# CS 696 Intro to Big Data: Tools and Methods
Fall Semester, 2019
Doc 2 Big Data Introduction
Jan 24, 2019

# Big Data

Data sets that are so large or complex that traditional data processing applications are inadequate

Wikipedia

# Big Data

Hulu

Imports 20GB per second continuously

Celeste Project

55 terabytes of data processed in 15 minutes

# Intel Ruler

32 TB SSD

Rack mounted

1PB in 1U

1 Rack holds 42 PB

# Amazon AWS Snowball

80 Terabytes

# Amazon AWS Snowmobile

100 Petabytes

| Value | Metric | |
|---|---|---|
| 1000 | kB | kilobyte |
| $1000^2$ | MB | megabyte |
| $1000^3$ | GB | gigabyte |
| $1000^4$ | TB | terabyte |
| $1000^5$ | PB | **petabyte** |
| $1000^6$ | EB | exabyte |
| $1000^7$ | ZB | zettabyte |
| $1000^8$ | YB | yottabyte |

# Big Data 3-5 V’s

Volume

Large datasets

Clusters - Spark

Velocity

Real time or near-real time streams of data

Kafka

Variety

Different formats

Structured, Numeric, Unstructured, images, email, etc.

NoSQL

Cassandra

Variability

Data flows can be inconsistent

Veracity

Accuracy

Complexity

# Scaling to Handle Large Data Sets

Scaling up (Vertically)

- Add more resources to single machine
- Memory, disk space, faster processor, etc
- Easier that scaling out but limited
- Amazon AWS has servers with 2 TB of memory

Scaling out (Horizontally)

- Using multiple machines/processors
- Adds complexity

# Scaling Up & Amdahl's Law

T(1) be the time it takes a sequential program to run
T(N) be the time it takes a parallel version of the program to run on N processors.

Speedup using N processors

$$S(N) = T(1)/T(N)$$

Let p = % of program that can be parallelized

Amdahl's Law

$$S(N) = 1/(1 - p + p/N)$$

# Amdahl's Law

Let p = % of program that can be parallelized

Amdahl's Law

$$S(N) = 1/(1 - p + p/N)$$

p = 1

$S(N) = 1/(1 - 1 + 1/N)$

$= 1/(1/N)$

$= N$

p = 0

$S(N) = 1/(1 - 0 + 0/N)$

$= 1$

# Amdahl’s Law

Let p = % of program that can be parallelized

Amdahl’s Law

$$S(N) = 1/(1 - p + p/N)$$

Given p = 0.5 how many processors does in make sense to use?

What does p have to be to get a speedup of

- 5 or greater using 10 processors?
- 10 or greater using 20 processors?
- 20 or greater using 40 processors?
- 50 or greater using 100 processors?

# Issues

What types of problems can be solved using cluster of commodity computers?

- When are setup time and communication time too high?
- How many machines?

How to distribute data?

How to find the data?

What to do when machine fails?

How to distribute computation? Load balancing?

How to share computation?

- Send computation result from node A to node B
- How does node B wait? How long is B idle?

