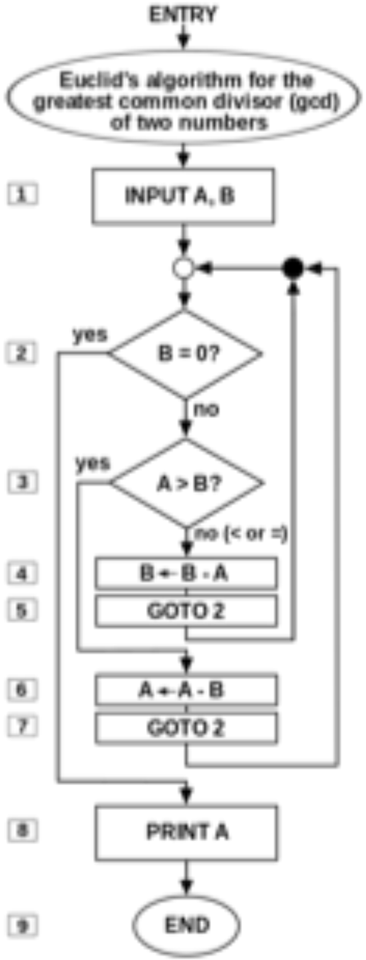
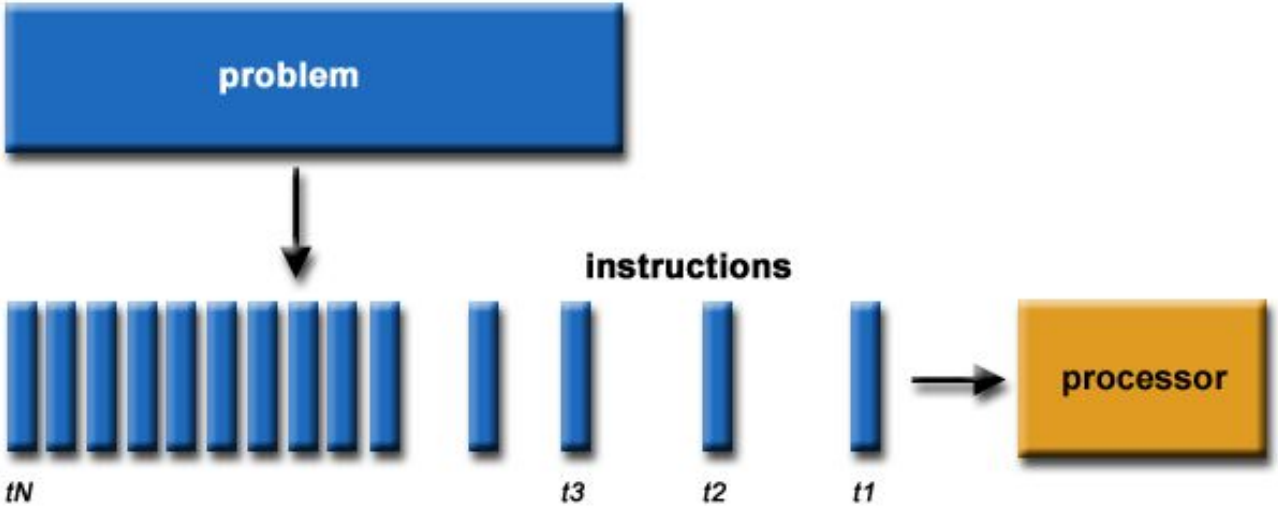
Parallel, Functional & Streaming Programming with Scala

