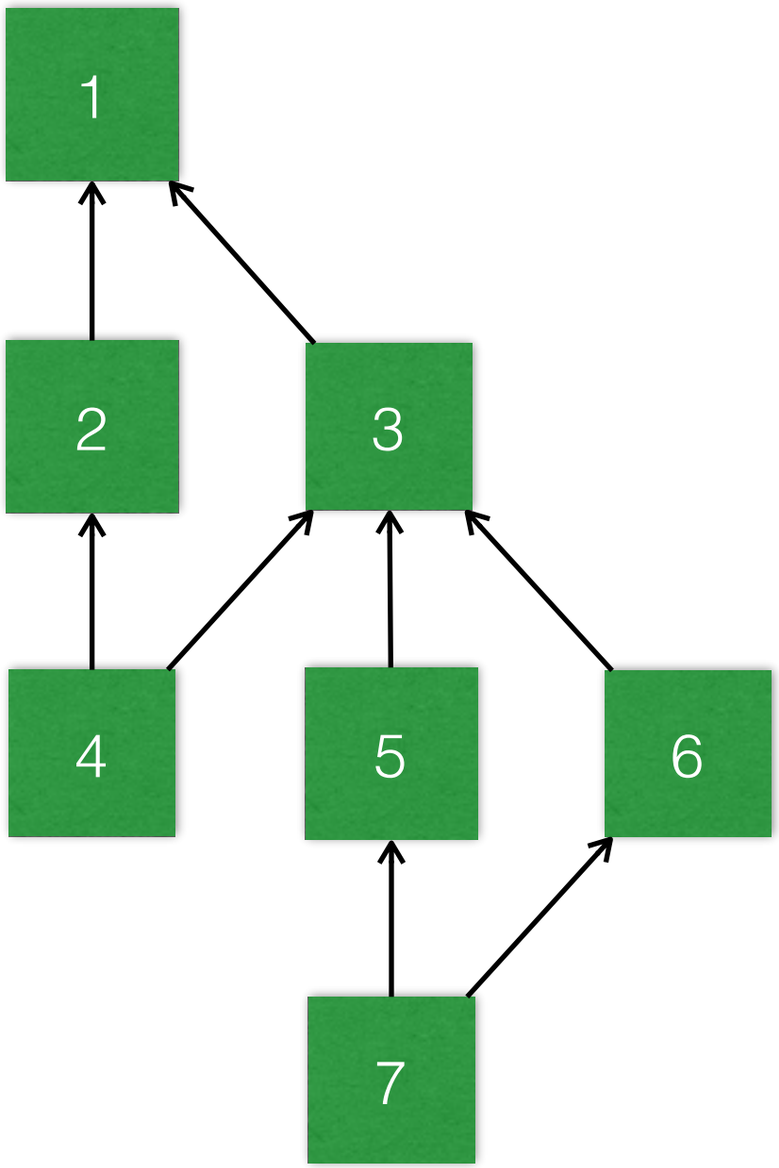
Work & Span



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

Tasks formar en riktad acyklisk graf

Viktig skillnad mot trådar!



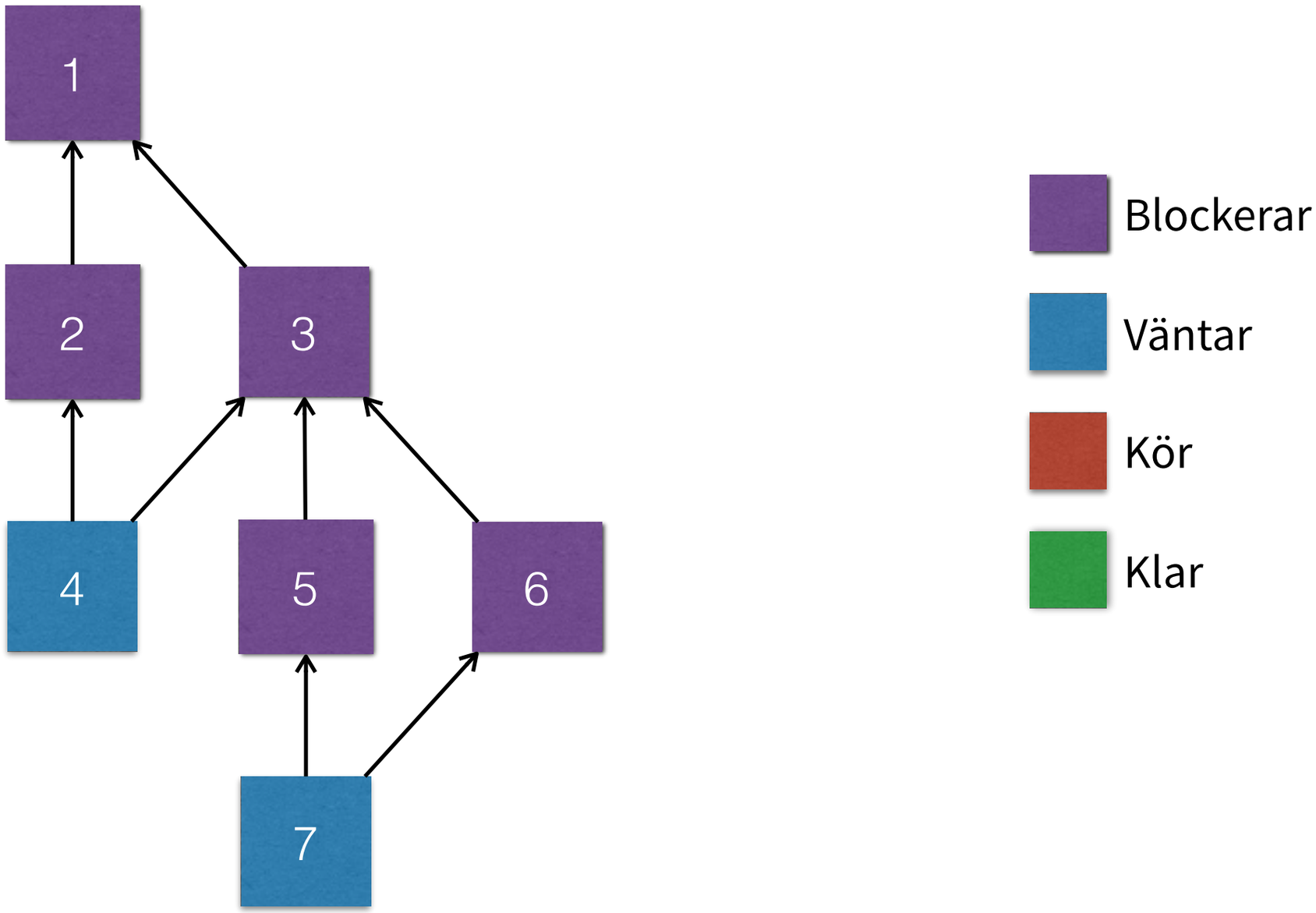
Noder är tasks

Kanter är beroenden (1 beror på 2 och 3)