How to combine results

Performance tuning

# Pleasingly Parallel

Compute Sum

2 -3 5 9 1 7 8 2 1 6

2 -3 5 9 1

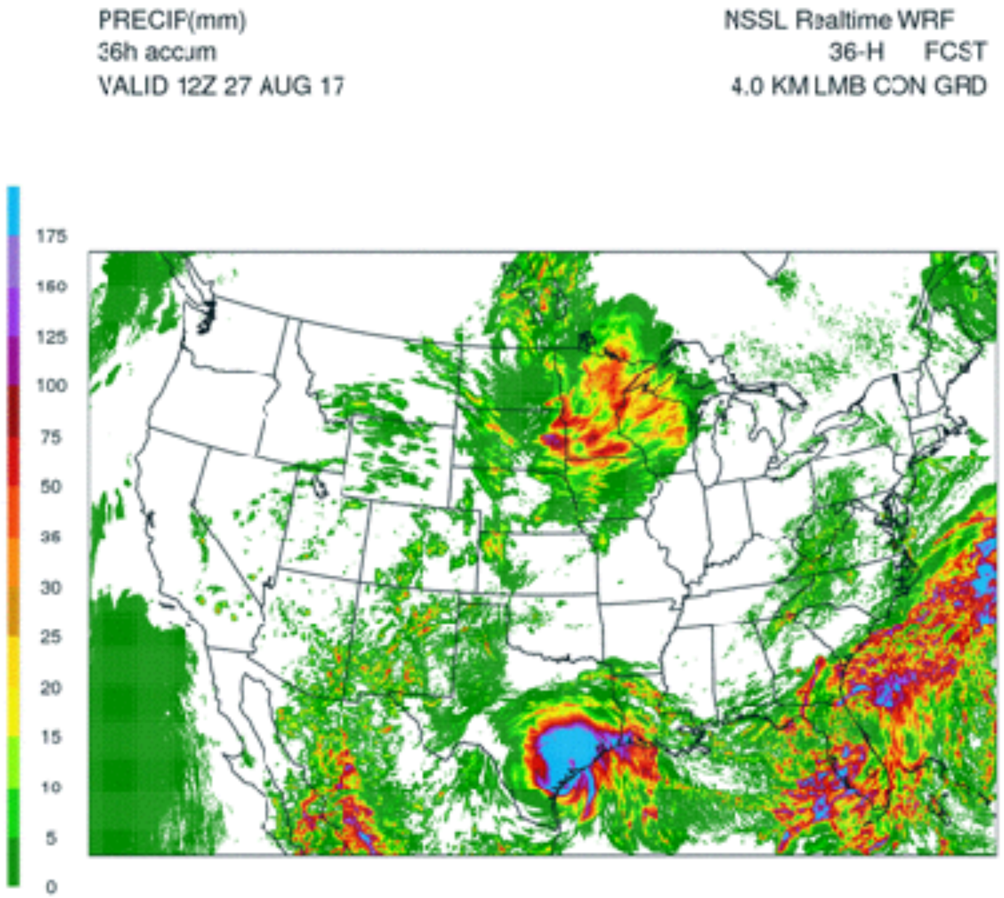
7 8 2 1 6

14

24

38

# Weather Simulation




Create 4km grid

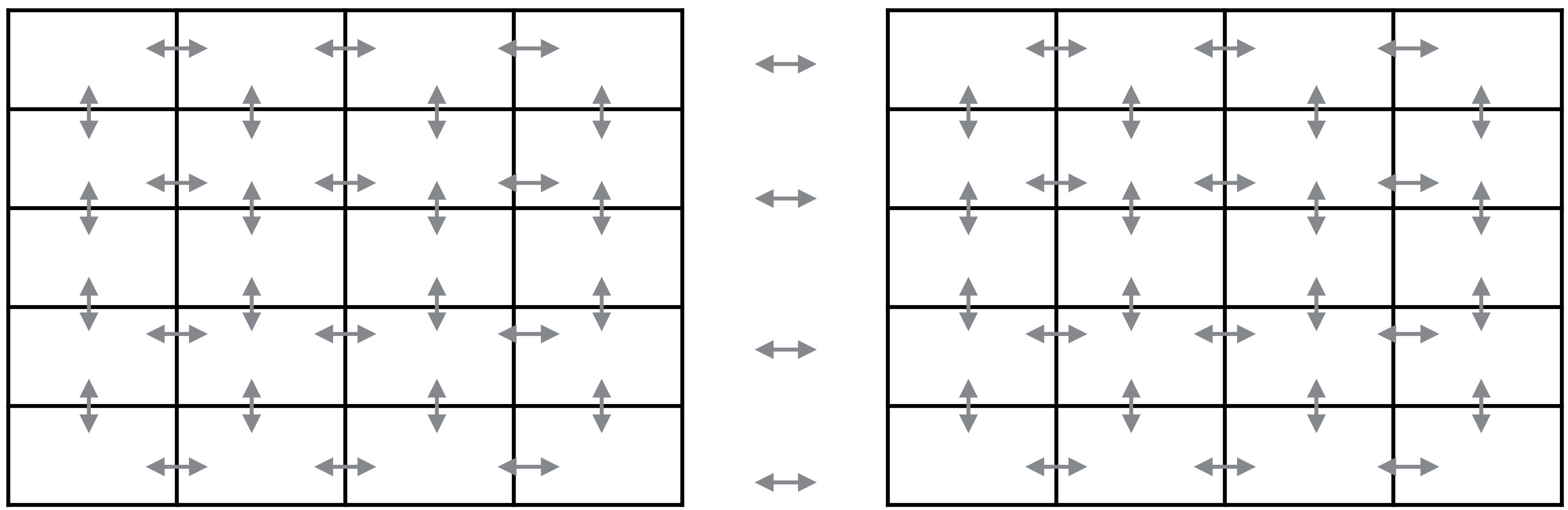
24 second time steps

35 vertical layers

Each time step

Compute effect of rain solar radiation in each square in grid

Propagate effect of change to neighboring grid cells and layers



Processor 1

Processor 2

# How to Distribute Data & Computation

Automate as much as possible

Want to run code on different number of nodes at different times
- Code should be independent of number of nodes

Node B should not know about Node C
- Is there a node C?
- Which is node B? C?