Introduction to PARALLEL COMPUTING

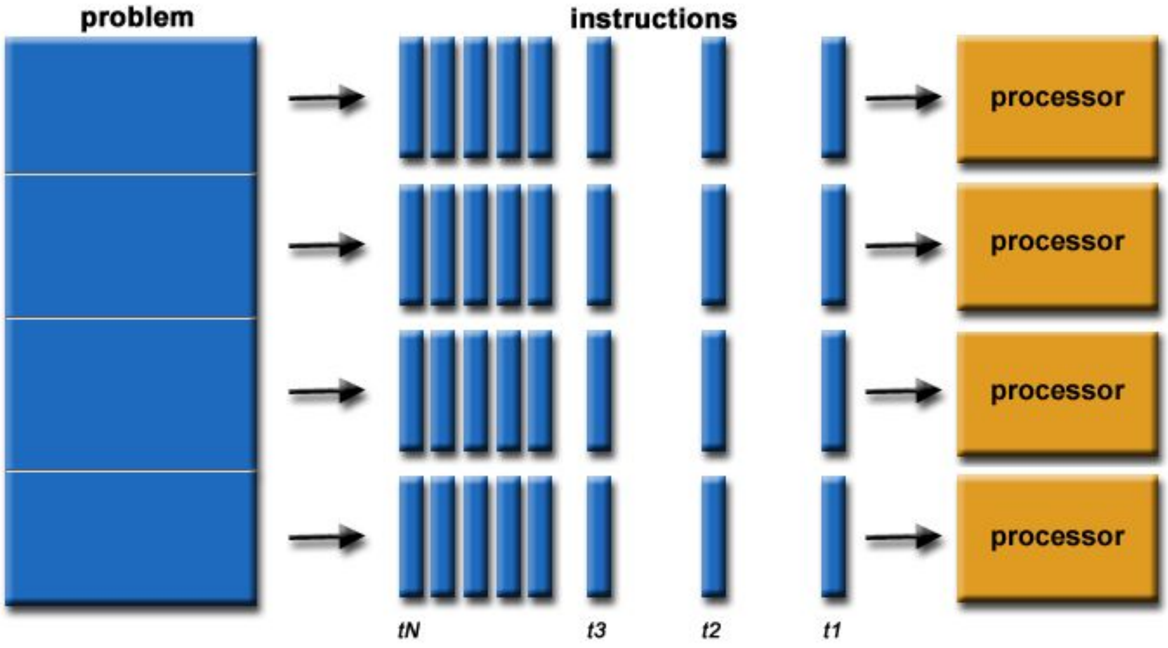
Turing machine



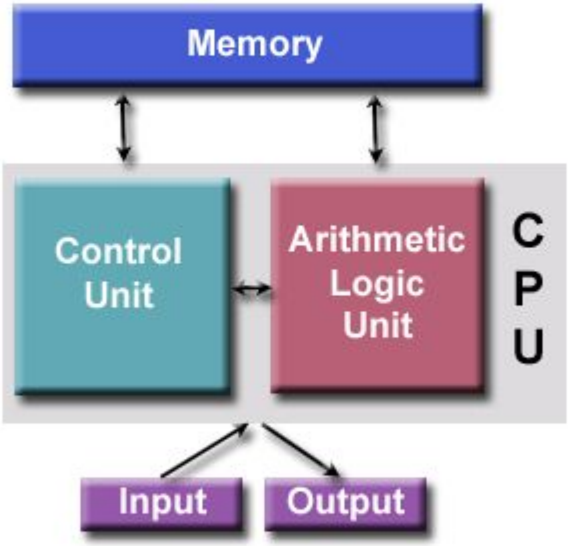
Sequential computing



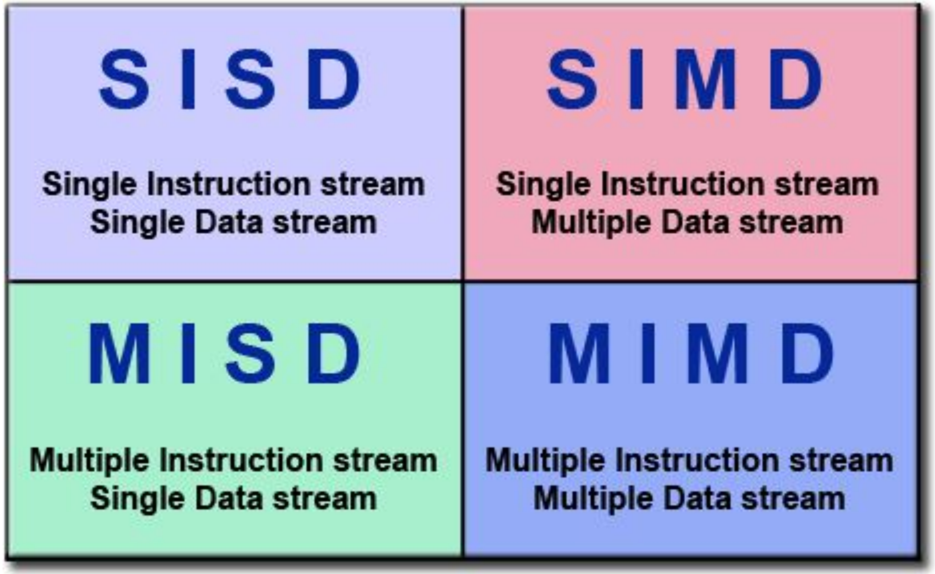
Parallel programming



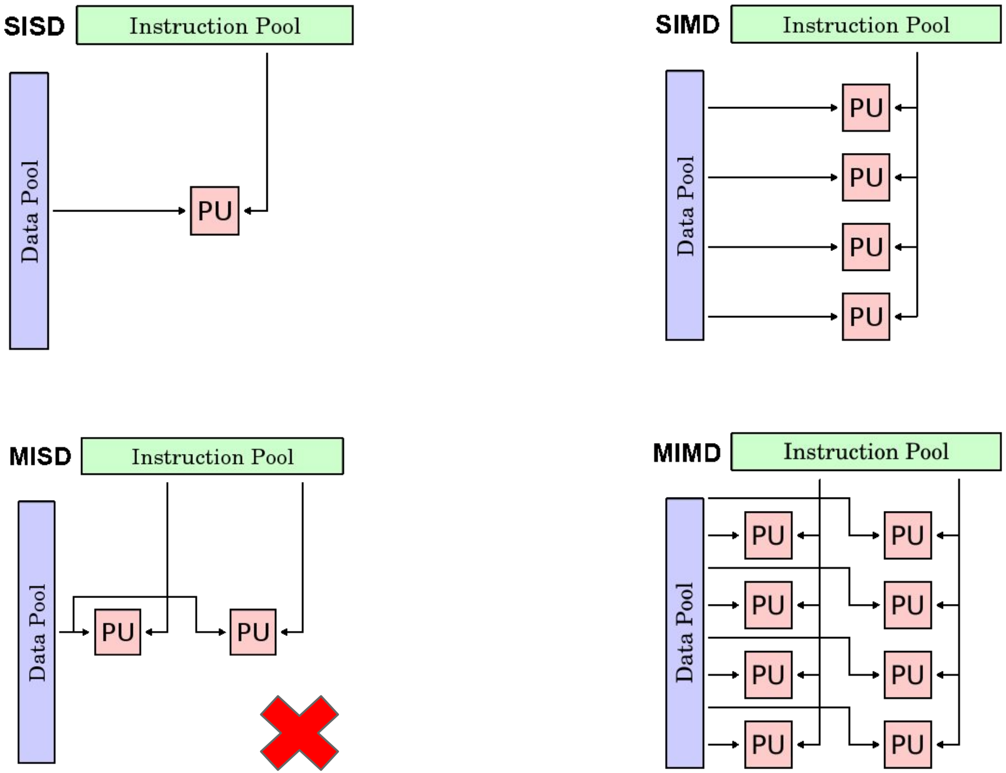
Von Neumann architecture



Flynn taxonomy



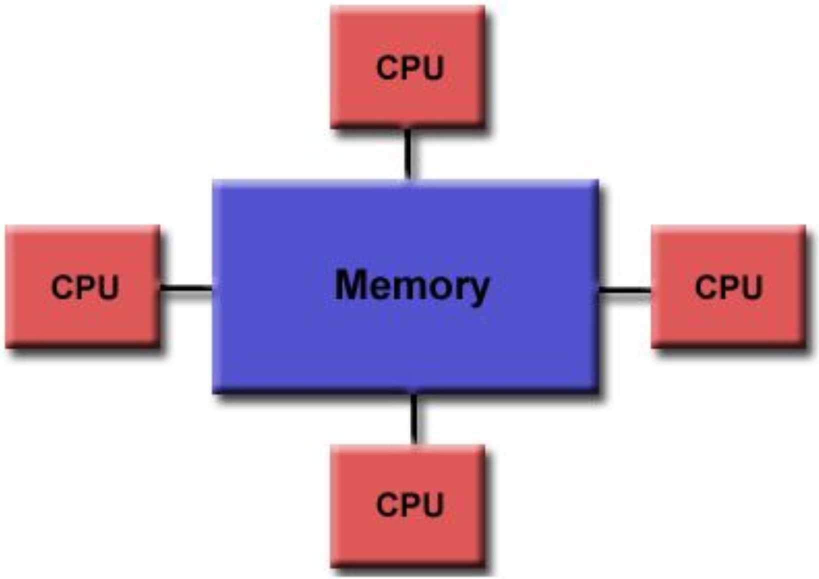
Flynn taxonomy



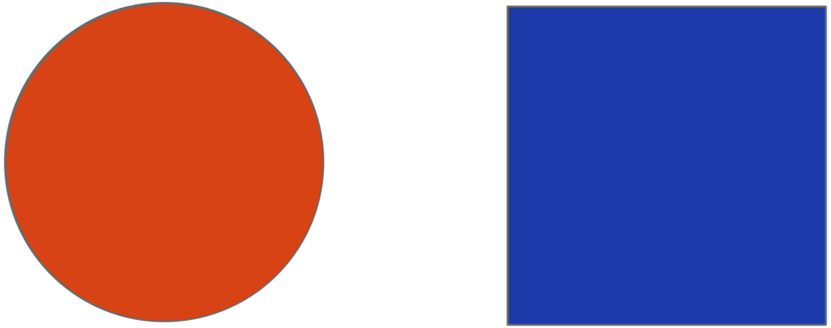
Three basic models

1. Shared memory
2. Network
3. Directed acyclic graph (DAG)

Shared memory



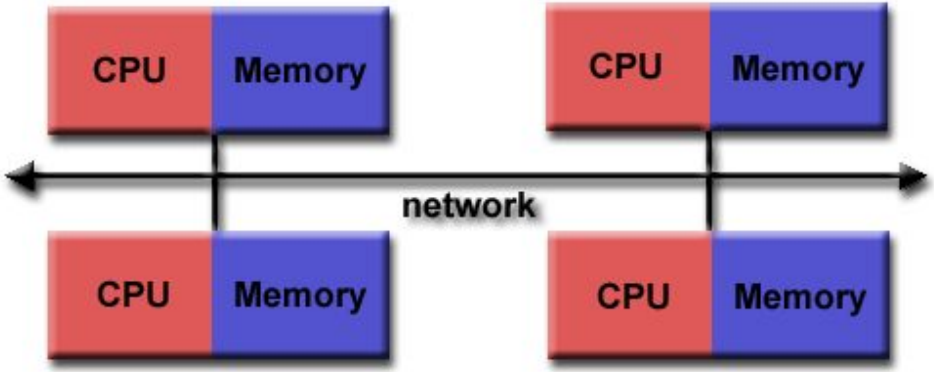
Elementals



state

process

Distributed memory



Elementals

state process message

Stateless vs stateful

Stateless:

What is the current temperature?

How many shares of IBM did he sell?

Stateful:

What is the change in temperature over the past hour?

How many shares of IBM does he currently own?

Datastore

Does not matter if you are

- In memory
- On disk
- In a database
- Distribute cache
- NOSQL
- On a message bus

```
store.put(x,y)

val y = store.get(x)
```

Multiple thread example

```scala
class Ticker(id: String) extends Runnable {
 def run: Unit = {
   var x = 0
   val pause = scala.util.Random.nextInt(1000)
   while (true) {
     x = x + 1
     System.out.println(id + ": " + x)
     Thread.sleep(pause)
   }
 }
}

object ParallelOne extends App{
 val names = List("A","B","C","D","E")
 for (name <- names) {
   new Thread(new Ticker(name)).start
 }
}
```

Sample output

Execution sequence is different than the coded sequence

Each thread emits events on different schedules

A: 1
D: 1
E: 1
C: 1
B: 1
A: 2
D: 2
A: 3
B: 2
A: 4
D: 3
A: 5
C: 2
B: 3
E: 2
A: 6

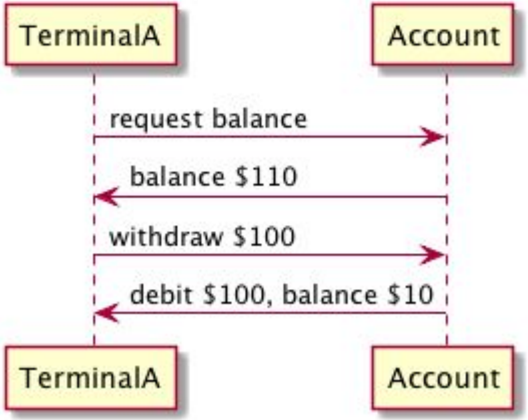
D: 4
A: 7
B: 4
A: 8
D: 5
A: 9
C: 3
B: 5
D: 6
E: 3
A: 10
A: 11
D: 7
A: 12
B: 6
A: 13

