# Distribuera mera - Spark och Hadoop utan Big Data

## En undersökning i hur verktyg designade för Big Data kan användas med små datamängder

**Oscar Nihlgård, 2016, Lunds Tekniska Högskola, Campus Helsingborg**

## Introduktion

Detta arbete har skett i samarbete med företaget Cybercom, som har bidragit med arbetsplats, hårdvara och handledning.

---

Distribuering innebär att programkod arbetar parallellt på flera datorer, så kallade noder, i ett kluster.

Vanligtvis används distribuering endast i extrema situationer - vid väldigt stora datamängder (Big Data) eller vid väldigt höga exekveringstider. Syftet med detta arbete har varit undersöka om två verktyg för distribuering, Spark och Hadoop Distributed File System, är relevanta även i små situationer (små datamängder, kort exekveringstid).

- Är distribuering relevant i små situationer?

## Teknisk backgrund

I arbetet användes två distribueringsverktyg med olika syften: Hadoop Distributed File System (HDFS) för lagring och Spark för exekvering.

### HDFS

HDFS är ett distribuerat filsystem baserat på Google File System (GFS). Syftet med HDFS är att möjliggöra snabb, parallell läsning av data. Datan i filsystemet sprids ut över noderna i klustret, vilket till exempel tillåter att datan kan läsas parallellt.

### Spark

Spark är ett verktyg för utveckling av distribuerbar kod. Distribuering möjliggörs genom att algoritmer implementeras med ett antal distribuerbara högnivåoperationer så som till exempel `Map`, `Reduce` och `Sort`. Detta API är mycket likt API:et hos Javas strömklass.

Spark och HDFS jobbar bra tillsammans. Genom att använda ett distribuerat filsystem tillsammans med Spark kan varje nod i första hand arbeta med den data som finns tillgängligt lokalt. På så viss minimeras kommunikationen mellan noderna.

## Metod

- Utveckling av ett abstraktionslager som möjliggör skrivandet av kod som gåt att köra både distribuerat och lokalt effektivt.
- Prestandatester av Spark och HDFS, primärt prestandatester av ett exempelprogram som utvecklades som en del av arbetet.

### Exempelprogram - generering av text

För att ha något att köra prestandatester på utvecklades ett litet program som använder sig av Spark. Programmet tar textdata som input och genererar utifrån det ett flöde av ny text. Detta görs genom att algoritmen "minns" de två senaste orden i flödet och utifrån dem väljer ett nytt ord baserat på vilket som är vanligast när de två orden kommer före (algoritmen är inte deterministisk utan väljer slumpmässigt ett av de fem vanligaste orden). Genom att upprepa detta kan en godtyckligt lång text genereras.

I prestandatesterna användes ett dataset bestående av 170MiB slumpmässigt utvald engelskspråkig text från Project Gutenberg (https://www.gutenberg.org/) som input till programmet. Här följer ett exempel på text genererad av programmet med detta dataset:

*"No one has ever given or received this token of affection."*

### Prestandatester

Ett kluster med fyra datorer (tre exekverare och en schemaläggare) användes till körningen av prestandatesterna. Syftet med dessa tester var följande:

- Undersöka kostnaden för distribuering
- Undersöka hur mycket effektivare det är att arbeta med strömmar än Spark i lokalt läge
- Undersöka förutsättnigarna för distribuering vid små situationer

Resultatet av de viktigaste delarna av dessa undersökningar presenteras som resultatdelen i denna poster.

### Abstraktionslager

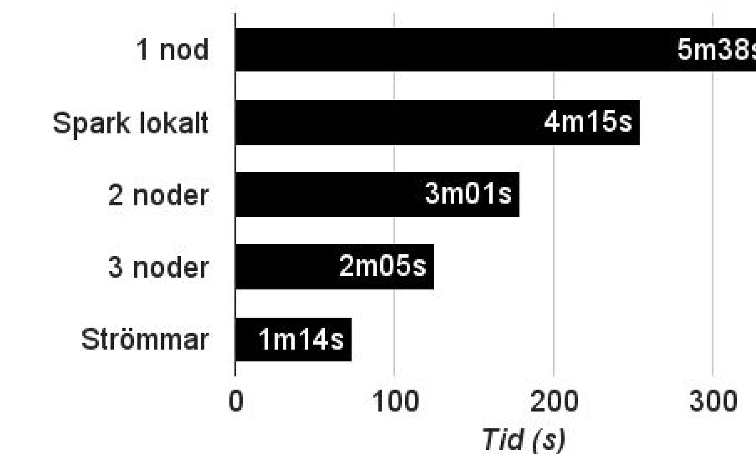
Ett problem vid distribuering i små situationer är att det det kanske bara är lämpligt med distribuering ibland. Till exempel kan det finnas rörliga faktorer som påverkar.