# Example

```
val data = readDataIntoArray(xxx)


var sum = 0

for (k <- 0 to data.length)
  sum += data(k)
```

```
val sum = data.reduce(_ + _)
```

Compiler issue

Has to handle all possible loop contents

Has to know where data is located

Library issue

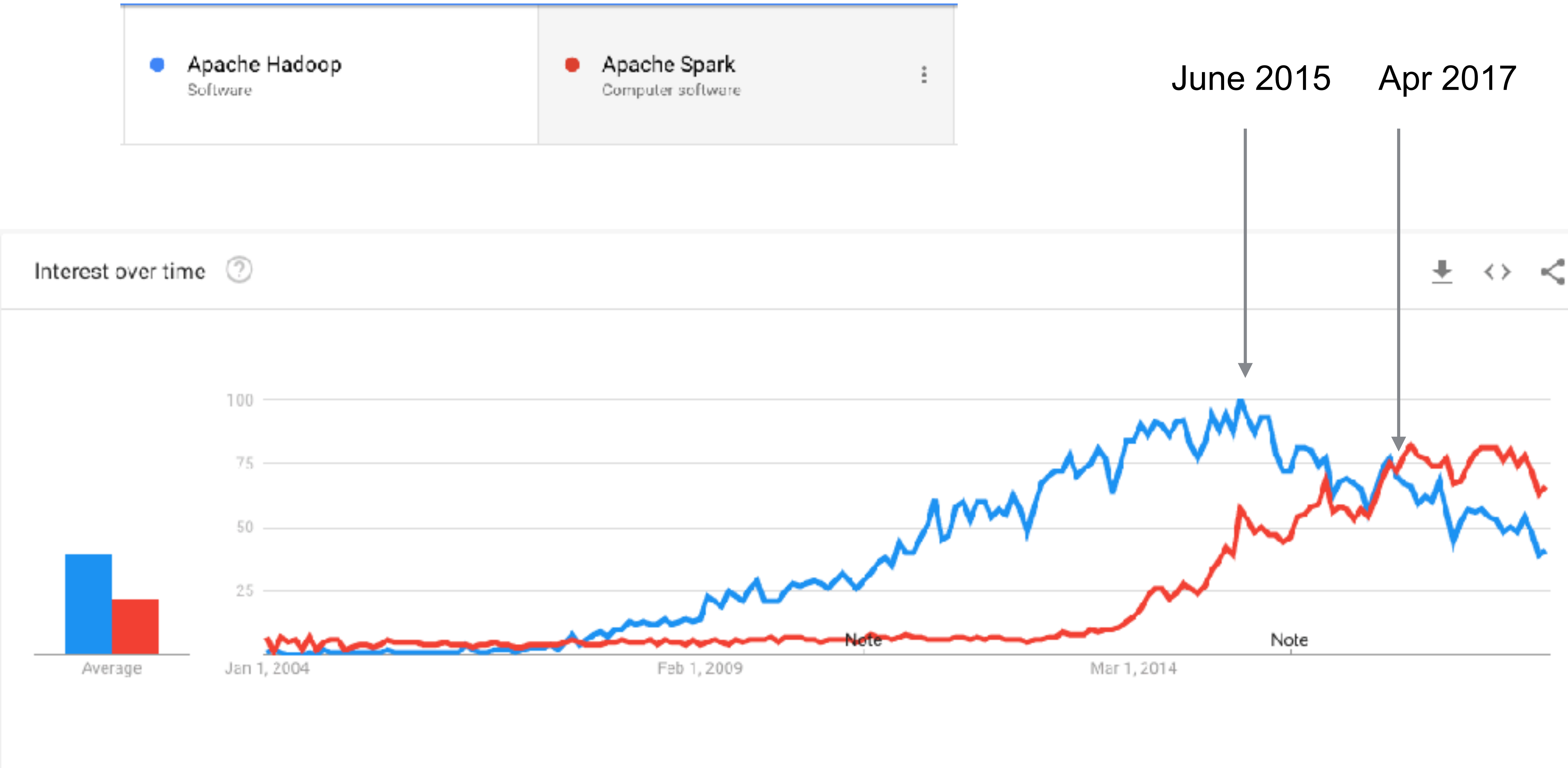
Handle one case

No direct access to array index

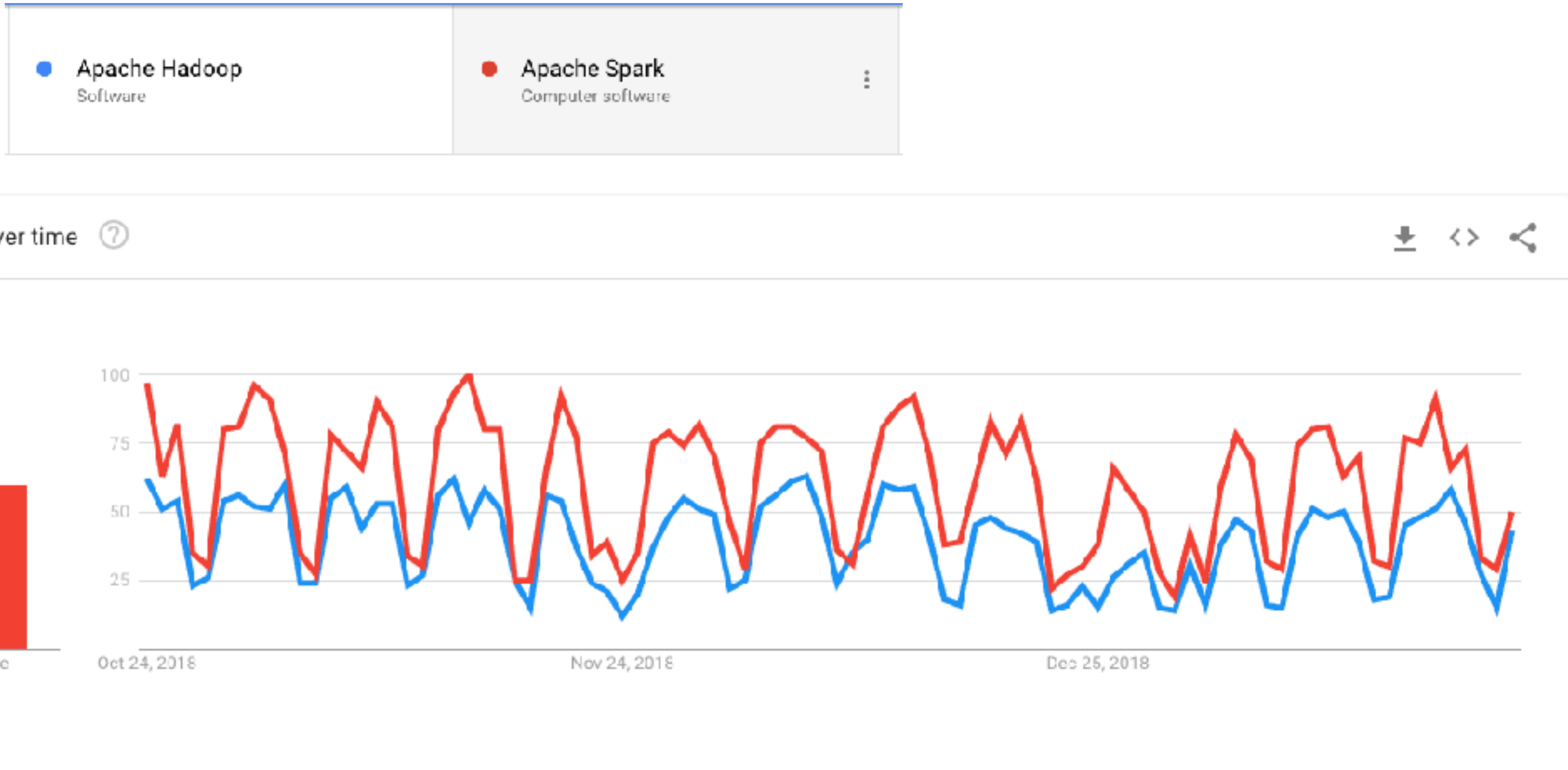
Library can distribute data

```
for (k <- 0 to data.length/2)
  sum += data(k) + data(data.length - k -1)
```