C: 4
D: 8
A: 14
B: 7
E: 4
A: 15
D: 9
A: 16
B: 8
A: 17
D: 10
C: 5
A: 18
B: 9
D: 11
E: 5

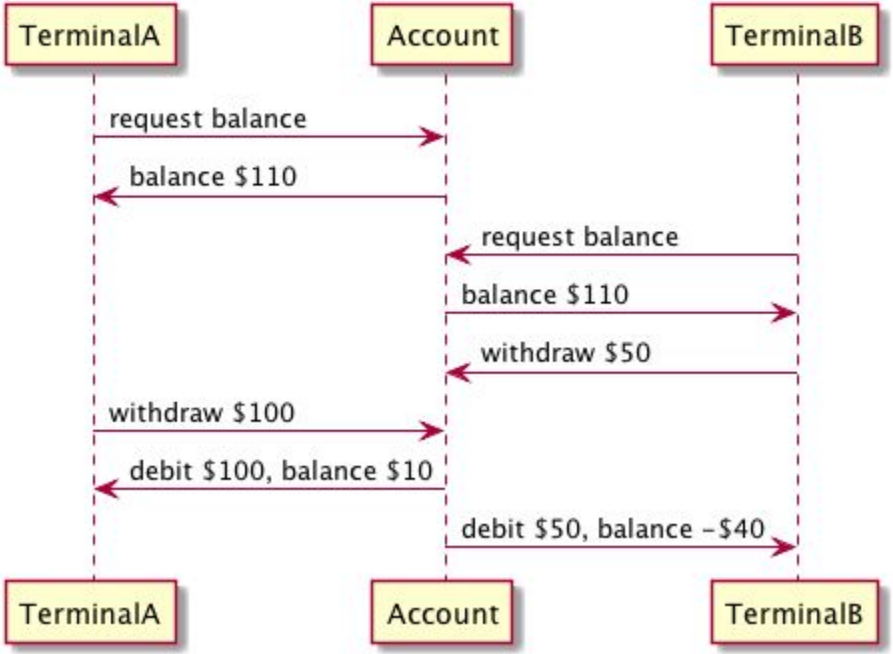
Fundamental challenge:

do not let different processes change the same state at the same time

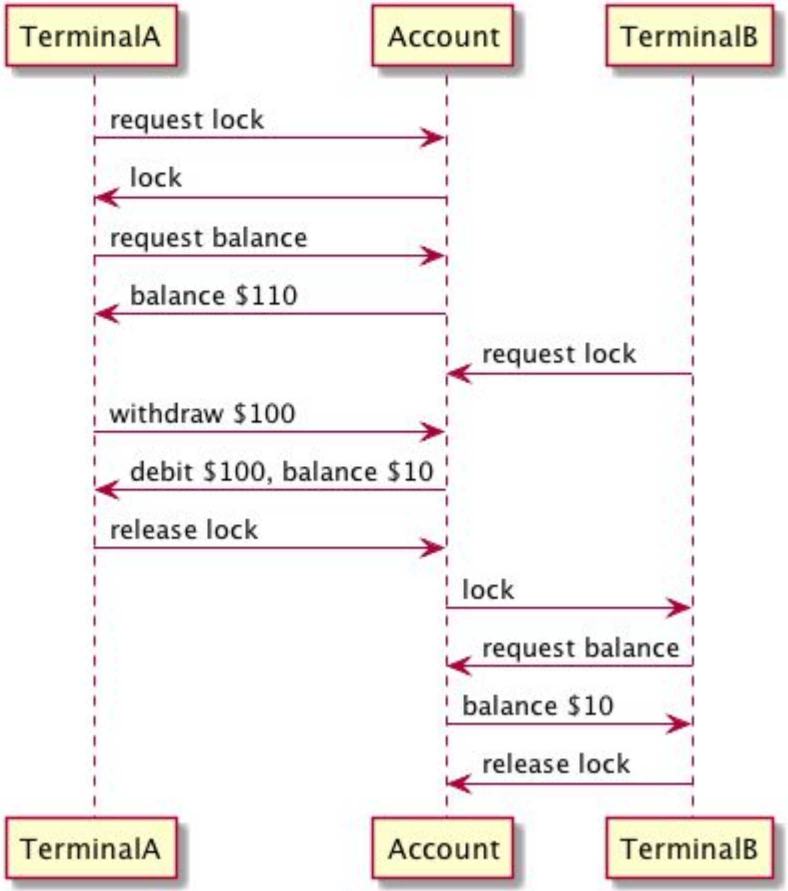
ATM example 1



ATM Example 2

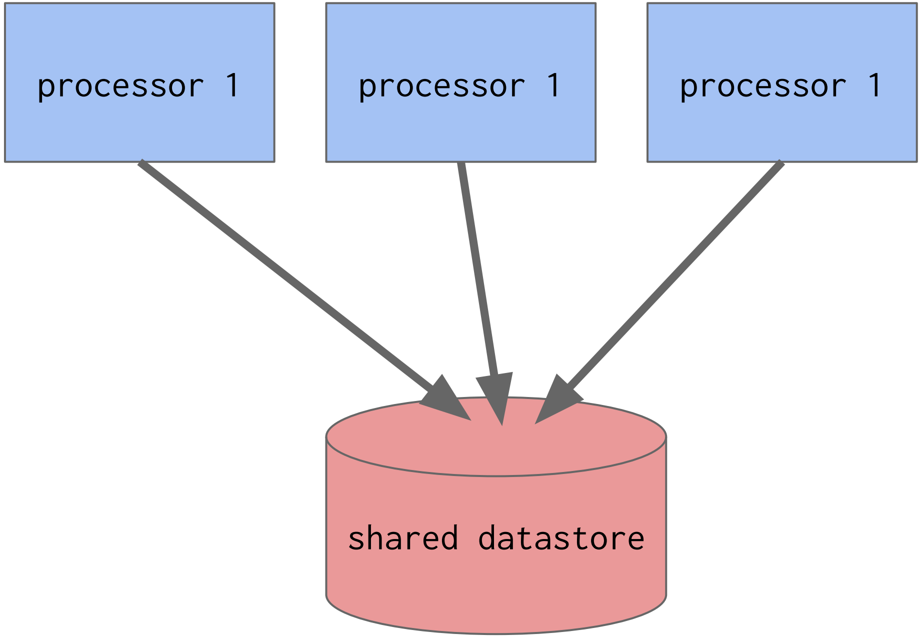


Lock (or mutex)