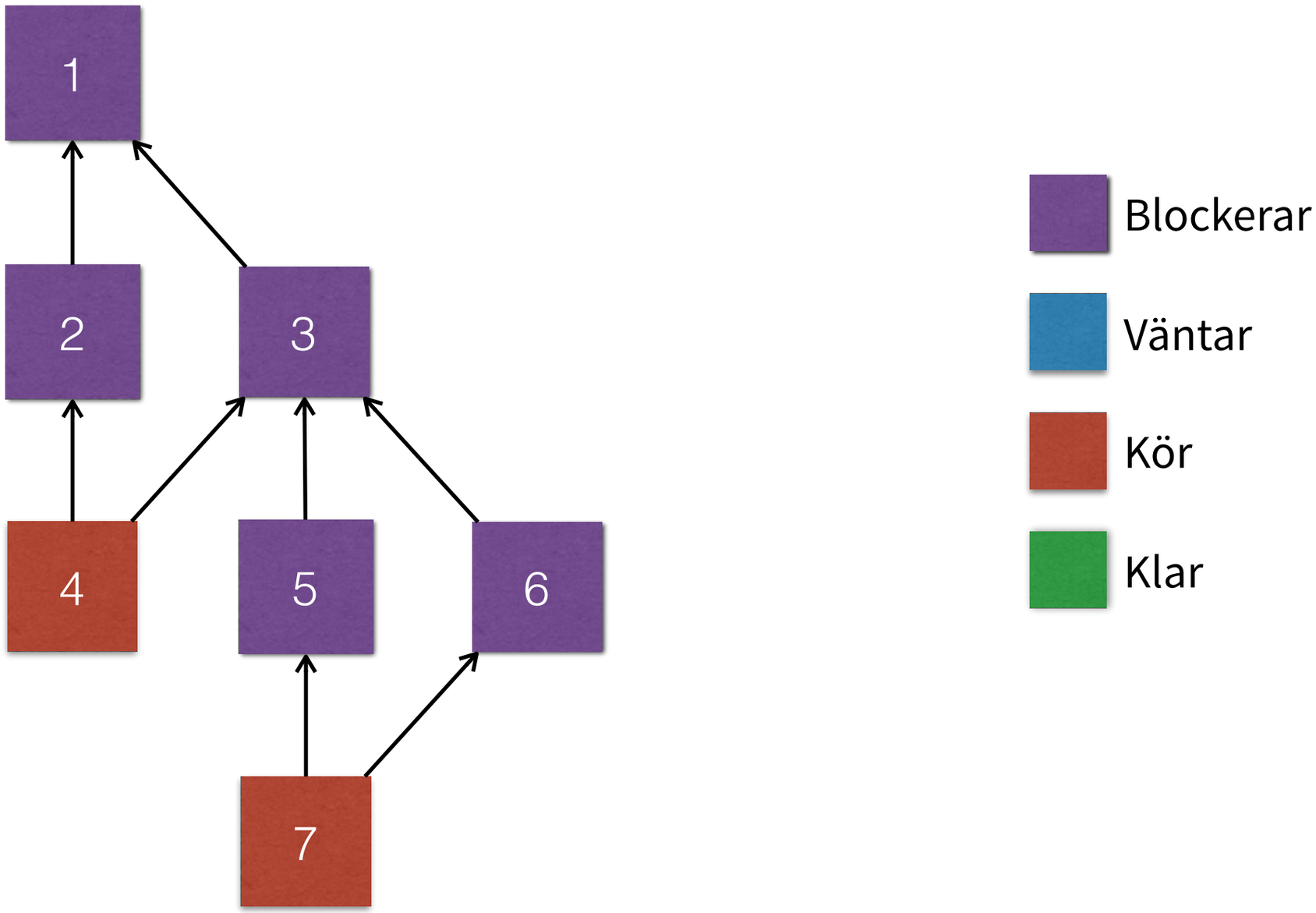
Vi kan enbart utföra task som inte har beroenden

Vilka tasks kan utföras?