# Hadoop vs Spark



# What is Going On



# Latency numbers every programmer should know

```
L1 cache reference ......................... 0.5 ns
Branch mispredict ............................ 5 ns
L2 cache reference ........................... 7 ns
Mutex lock/unlock ........................... 25 ns
Main memory reference ...................... 100 ns
Compress 1K bytes with Zippy ............. 3,000 ns  =   3 µs
Send 2K bytes over 1 Gbps network ....... 20,000 ns  =  20 µs
SSD random read ........................ 150,000 ns  = 150 µs
Read 1 MB sequentially from memory ..... 250,000 ns  = 250 µs
Round trip within same datacenter ...... 500,000 ns  = 0.5 ms
Read 1 MB sequentially from SSD* ..... 1,000,000 ns  =   1 ms
Disk seek ........................... 10,000,000 ns  =  10 ms
Read 1 MB sequentially from disk .... 20,000,000 ns  =  20 ms
Send packet CA->Netherlands->CA .... 150,000,000 ns  = 150 ms
```

# Multiply by 1 Billion

Minute:

```
L1 cache reference        0.5 s            One heart beat (0.5 s)
Branch mispredict         5 s              Yawn
L2 cache reference        7 s              Long yawn
Mutex lock/unlock        25 s              Making a coffee
```

Hour:

```
Main memory reference        100 s       Brushing your teeth
Compress 1K bytes with Zippy  50 min     One episode of a TV show
```

```
Day:
Send 2K bytes over 1 Gbps network   5.5 hr
```

# Multiply by 1 Billion

Week

```
SSD random read                        1.7 days
Read 1 MB sequentially from memory     2.9 days
Round trip within same datacenter      5.8 days
Read 1 MB sequentially from SSD       11.6 days
```

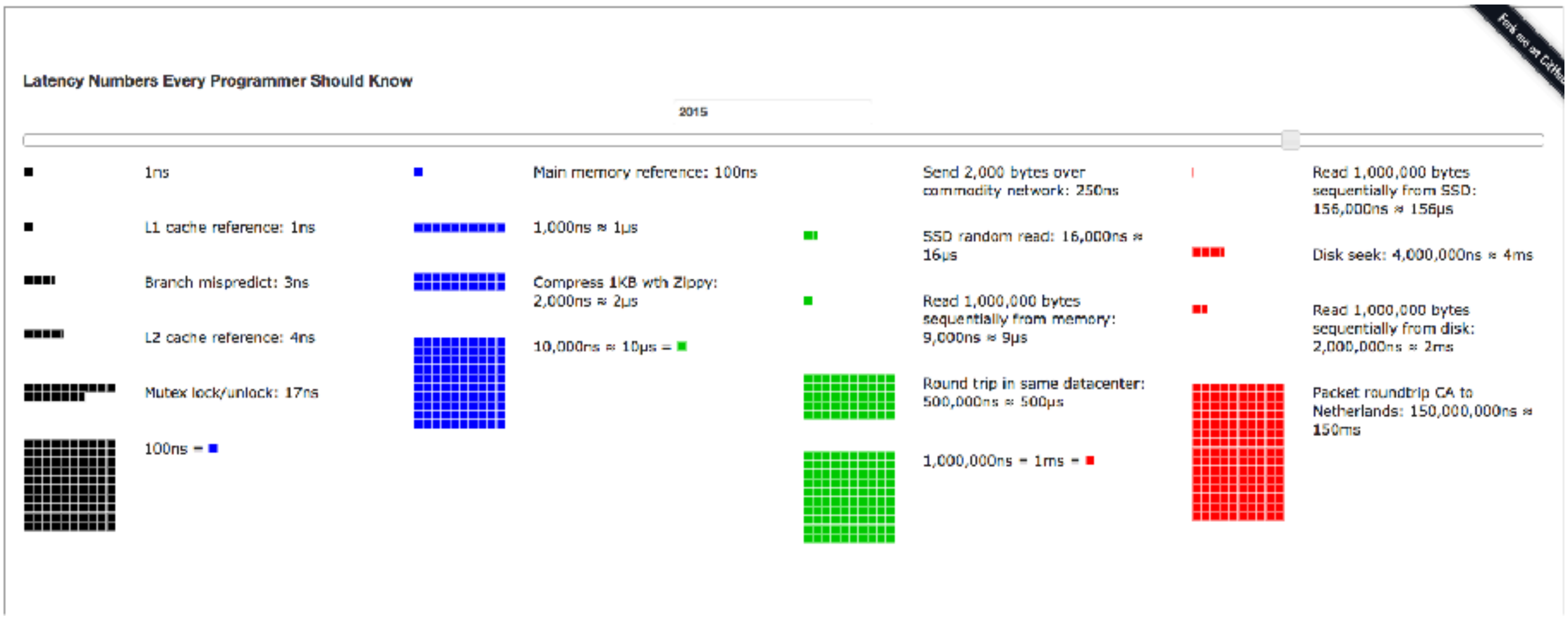
Year

```
Disk seek                              16.5 weeks
Read 1 MB sequentially from disk       7.8 months
The above 2 together                   1 year
```