Shared datastore

Process in multiple machines and let the database handle the data consistency



ACID

Atomicity

Transactions succeed or fail completely

Consistency

Transactions change from one valid state to another

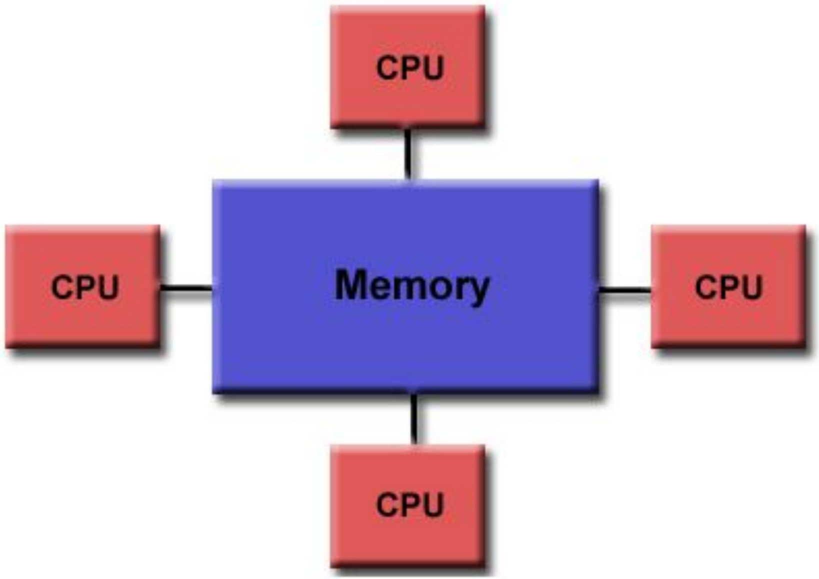
Isolation

Concurrency control between transactions

Durability

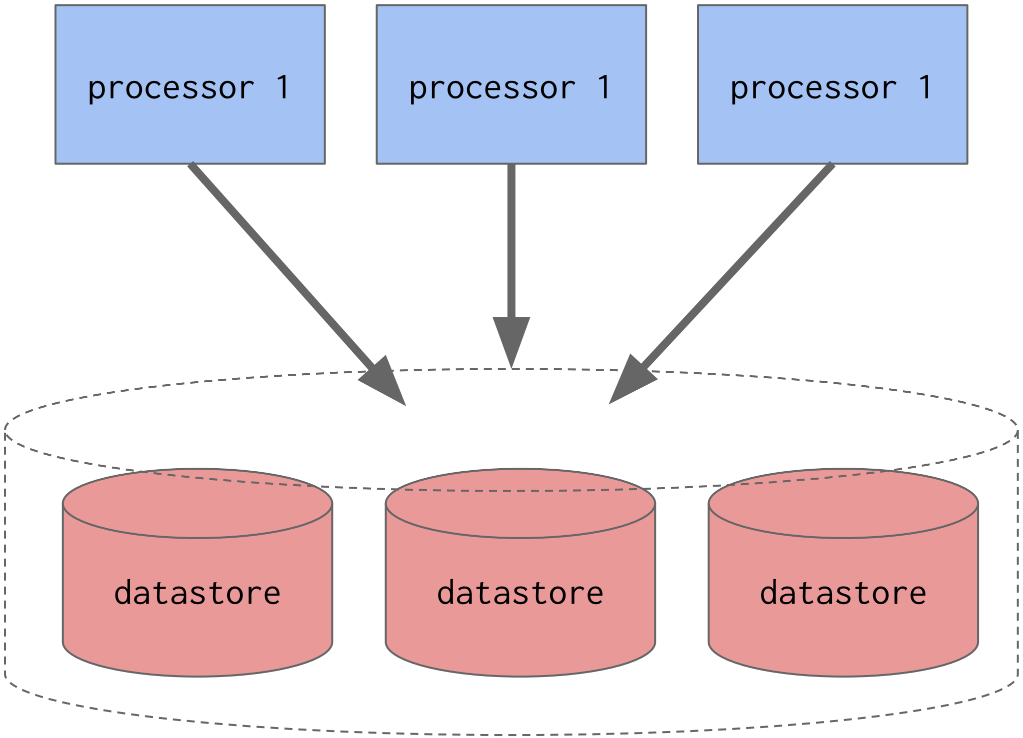
Non-volatile recording

Shared memory

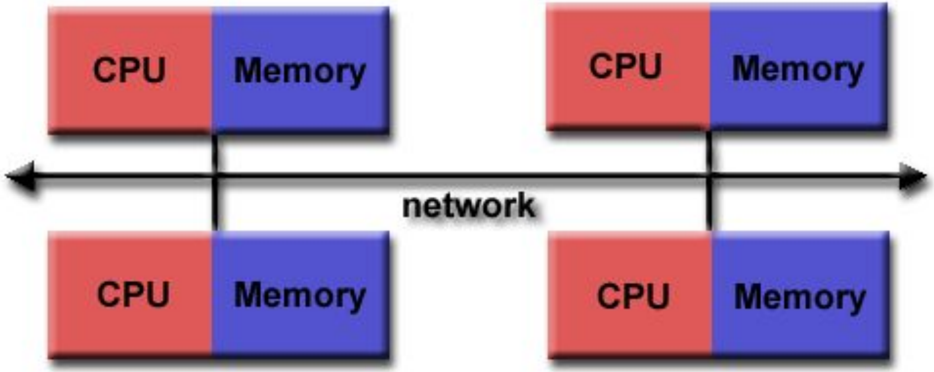


Distributed datastore

The state store is distributed



Distributed memory



Hurst's Law

Complexity can neither be created nor destroyed;
it can only be displaced.

Pay attention to where it went!

CAP theorem

Consistency**:

Every read receives the most recent write or an error

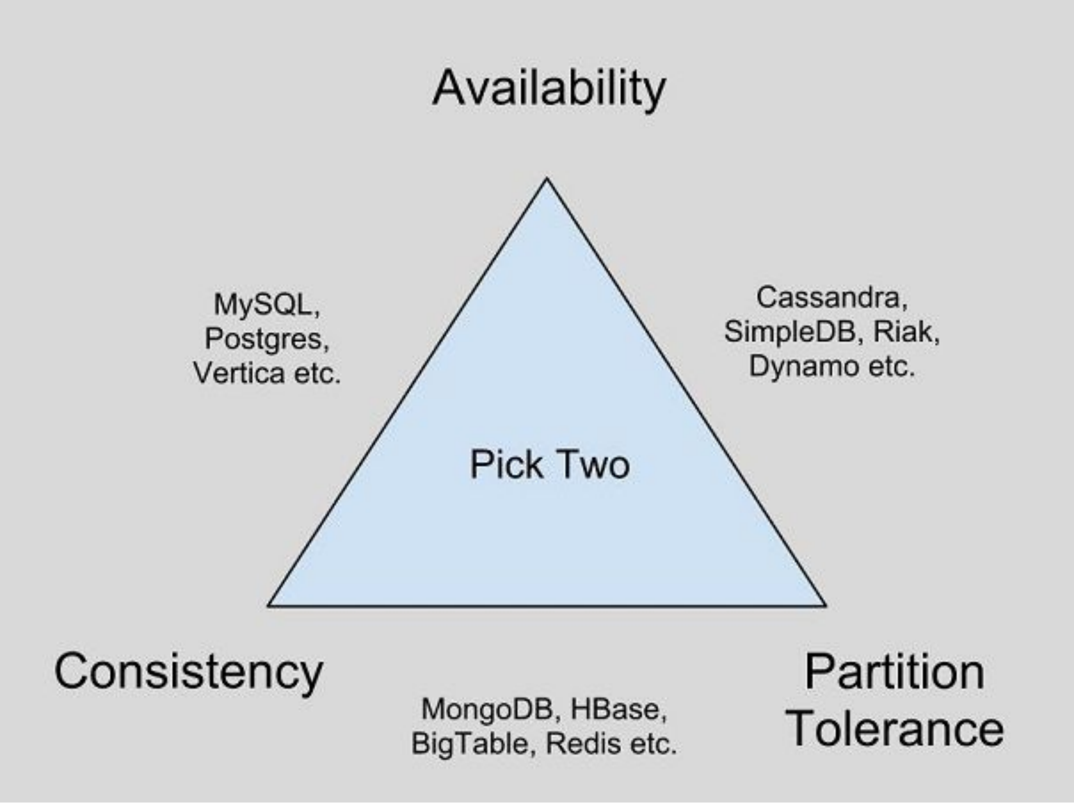
Availability:

Every request receives a response that is not an error

Partition tolerance:

The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes

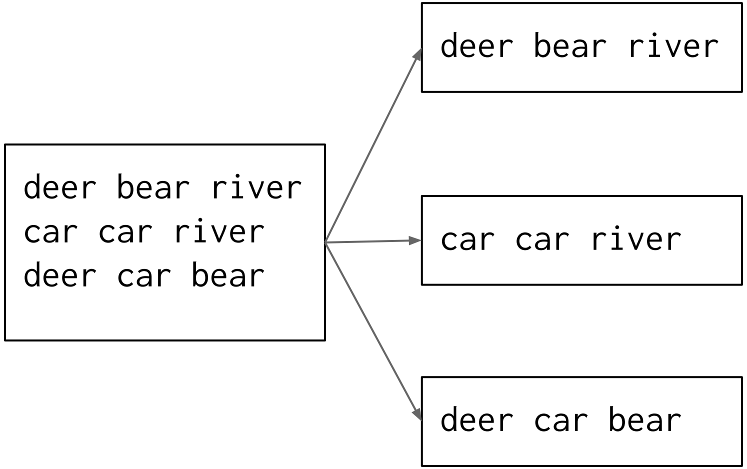
CAP theorem


Availability
MySQL,
Postgres,
Vertica etc.
Cassandra,
SimpleDB, Riak,
Dynamo etc.
Pick Two
Consistency
MongoDB, HBase,
BigTable, Redis etc.
Partition
Tolerance