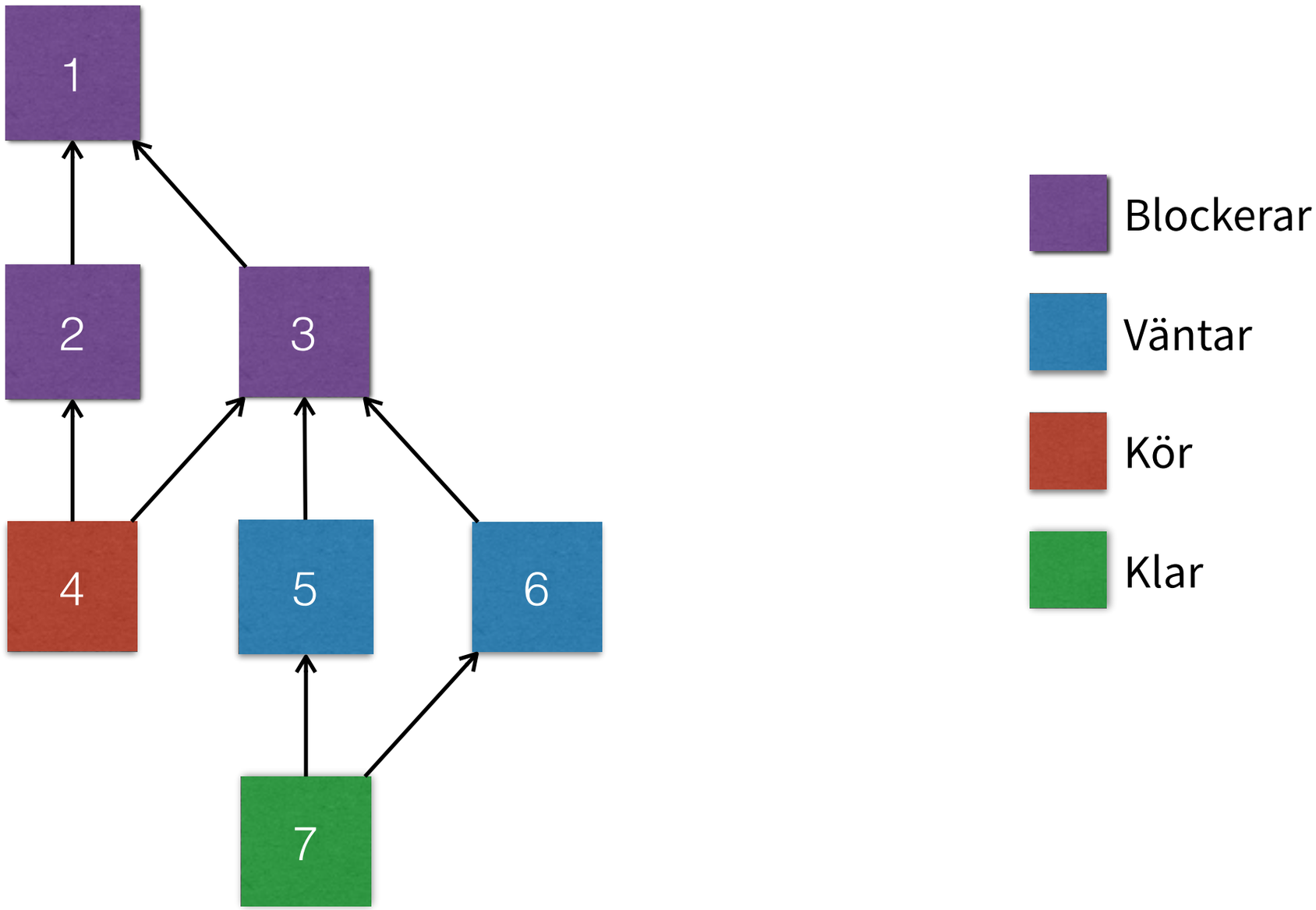
En tasks tillstånd



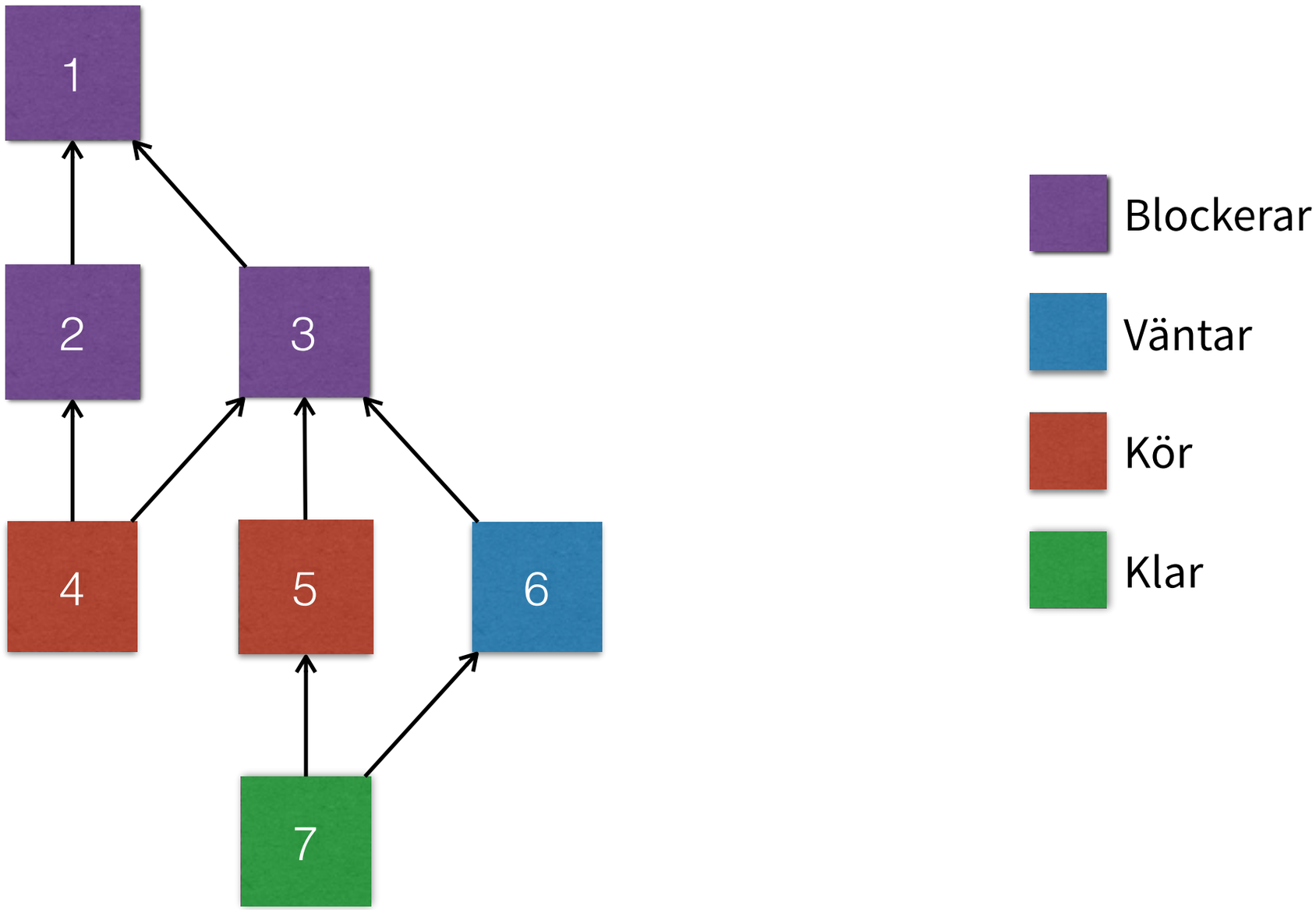
Vi kör!



Vi kör!



Vi kör!



Exempel: parallellisera en map

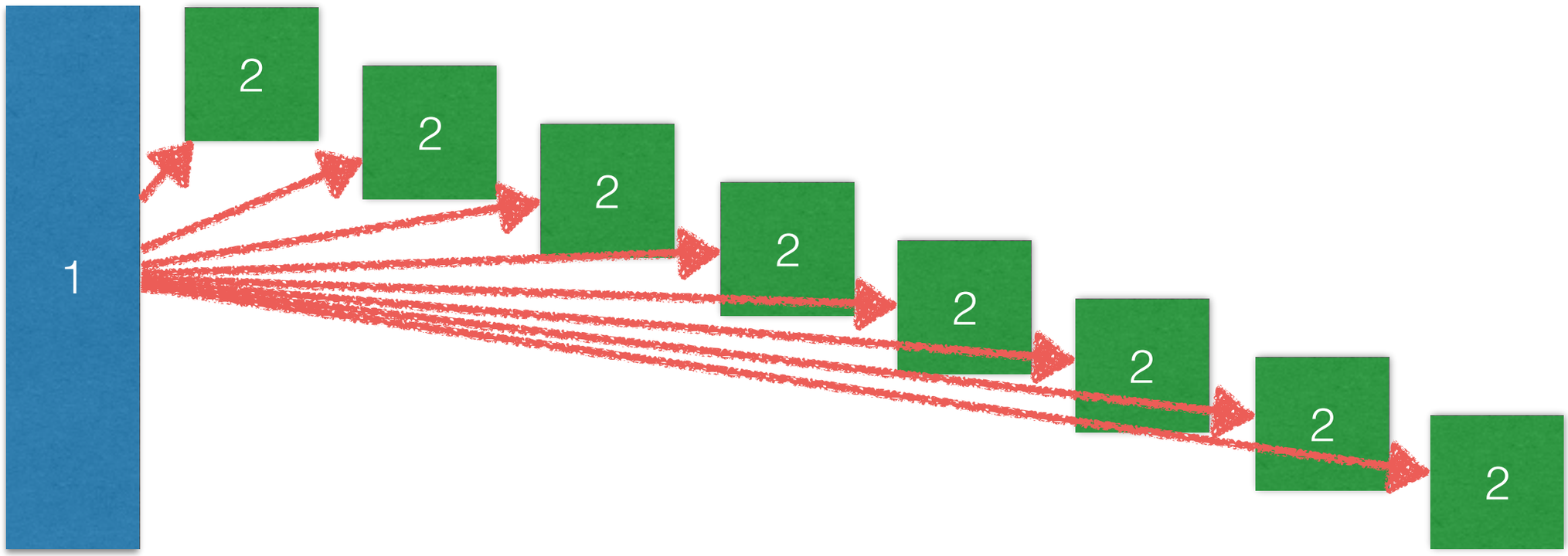
- Var finns våra tasks?

```
[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

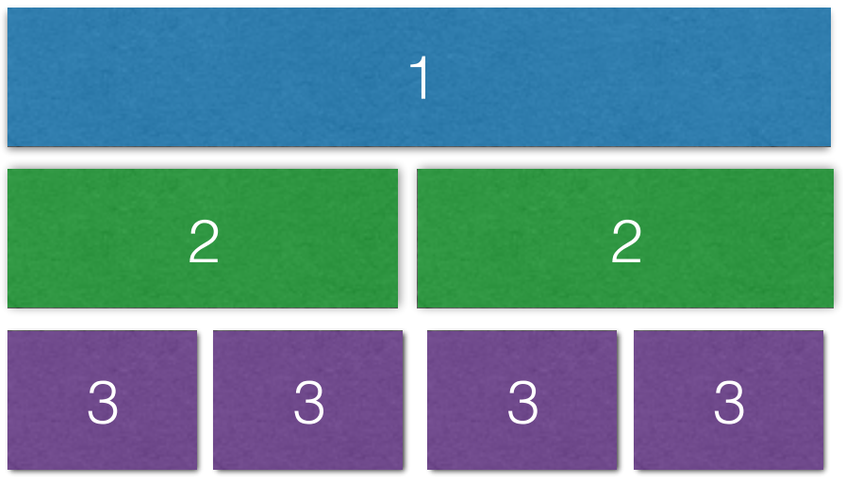


Exempel: parallellisera en map