Detta löses med ett abstraktionslager som hanterar lokal körning bättre än verktyg för distribuering. I detta fallet implementerades den lokala körningen med Javas strömklass från version 1.8.

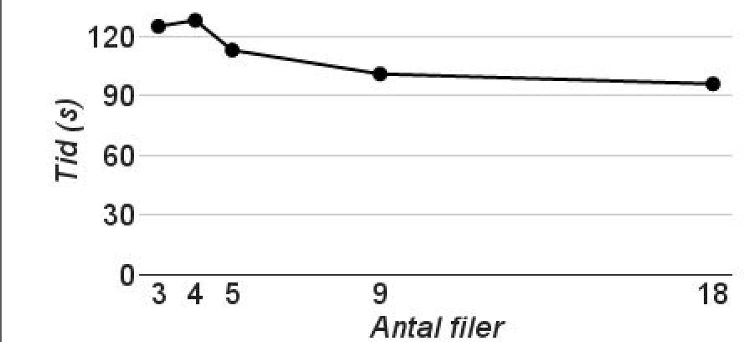
Till vänster ses ett kodexempel på hur abstraktionslagret används - en berräkning av vinstfördelningen i en samling schackmatcher. API:et är baserat på Sparks API.

```
public void analyseChessGames(Provider provider) {
  lines = provider.textfile("path/to/chess/files");
  lines = lines.filter((line) ->
    line.startsWith("result: "));

  ParallelMap<Winner, Integer> results =
      lines.mapToPair((line) -> {
    char khar = line.charAt(8);
    if (khar == '0') return new Tuple2<>(Winner.White, 1);
    if (khar == '1') return new Tuple2<>(Winner.Black, 1);
    if (khar == '2') return new Tuple2<>(Winner.Draw, 1);
    return new Tuple2<>(Winner.Invalid, 1);
  });

  results = results.reduceByKey(0, (a, b) -> a + b);
  results.collect().forEach(System.out::println);
}
```

## Resultat prestandamätningar

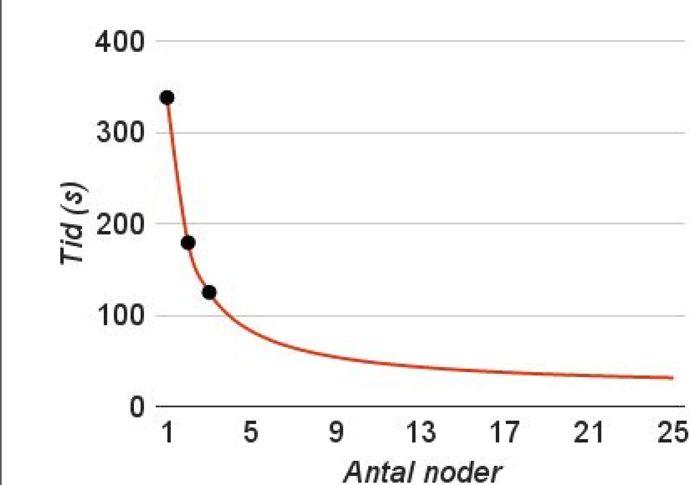



*Strömmar* & *Spark lokalt* kördes med 7GiB primärminne, övriga kördes med 2GiB. Detta på grund av brister i abstraktionslagret som ledde till att strömmetoden krashar vid för lite minne.




Första mätvärdet är ej inkluderat för att göra grafen enklare att läsa:

2 noder - 350 s




Trendlinjens ekvation är

`y = 18.5 + (338 - 18.5) / x`

Mätvärdet för strömmar passeras vid 6 noder, ty

`6 > (338 - 18.5) / (71 - 18.5)`

## Slutsats

- Verktygen Spark och HDFS kan, trots att de är designade med stora datamängder i åtanke, vara effektiva även vid så kallade små situationer.
- Även om det finns stora kostnader (i textgenereringsfallet uppskattades kostnaden till ~18 sekunder vid ett oändligt antal noder och flera minuter vid en nod) vägs det upp av den ökade parallelliseringen.
- Ett problems distribuerbarhet är begränsad av dess uppdelningsbarhet - fler noder än antalet delar ger ingen extra vinst.