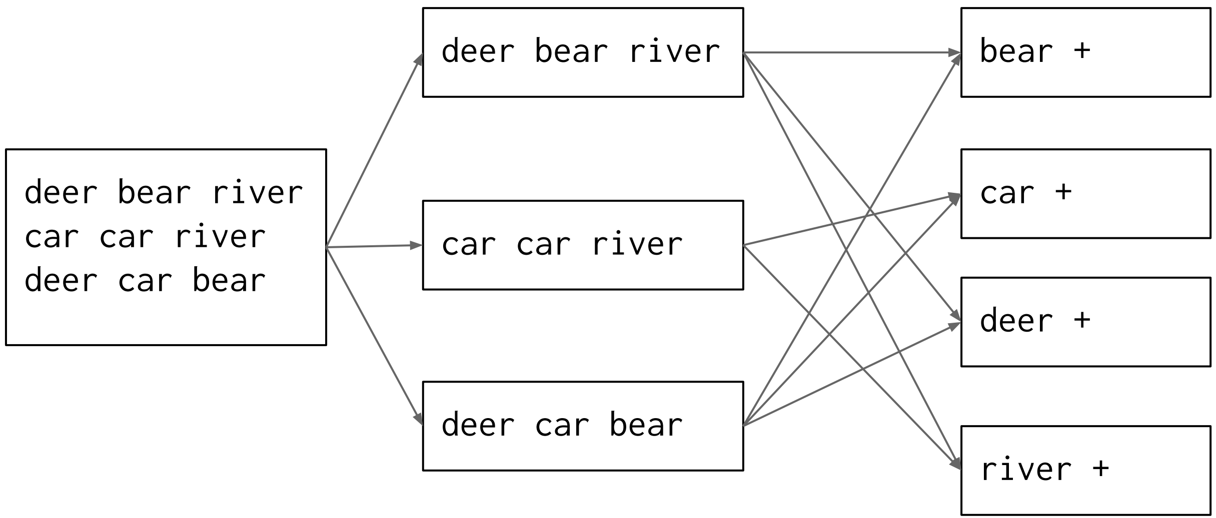
For <u>stateful parallel processing</u>,
you can either
<u>lock</u> or <u>partition</u>.

And locks are expensive.

Word count problem



Lock approach

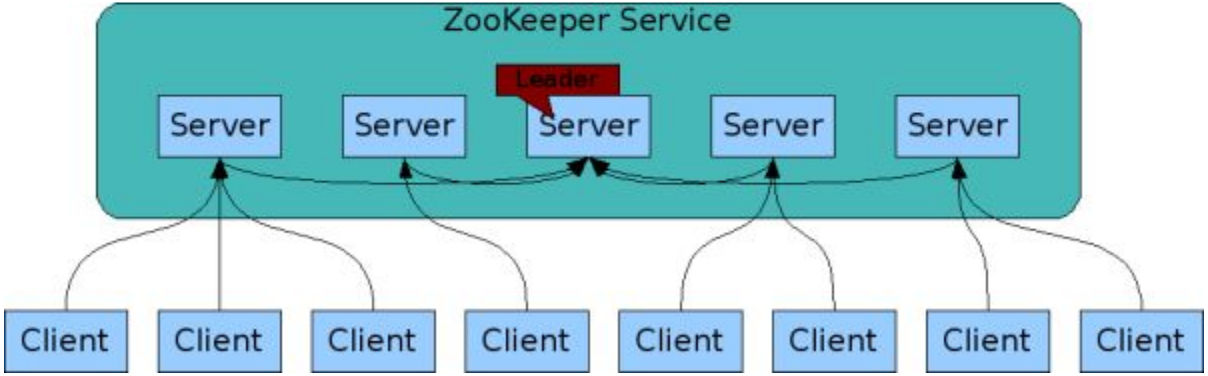


Zookeeper

Distributed coordination service for distributed applications

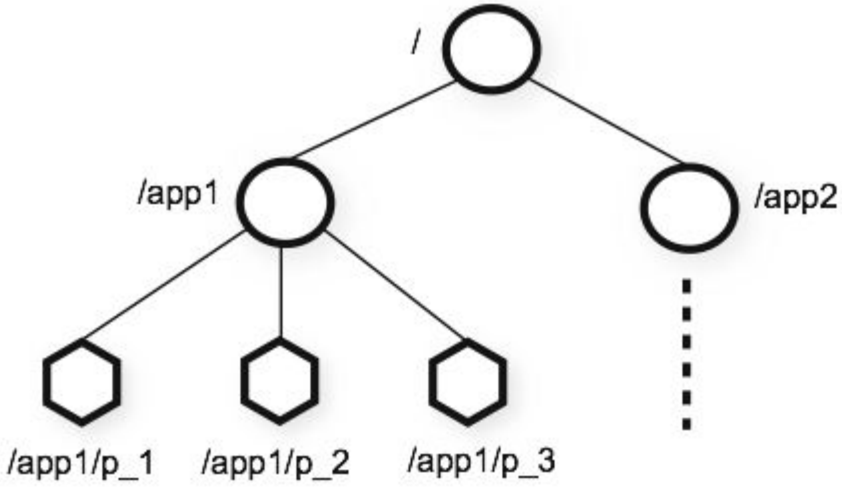
- Simple
- Fast
- Replicated
- Ordered
- Quorum
- Watches
- High availability

Zookeeper distributed service



Zookeeper hierarchical data structure

```
create
delete
exists
get data
set data
get children
sync
```



Sample use cases

Elect a leader

Name service

Load balance the partitions

Share configuration

Mutex

Pub/sub

Actor model

An actor is the primitive unit of computation.

Actors communicate with each other by sending asynchronous messages.

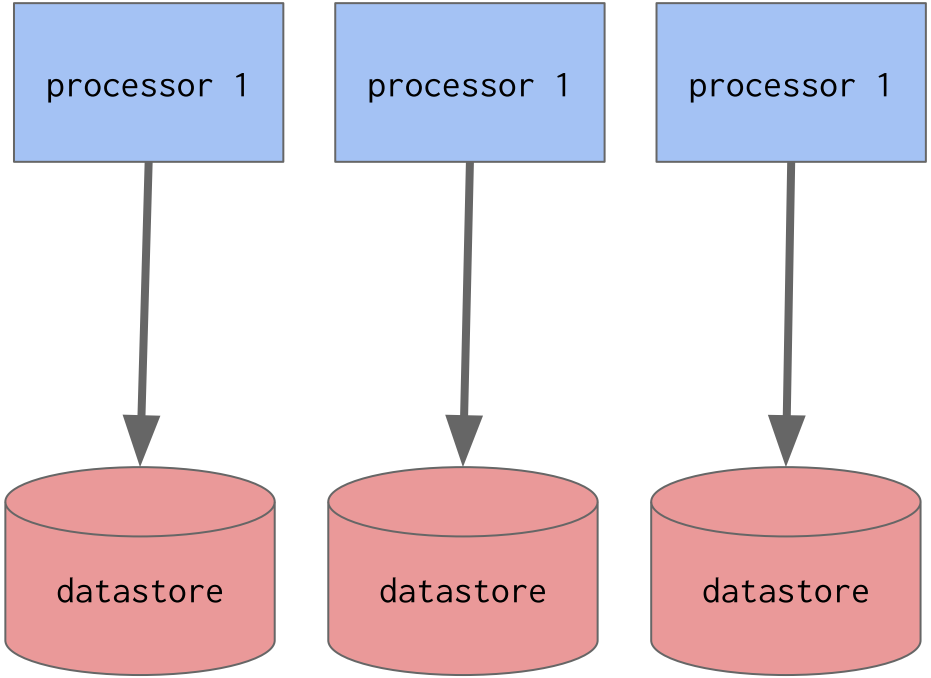
When an actor receives a message, it can do one of these 3 things:

- Create more actors
- Send messages to other actors
- Designate what to do with the next message