Decade

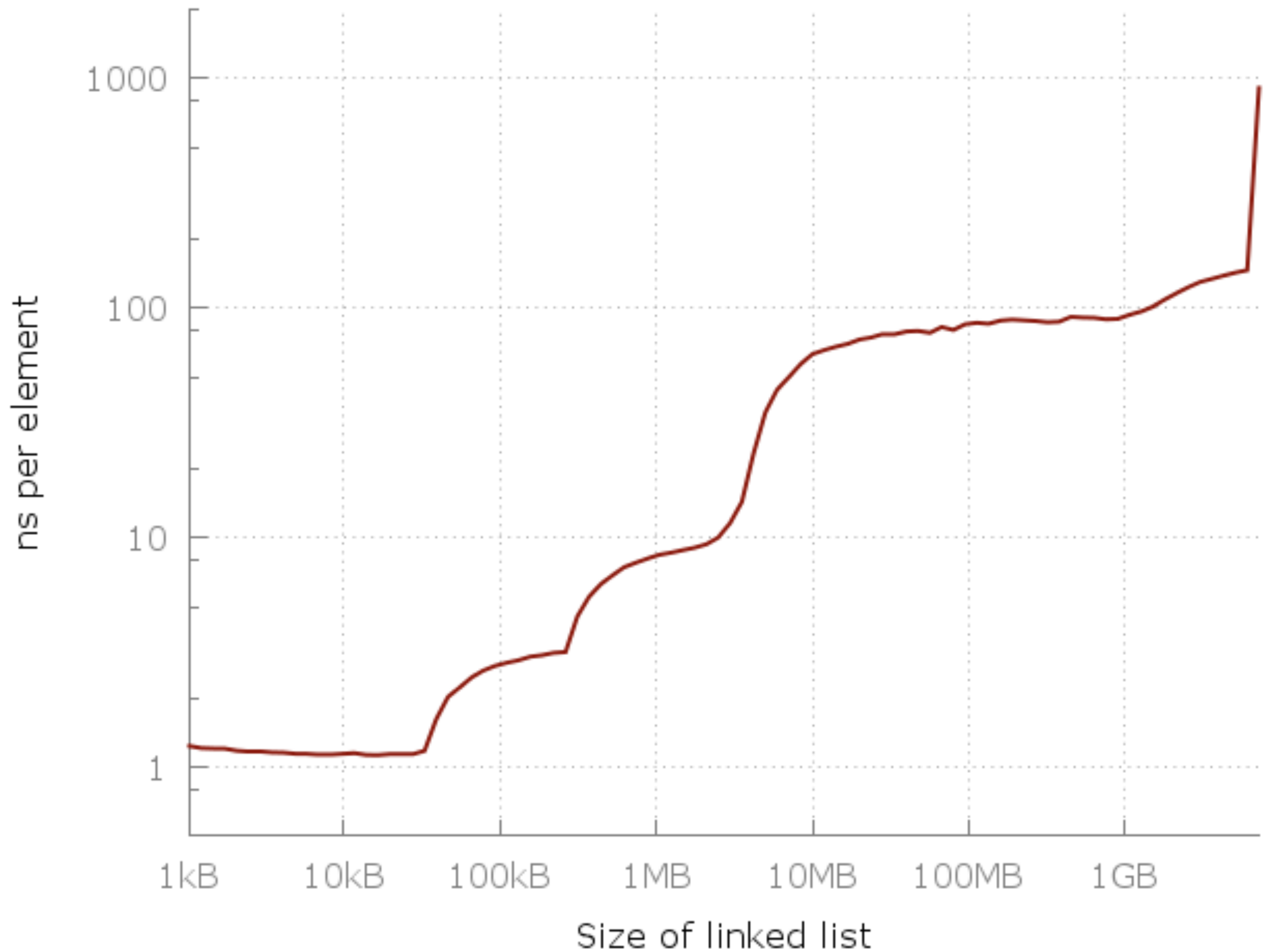
```
Send packet CA->Netherlands->CA        4.8 years
```

