H2O Design and Infrastructure

R Summit and Workshop, Copenhagen

27 Jun 2015

Matt Dowle

Overview

1. What exactly is H2O

2. How it works

I'll start an 8 node cluster live on EC2 now

H_2O Play Home

Sign In

Sign into H_2O Play

Email

mattd@h2o.ai

Password

•••••••••••

Remember me

Sign In

Sign up for an account

H_2O Play Home

H_2O Play

My Clouds

Name	Node Count	Version	Status
matt-4	4	Shannon (3.0.0.22)	Deleted
MattDowle2	4	Shannon (3.0.0.22)	Deleted
Matt3	4	Shannon (3.0.0.22)	Deleted
Matt4	8	Shannon (3.0.0.22)	Deleted

Create a cloud...

Click

4 mins to start up, 2 slides

H2O

Machine learning e.g. Deep Learning

In-memory, parallel and distributed

1. Data **>** 240GB needle-in-haystack; e.g. fraud

2. Data **<** 240GB compute intensive, parallel 100's cores

3. Data **<** 240GB where feature engineering > 240GB

Speed for i) production and ii) interaction

Developed in the open on GitHub

Liberal Apache license

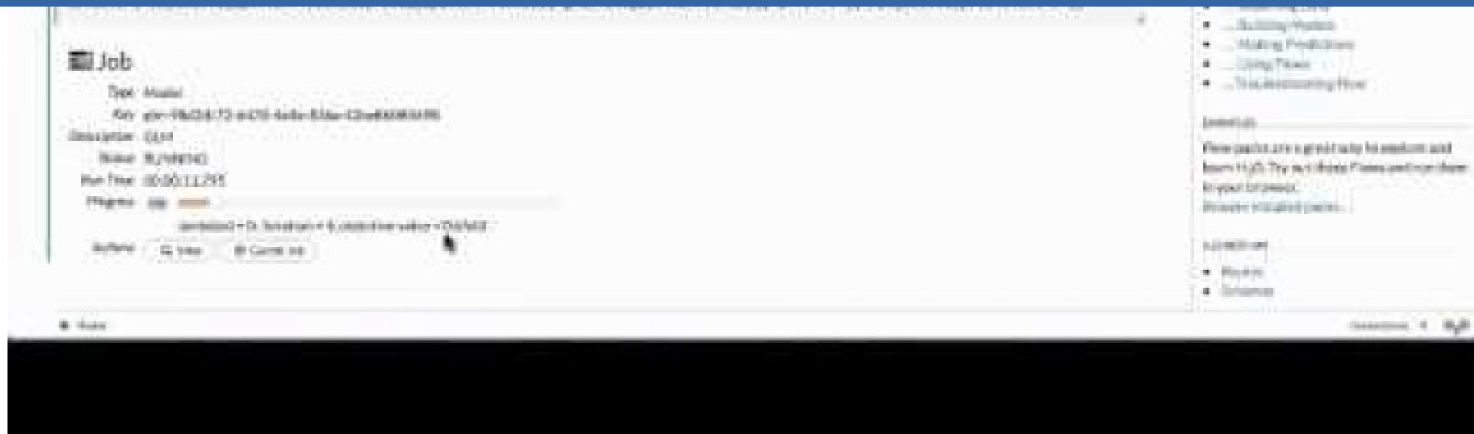
Use from R, Python or H2O Flow ... simultaneously

H_2O.ai
Machine Intelligence

I now work here

8-node cluster on EC2 is now ready

LIVE 15MIN DEMO



To use from R

```
# If java is not already installed :
$ sudo add-apt-repository -y ppa:webupd8team/java
$ sudo apt-get update
$ sudo apt-get -y install oracle-java8-installer
$ sudo apt-get -y install oracle-java8-set-default
$ java -version
```

```
$ R
> install.packages(“h2o”)
```

That's it.