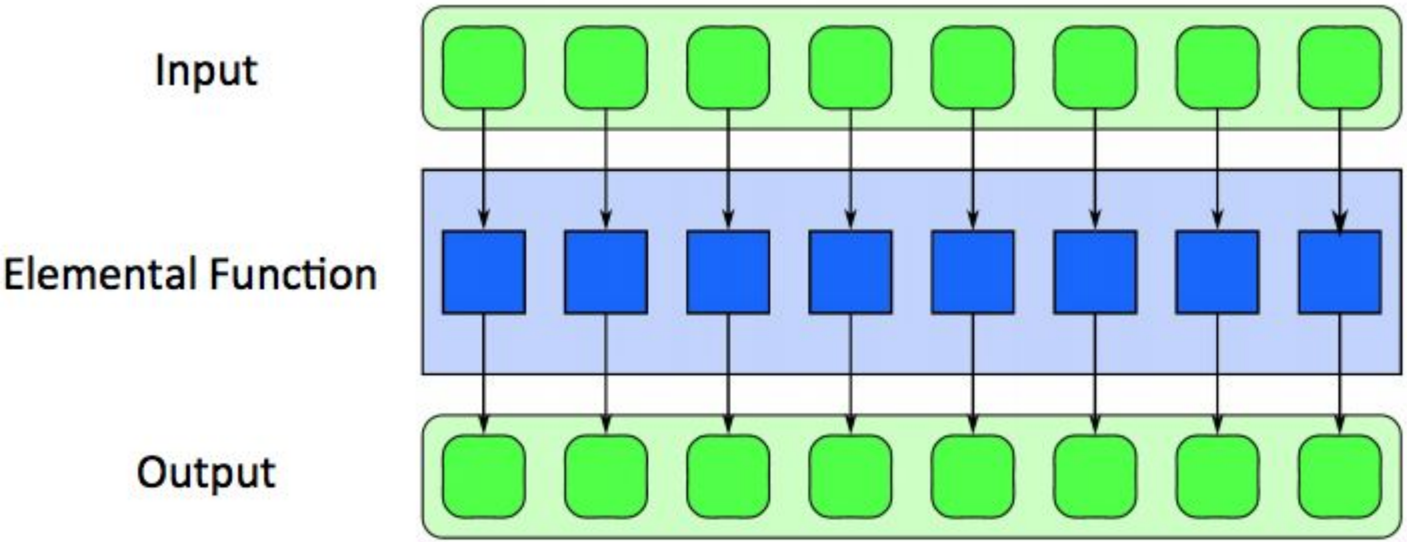
Actors have their own internal isolated state

Partitioned

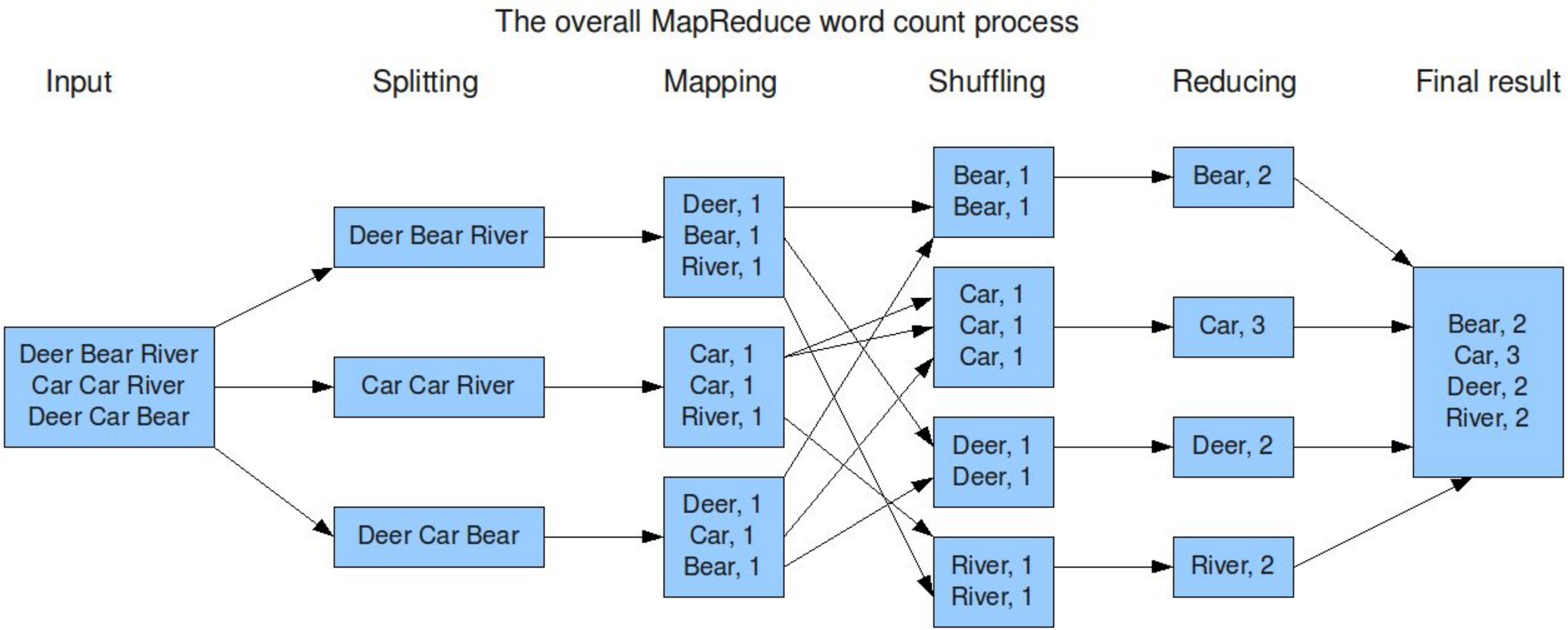
Use a different state store for each process



Map



Map reduce


The overall MapReduce word count process
Input
Splitting
Mapping
Shuffling
Reducing
Final result
Deer Bear River
Car Car River
Deer Car Bear
Deer Bear River
Car Car River
Deer Car Bear
Deer, 1
Bear, 1
River, 1
Car, 1
Car, 1
River, 1
Deer, 1
Car, 1
Bear, 1
Bear, 1
Bear, 1
Car, 1
Car, 1
Car, 1
Deer, 1
Deer, 1
River, 1
River, 1
Bear, 2
Car, 3
Deer, 2
River, 2
Bear, 2
Car, 3
Deer, 2
River, 2

2004 Google MapReduce

Resolved:

1. Parallelization – how to parallelize the computation
2. Distribution – how to distribute the data
3. Fault-tolerance – how to handle component failure

Move the program to the data.

There's simply too much data to be moved around.

MR example

Ticker data

Every 5 minutes

High, low, volume

Want the daily value weighted average price (VWAP)

```
<symbol>,<date>,<open>,<high>,<low>,<close>,<vol>
AAPL,201010110900,295.01,295.05,294.82,294.82,5235
MSFT,201010110900,67.23,67.70,67.04,67.65,72383
IBM,201010110900,100.20,100.34,100.20,100.31,8921
...
AAPL,201010110905,294.81,294.9,294.8,294.85,7441
...
```

Volumn weighted average price (VWAP)

$$VWAP(t_1, t_2) = \frac{\sum_{t=t_1}^{t_2} \delta V(t) P(t)}{\sum_{t=t_1}^{t_2} \delta V(t)}$$

```scala
case class Tick(symbol:String, date:String, time:String, open:Double, high:Double, low:Double, close:Double, volume:Int)
case class TickDate(date:String, symbol:String)
case class Vwap(price:Double,volume:Int)


class VwapMapper extends Mapper[Object,Text,TickDate,Tick] {
 def map(key:Object, value:Text, context:Context) = {
   val tick = parseTick(value)
   context.write( TickDate(tick.date, tick.symbol), tick)
 }
 def parseTick(value:Text): Tick = {
   Tick("","","",0,0,0,0,0) // TODO
 }
}

class VwapReducer extends Reducer[TickDate,Tick,TickDate,Double] {
 def reduce(key:TickDate, values:Seq[Tick], context:Context) = {
   val vwap = values.foldLeft(Vwap(0,0)) { (z, t) =>
     val price = (t.high + t.low)/2
     val totalVolume = z.volume + t.volume
     new Vwap((z.price * z.volume + price * t.volume)/totalVolume, totalVolume)
   }
   context.write(key,vwap)
 }
}
```

MR example

Mapper

Input:
<symbol>,<date>,<open>,<high>,<low>,<close>,<vol>
AAPL,201010110900,295.01,295.05,294.82,294.82,5235

Output:
<tickdate>,<tick>
20101011,AAPL
AAPL,201010110900,295.01,295.05,294.82,294.82,5235