```
[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 en array i Java

Fork/Join

Recursive Task

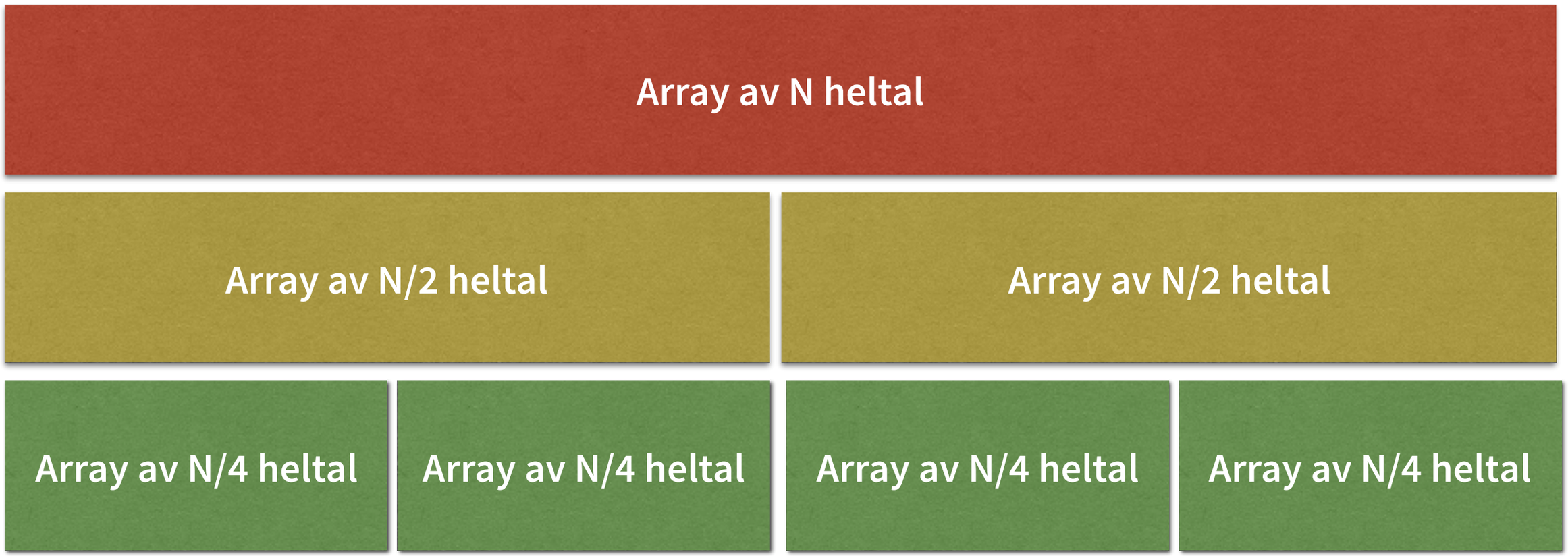
Summera tal i en array

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

Sum.java

```java
public class Sum {
    public static void main(String[] args) {
        int array[] = new int[] { … };
        int sum = 0;
        for (int i : array) {
            sum += i;
        }
        System.out.println("Sum: " + sum);
    }
}
```

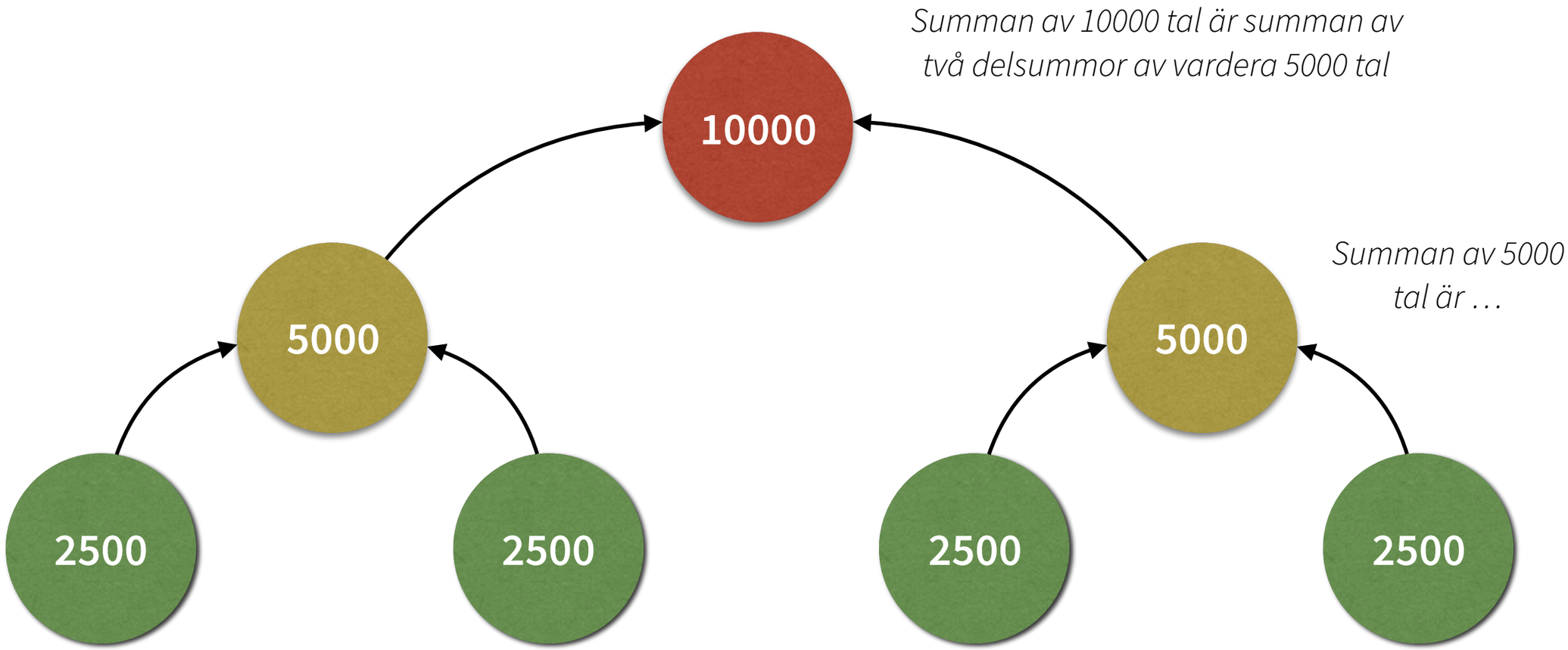
Summera med divide-and-conquer



. . .