Start H2O

```
> library(h2o)
> h2o.init()
H2O is not running yet, starting it now...
Successfully connected to http://127.0.0.1:54321
R is connected to H2O cluster:
    H2O cluster uptime:          1 sec 397 ms
    H2O cluster version:         2.8.4.4
    H2O cluster total nodes:     1
    H2O cluster total memory:    26.67 GB
    H2O cluster total cores:     32
```

h2o.importFile

23GB .csv, 9 columns, 500e6 rows

```
> DF <- h2o.importFile("/dev/shm/test.csv")
   user  system elapsed
  0.775   0.058  50.559
> head(DF)
    id1   id2            id3 id4 id5  id6 v1 v2      v3
1 id076 id035 id0000003459  20  80 8969  4  3 43.1525
2 id062 id023 id0000002848  99  49 7520  5  2 86.9519
3 id001 id052 id0000007074  89  16 8183  1  3 19.6696
```

H2O.ai
Machine Intelligence

```
library(h2o)
h2o.importFile("/dev/shm/test.csv") # 50 seconds
```

Parallel

```
library(data.table)
fread("/dev/shm/test.csv")          # 5 minutes
```

Single thread

```
library(readr)
read_csv("/dev/shm/test.csv")       # 12 minutes
```

Single thread

23GB .csv, **9** columns, **500e6** rows

H2O.ai
Machine Intelligence

h2o.importFile also

- compresses the data in RAM
- profiles the data while reading; e.g. stores min and max per column, for later efficiency gains
- included in 50 seconds
- accepts a directory of multiple files

Standard R

```
hex <- h2o.importFile(conn, path)

summary(hex)

hex$Year <- as.factor(hex$Year)

myY <- "IsDepDelayed"

myX <- c("Origin", "Dest", "Year", "UniqueCarrier",
"DayOfWeek", "Month", "Distance", "FlightNum")

dl <- h2o.deeplearning(y = myY, x = myX,
training_frame = hex, hidden=c(20,20,20,20),
epochs = 1, variable_importances = T)
```

How it works

Slides by Cliff Click

CTO and Co-Founder

Simple Data-Parallel Coding

- Map/Reduce Per-Row: **Stateless**
  - Example from Linear Regression, $\Sigma\ y^2$

```
double sumY2 = new MRTask() {
  double map( double d ) { return d*d; }
  double reduce( double d1, double d2 ) {
    return d1+d2;
  }
}.doAll( data );
```

- Auto-parallel, auto-distributed
- Fortran speed, Java Ease

Simple Data-Parallel Coding

- Map/Reduce Per-Row: **Statefull**
  - Linear Regression Pass1: Σx, Σy, Σy^2

```
class LRPass1 extends MRTask {
  double sumX, sumY, sumY2;// I can have State?
  void map( double X, double Y ) {
    sumX += X;  sumY += Y;  sumY2 += Y*Y;
  }
  void reduce( LRPass1 that ) {
    sumX  += that.sumX ;
    sumY  += that.sumY ;
    sumY2 += that.sumY2;
  }
}
```

Non-blocking distributed KV

- Uniques
  - Uses distribu

Setting dnbhs in <init> makes it an **input** field. Shared across all maps(). Often read-only. This one is written, so needs a **reduce**.

```
class Uniques extends MRTask {
  DNonBlockingHashSet<Long> dnbhs = new ...;
  void map( long id ) { dnbhs.add(id); }
  void reduce( Uniques that ) {
    dnbhs.putAll(that.dnbhs);
  }
};
long uniques = new Uniques().
  doAll( vecVistors ).dnbhs.size();
```

Limitations

- Code runs distributed...
  - No I/O or Machine Resource allocation
  - No new threads, no locks, no System.exit()
- No global / static variables
  - Instead they become node-local
  - "Small" global read state: in constructor
  - "Small" global writable state: use **`reduce()`**
  - "Big" state: read/write distributed arrays (Vecs)
- Runs one (big step) to completion, then another...