Reducer

Input:
<tickdate>,Seq<tick>

Output:
<tickdate>,<vwap>
20101011,AAPL   293.23

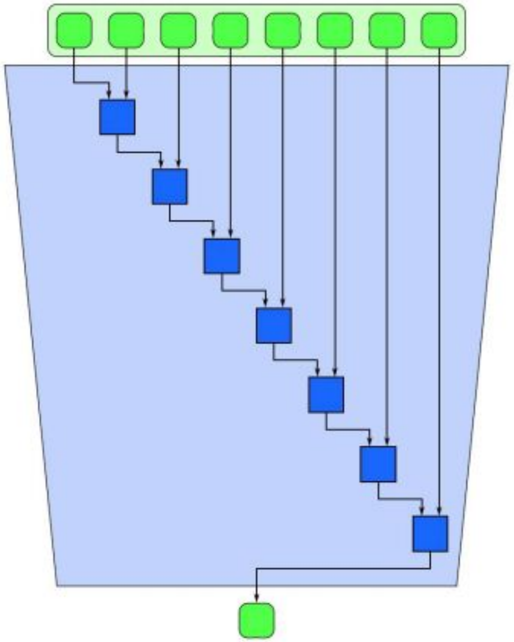
Map and reduce in Scala

```
scala> val a = List(1, 2, 3, 4, 5)

scala> a.map(x => x*2)
res0: List[Int] = List(2, 4, 6, 8, 10)

scala> def f(x:Int)= if (x>2) Some(x) else
None

scala> a.map(x => f(x))
res1: List[Option[Int]] = List(None, None,
Some(3), Some(4), Some(5))
```

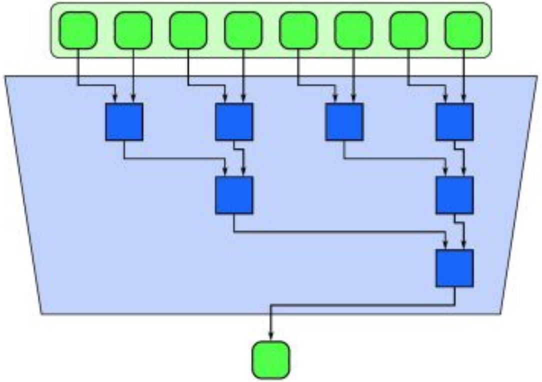
```
scala> val a = Array(12, 6, 15, 2, 20, 9)

scala> a.reduceLeft(_ + _)
res0: Int = 64

scala> a.reduceLeft(_ * _)
res1: Int = 388800

scala> a.reduceLeft(_ min _)
res2: Int = 2

scala> a.reduceLeft(_ max _)
res3: Int = 20
```