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

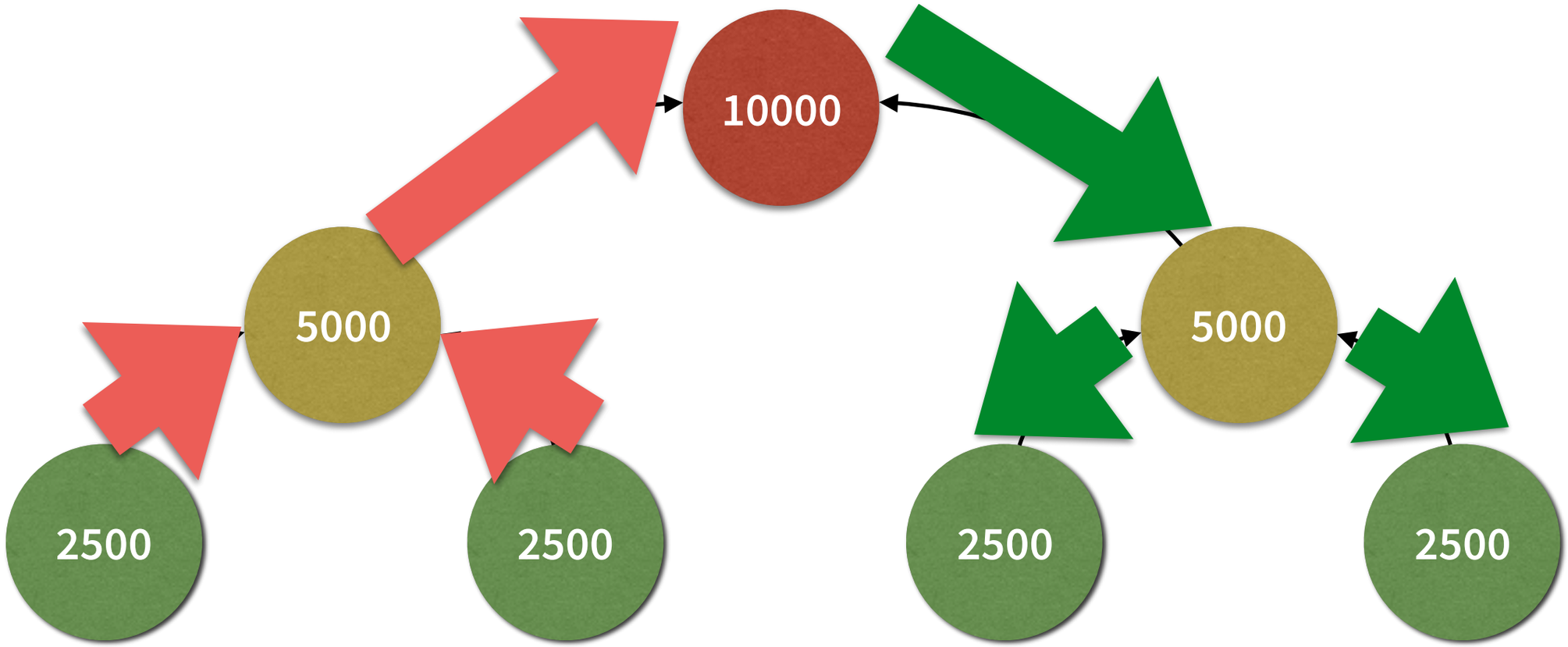
Acyklisk graf av beroenden



…

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

```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-
    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[] { … }));
    }
}

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

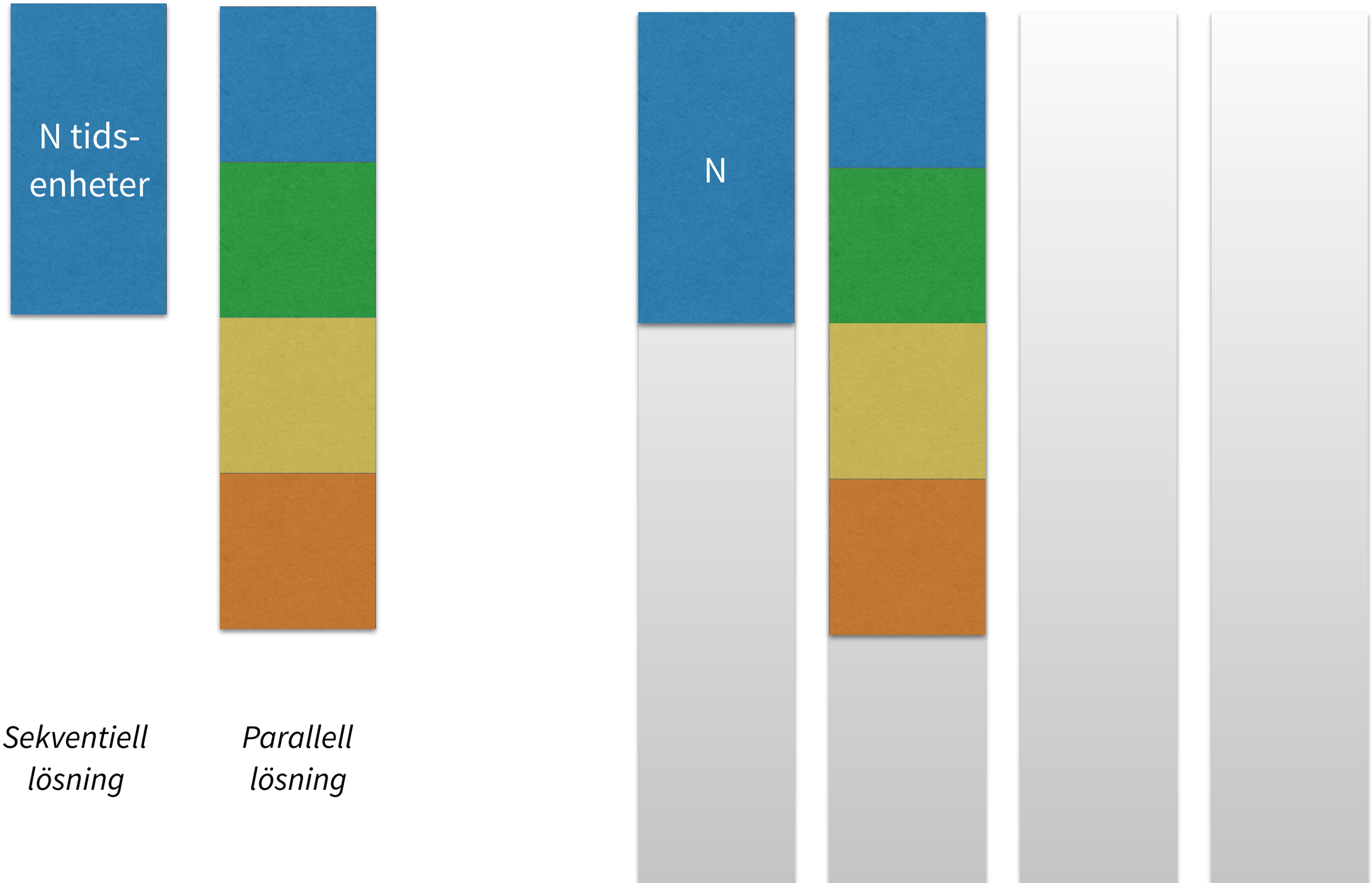
```
$ javac FJ.java
$ java FJ
<large number>
```