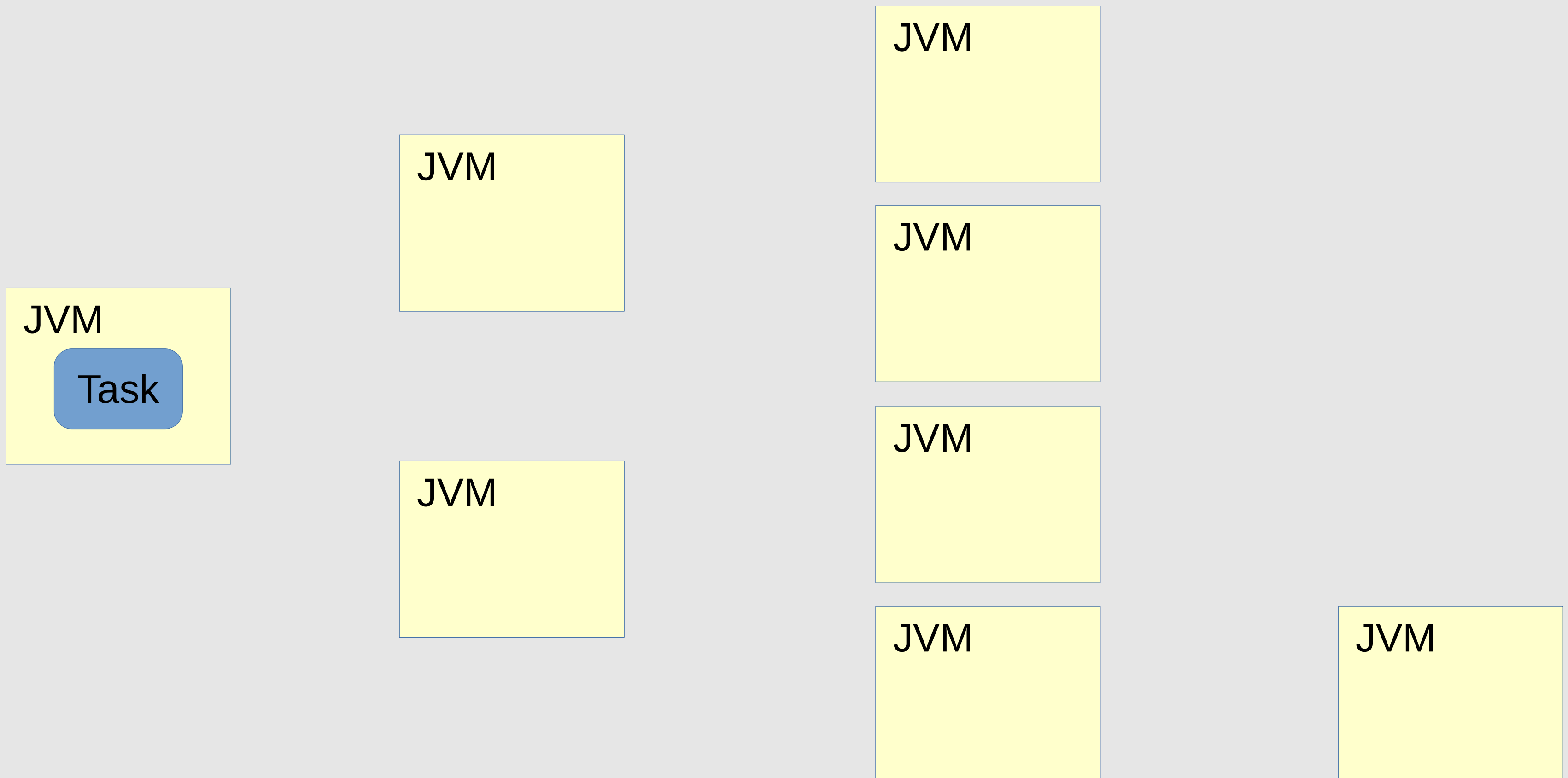
Strengths

- Code runs distributed & parallel without effort
  - Millions & billions of rows; 1000's of cores
- Single-threaded coding style
  - No concurrency issues
- Excellent resource management
  - "No knobs needed" for GC or CPUs or network
  - No "data placement", no "hot blocks" or "hot locks"

How Does It Work?
(Code)