https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html



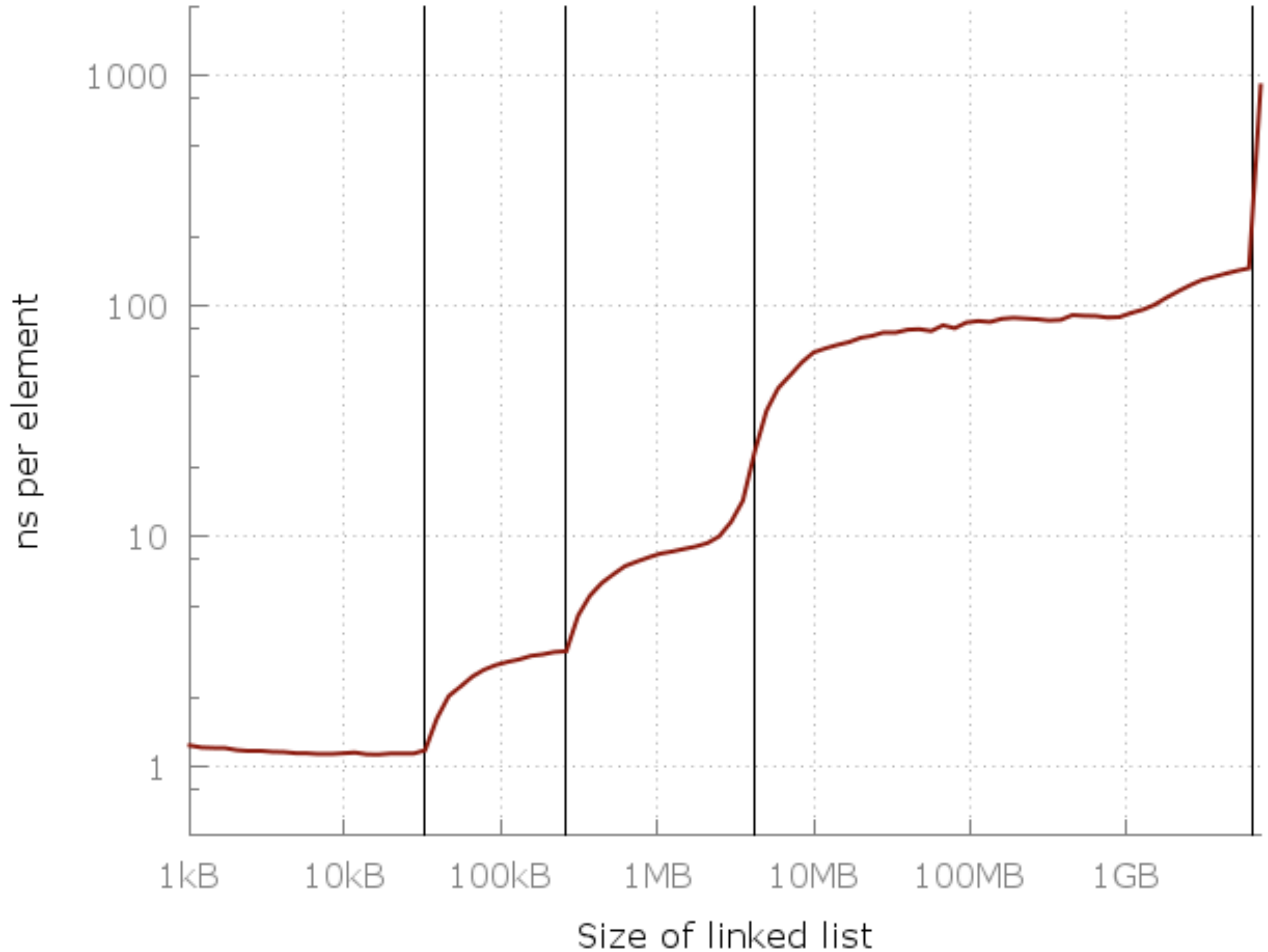
# Myth of Ram Access Being O(1)