```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-
    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));
    }
}
```

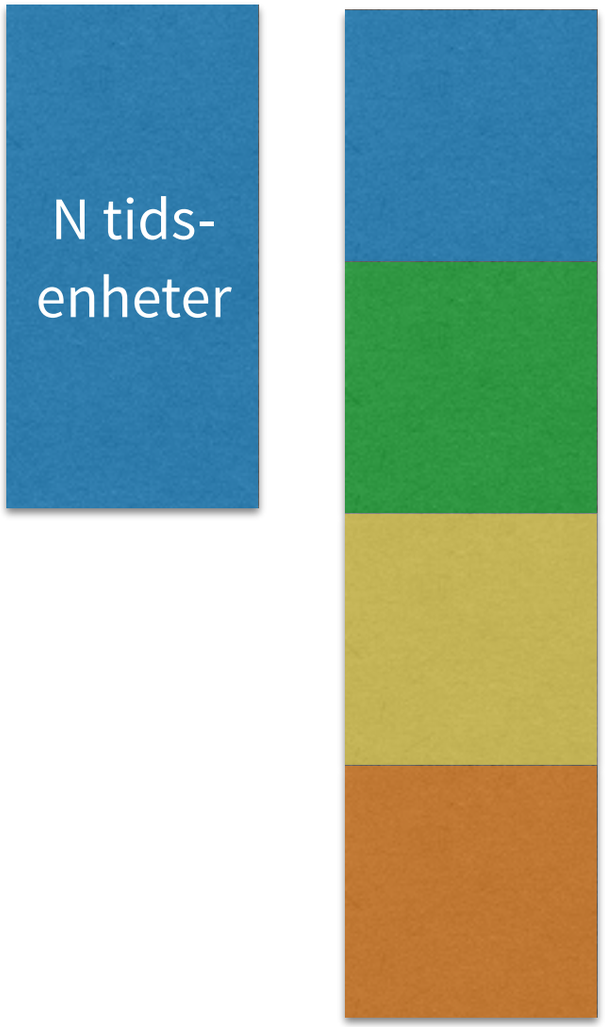
```java
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