Reduction

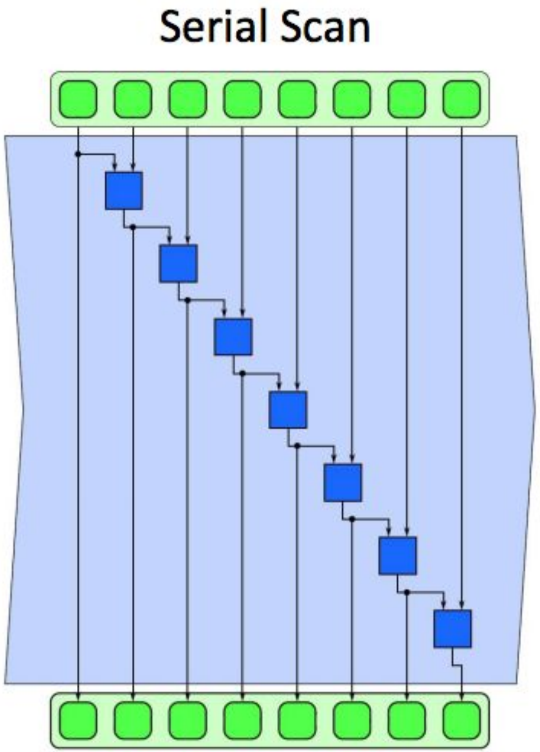
Serial Reduction

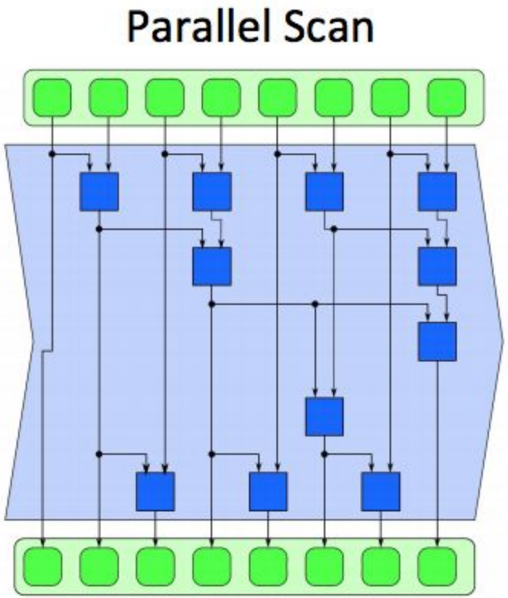


Parallel Reduction



Scan





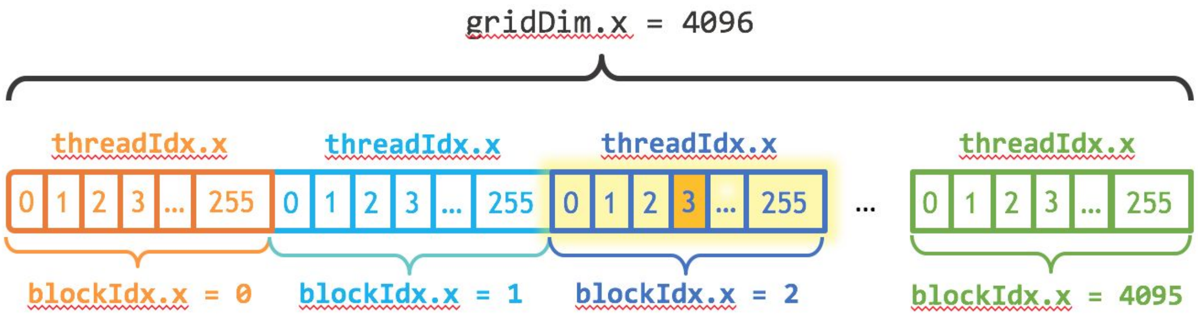
GPU

Kernel

Grid

Block - shared memory

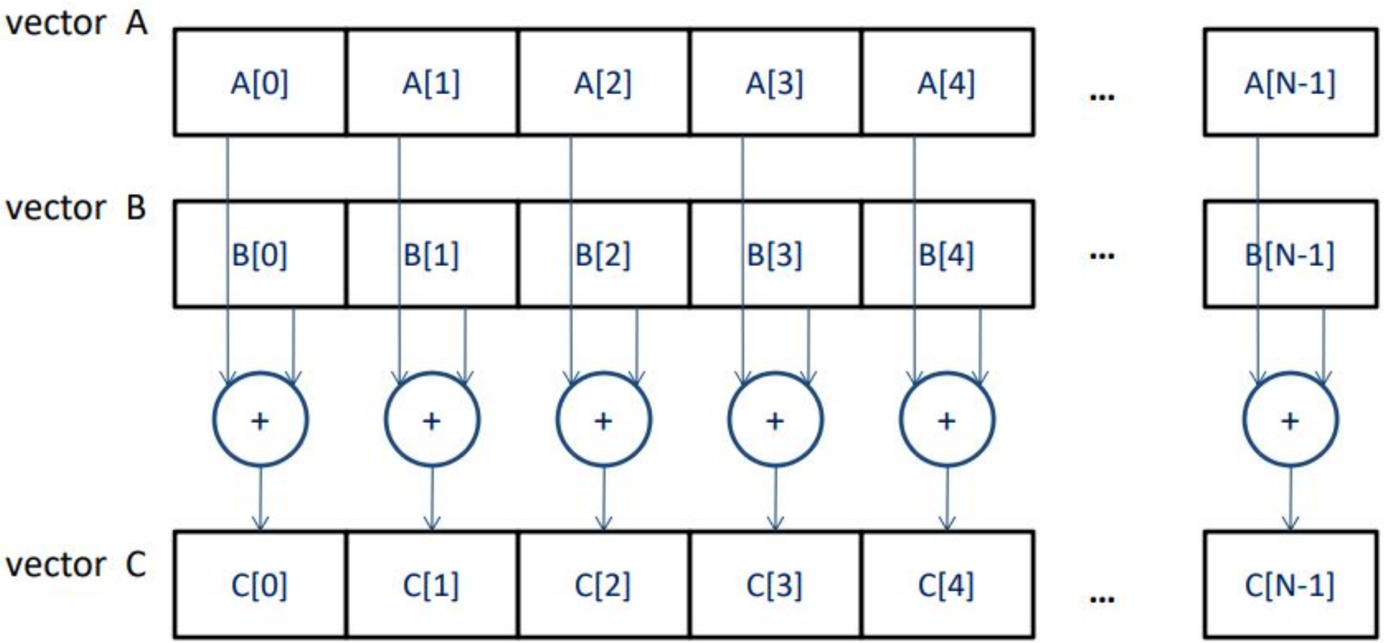
Thread



index = blockIdx.x * blockDim.x + threadIdx.x

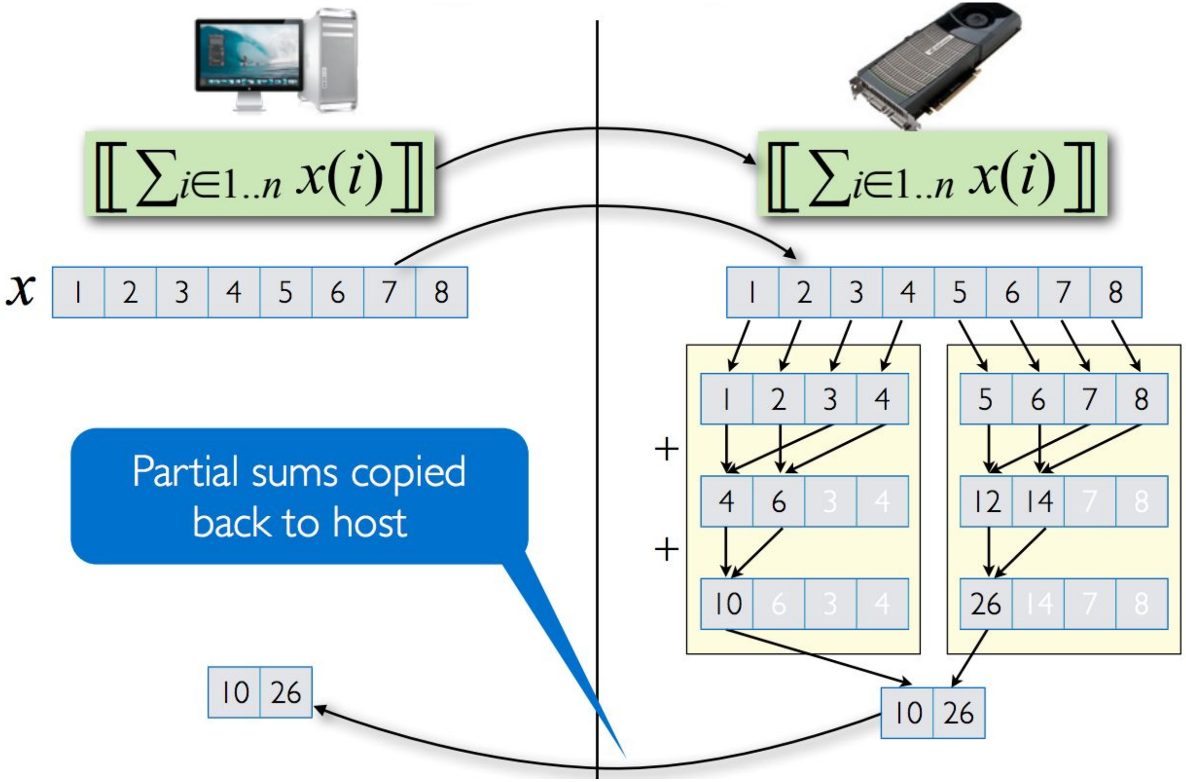
index = (2) * (256) + (3) = 515

GPU



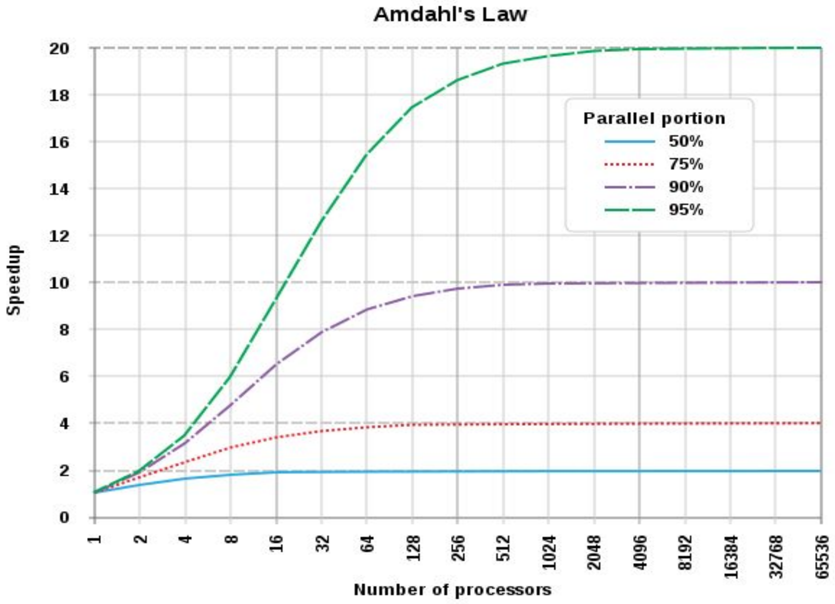
vector A
A[0]
A[1]
A[2]
A[3]
A[4]
...
A[N-1]
vector B
B[0]
B[1]
B[2]
B[3]
B[4]
...
B[N-1]
+
+
+
+
+
+
vector C
C[0]
C[1]
C[2]
C[3]
C[4]
...
C[N-1]

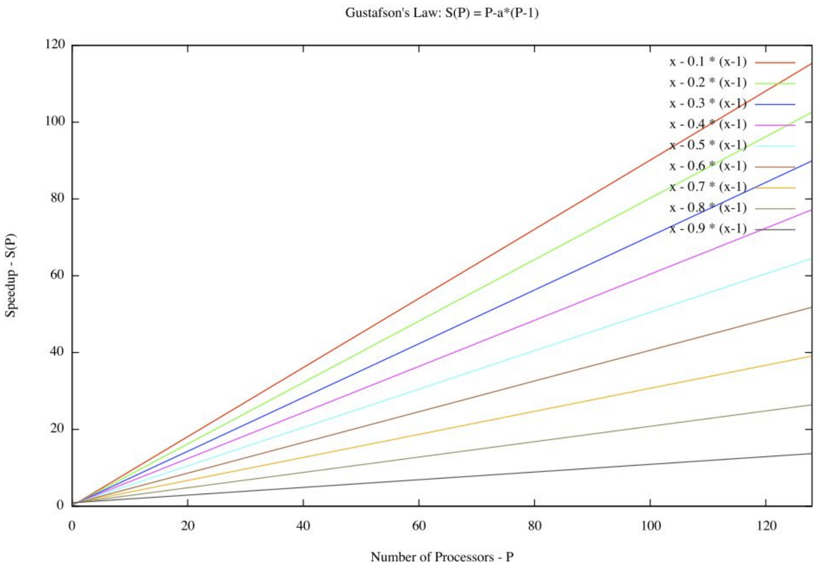
GPU



$[\![\sum_{i \in 1..n} x(i)]\!]$
$[\![\sum_{i \in 1..n} x(i)]\!]$
x
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8
1 2 3 4
5 6 7 8
+
4 6 3 4
12 14 7 8
+
10 6 3 4
26 14 7 8
Partial sums copied back to host
10 26
10 26

Limits to parallel processing





RAM vs disk vs network

Accessing the RAM is in the order of nanoseconds (10e-9 seconds), while accessing data on the disk or the network is in the order of milliseconds (10e-3 seconds).

If reading from RAM took one minute, then reading from disk or network would take 60 days.

Covered in next lectures

Directed acyclic graph (DAG)

Actors

Managing distributed state

THANK YOU!

CQRS

Command Query Responsibility Segregation