http://goo.gl/JwtF5v



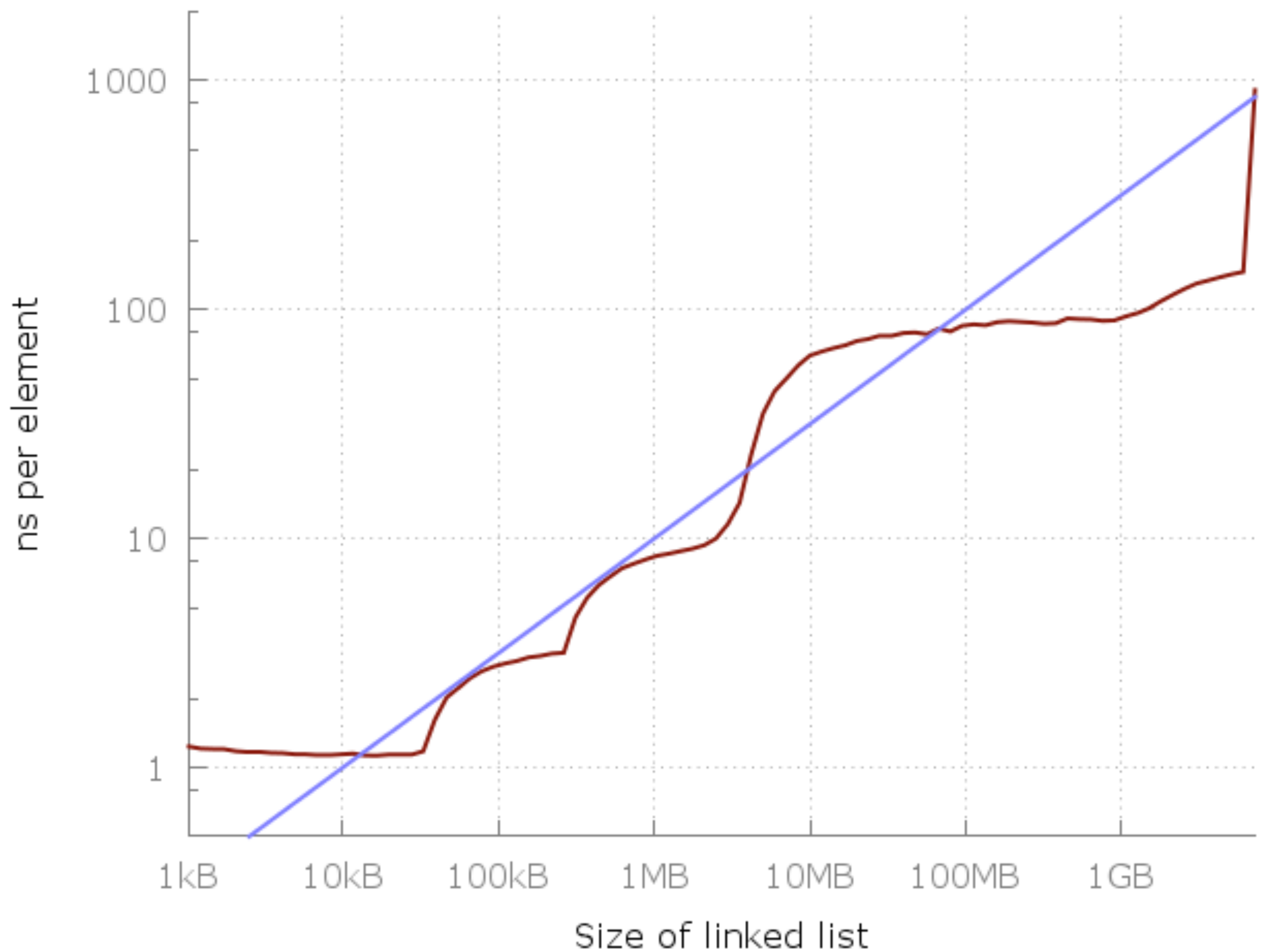
# Myth of Ram Access Being O(1)

Lines - L1=32kiB, L2=256kiB, L3=4MB and 6 GiB of free RAM



# Myth of Ram Access Being O(1)

Blue Line = $O(\sqrt{N})$



# History

| | | 1GB Ram |
|---|---|---|
| 1990 | | $103,880 |
| 1995 - Java 1.0 | Haskell (92) | $30,875 |
| 2000 - Java 3 | | $1,107 |
| 2001 - | Scala started | |
| 2002 - Nutch (Hadoop) started | | |
| 2004 - Google MapReduce paper | Scala v1 | |
| 2005 - | F# | $189 |
| 2006 - Hadoop split from Nutch | Scala v2 | |
| 2007 - | Clojure | |
| 2009 - Spark started | | |
| 2010 | Scala on Tiobe index | $12 |
| 2012 - Hadoop 1.0 | | |
| 2014 - Spark 1.0 | | |
| 2015 | | $4 |

# Hadoop

Hadoop Distributed File System (HDSF)

Map Reduce

# Hadoop MapReduce vs Spark

Spark - 10 to 100 time faster

- Hadoop stores data on disk
- Spark keeps as much data in memory as possible

Spark

- Has much more functionality
- Uses most functional programming
- Hadoop only uses Map & Reduce

Spark

- Easier to use
- REPL

# Two Language Problem