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

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

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

Parallella och sekventiella lösningar

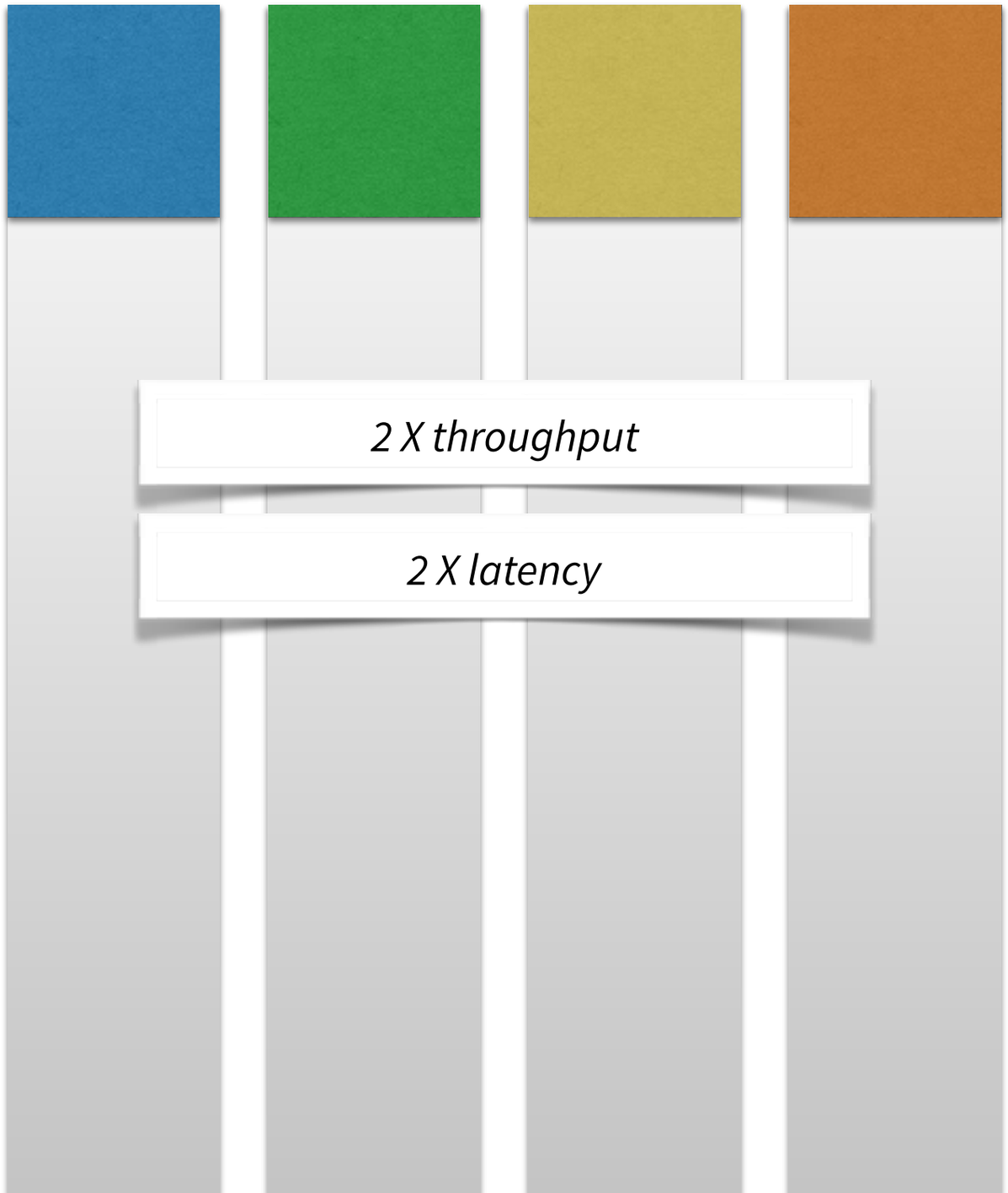


Parallella och sekventiella lösningar

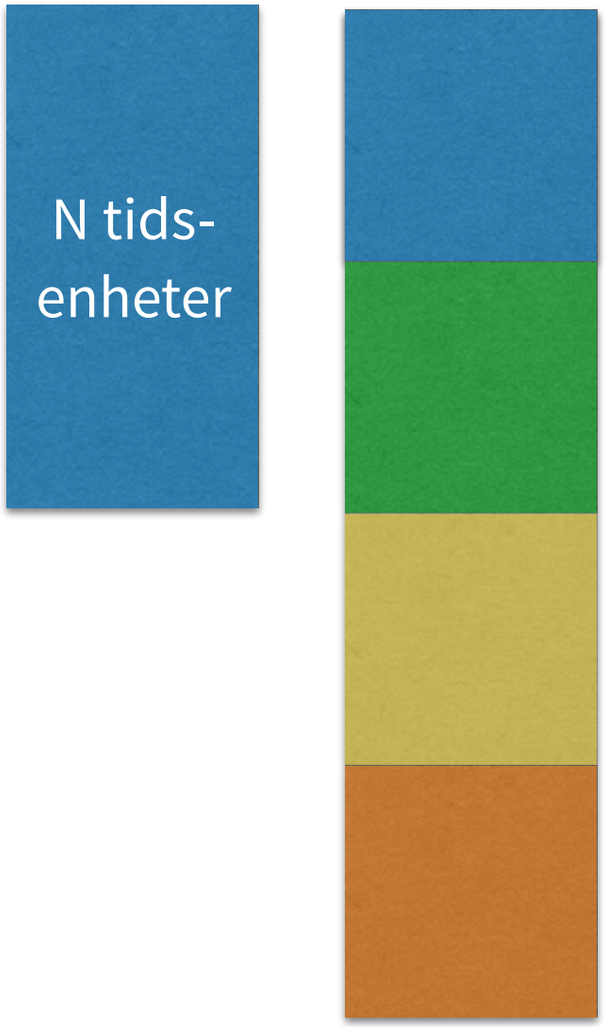


Sekventiell lösning

Parallell lösning

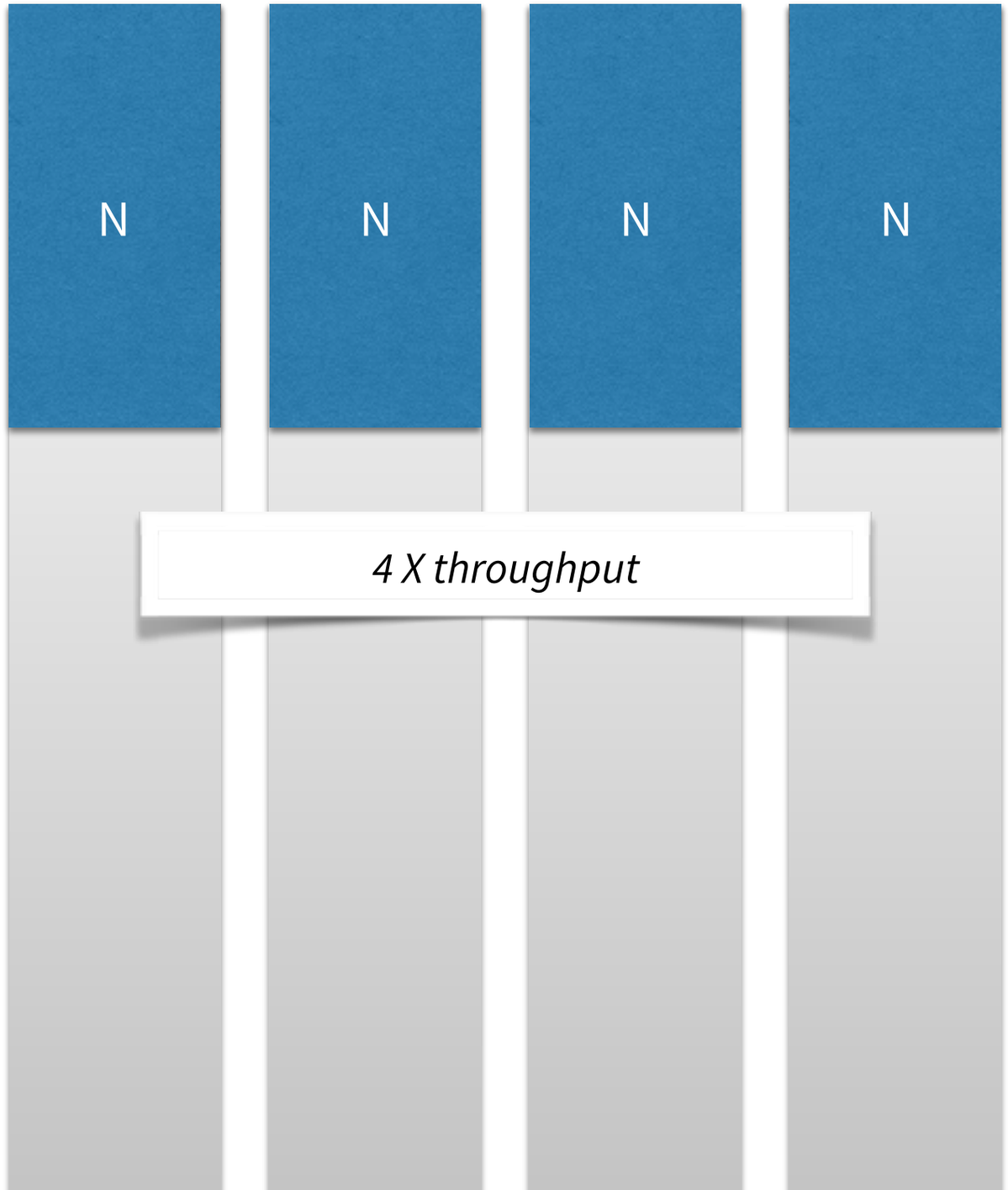


Parallella och sekventiella lösningar

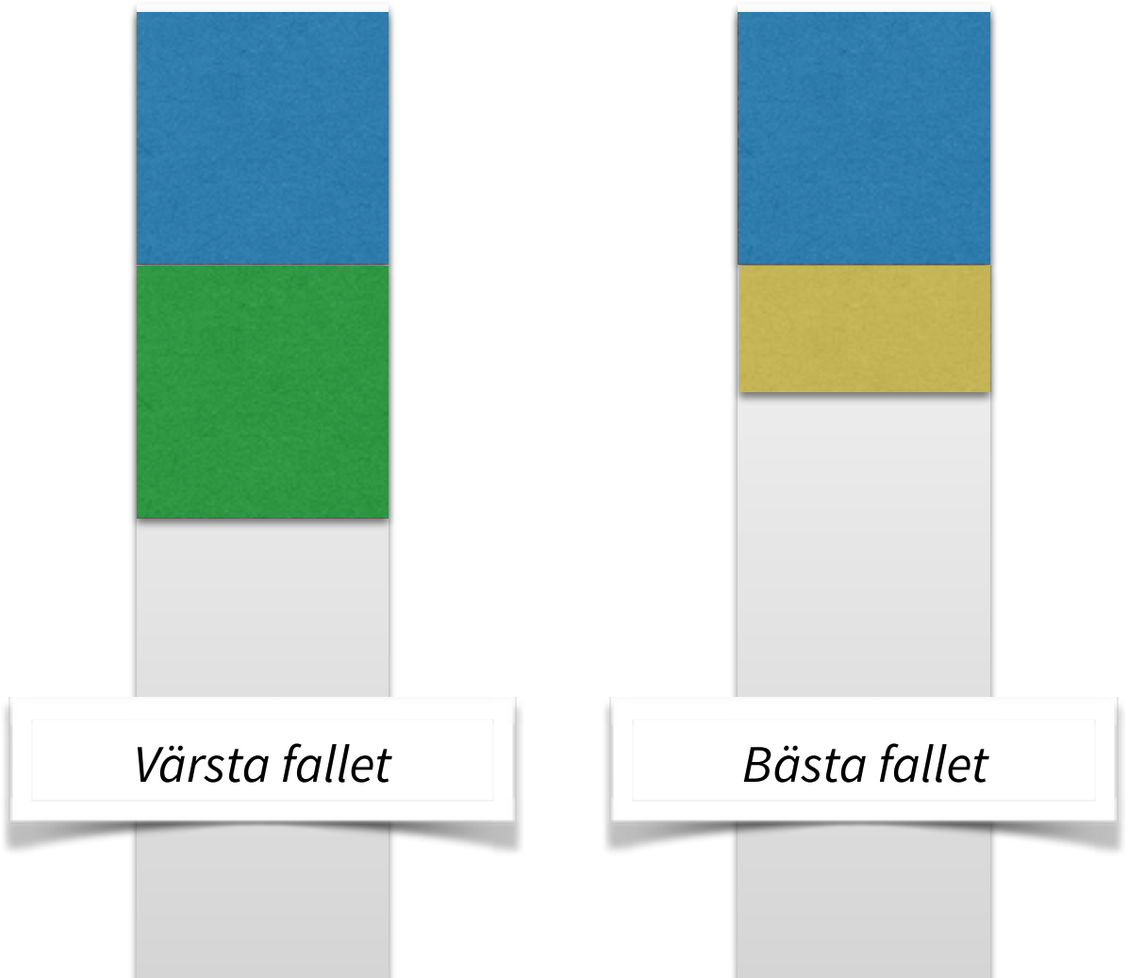


Sekventiell lösning

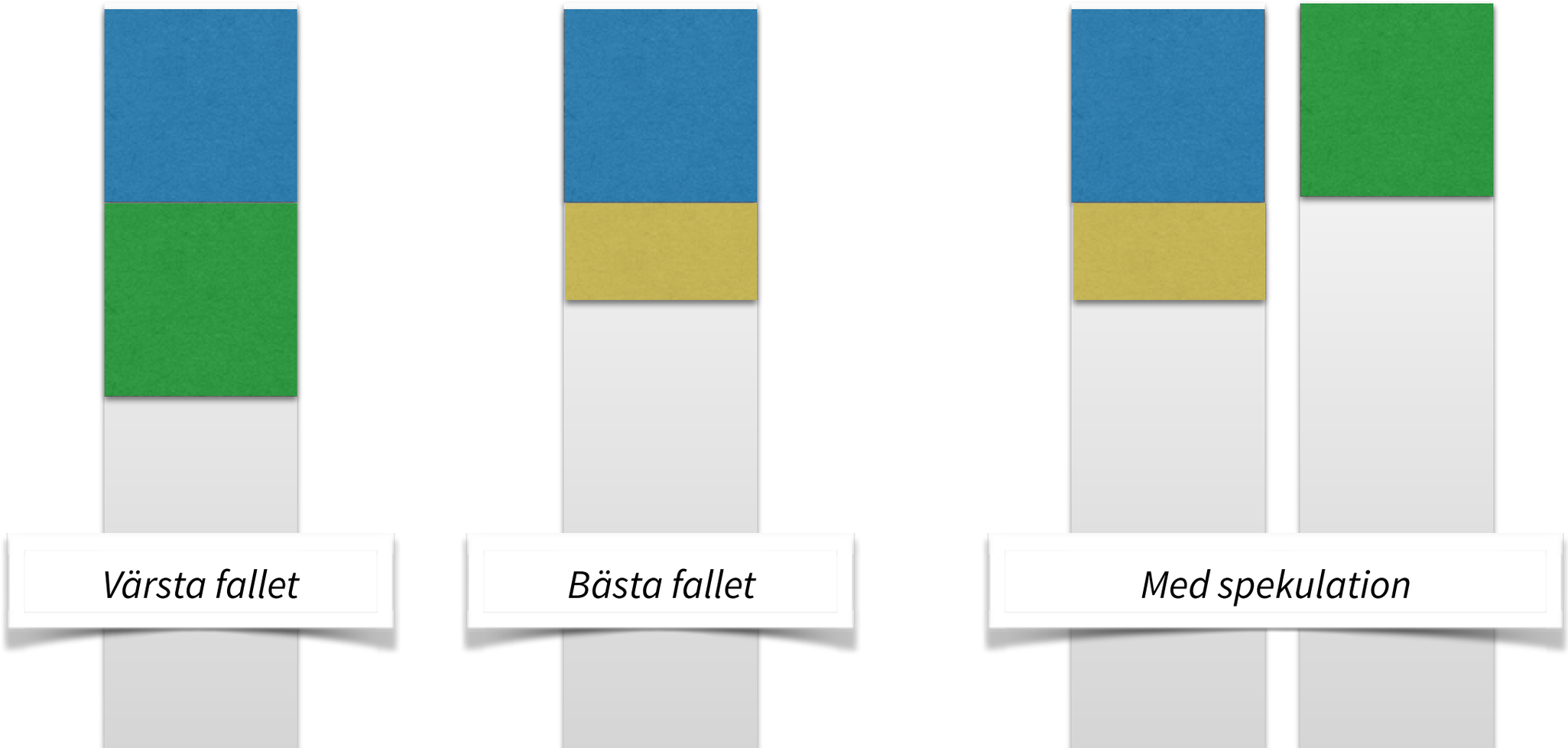
Parallell lösning



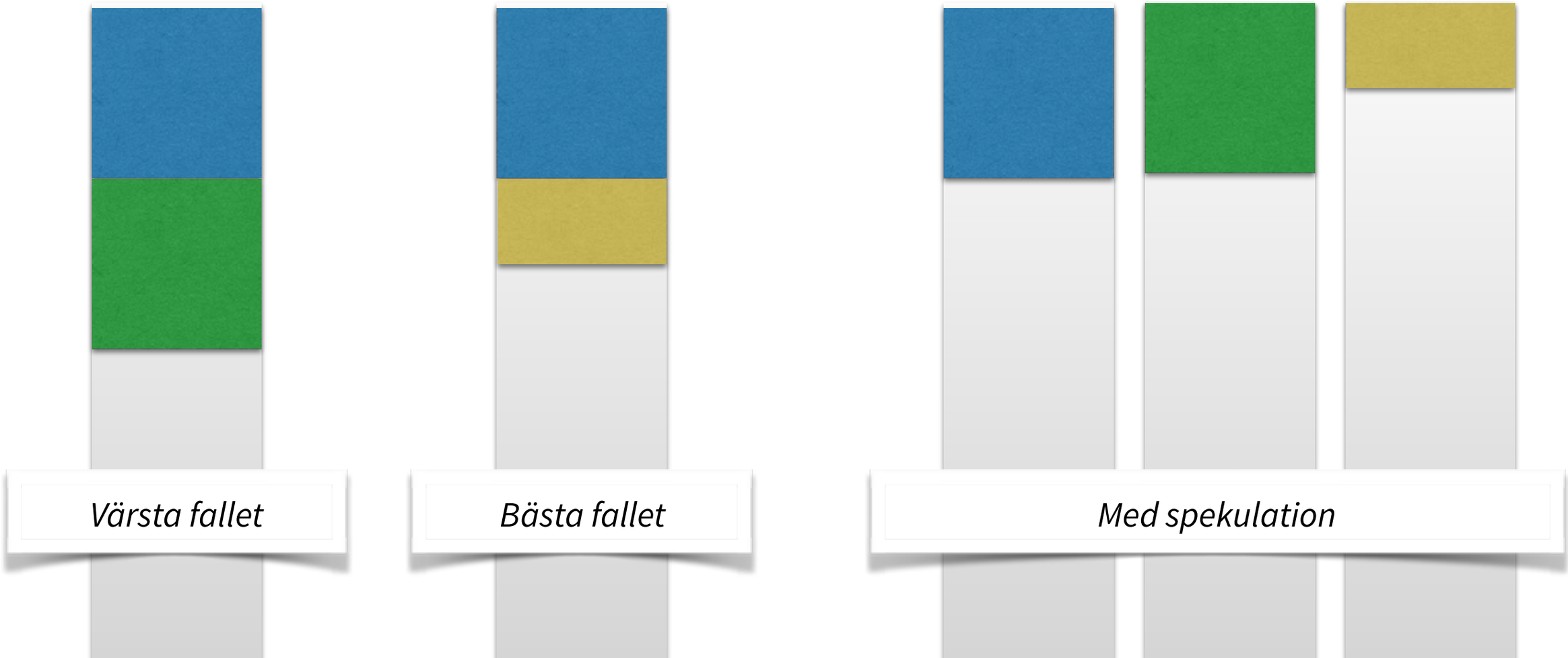
Spekulativ parallelism



Spekulativ parallelism



Spekulativ parallelism



Sammanfattningsvis

- De program vi har tittat på under denna kurs har varit av en *särart* – sekventiella!

  Moderna program måste i allt högre grad lösa problem parallellt

  …*och concurrent*

- Detta komplicerar programmeringen

- Var får du lära dig mer om detta?

  Operativsystem (går in på delar av detta i mer detalj)

  Master-spåret i Concurrent- och parallellprogrammering (flera enskilda kurser)