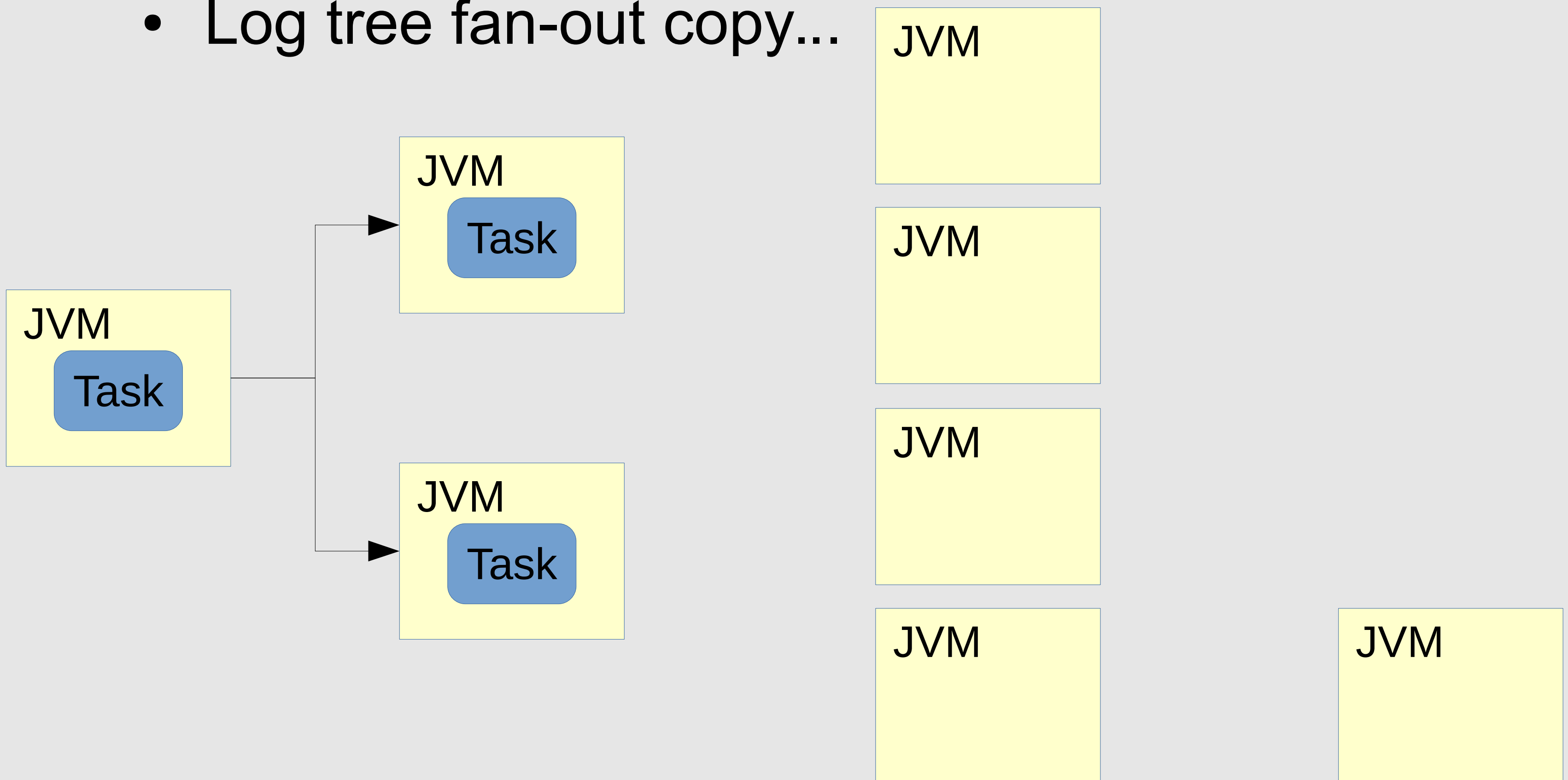
Distributed Fork / Join

- `T = new MRtask().doAll(data);`



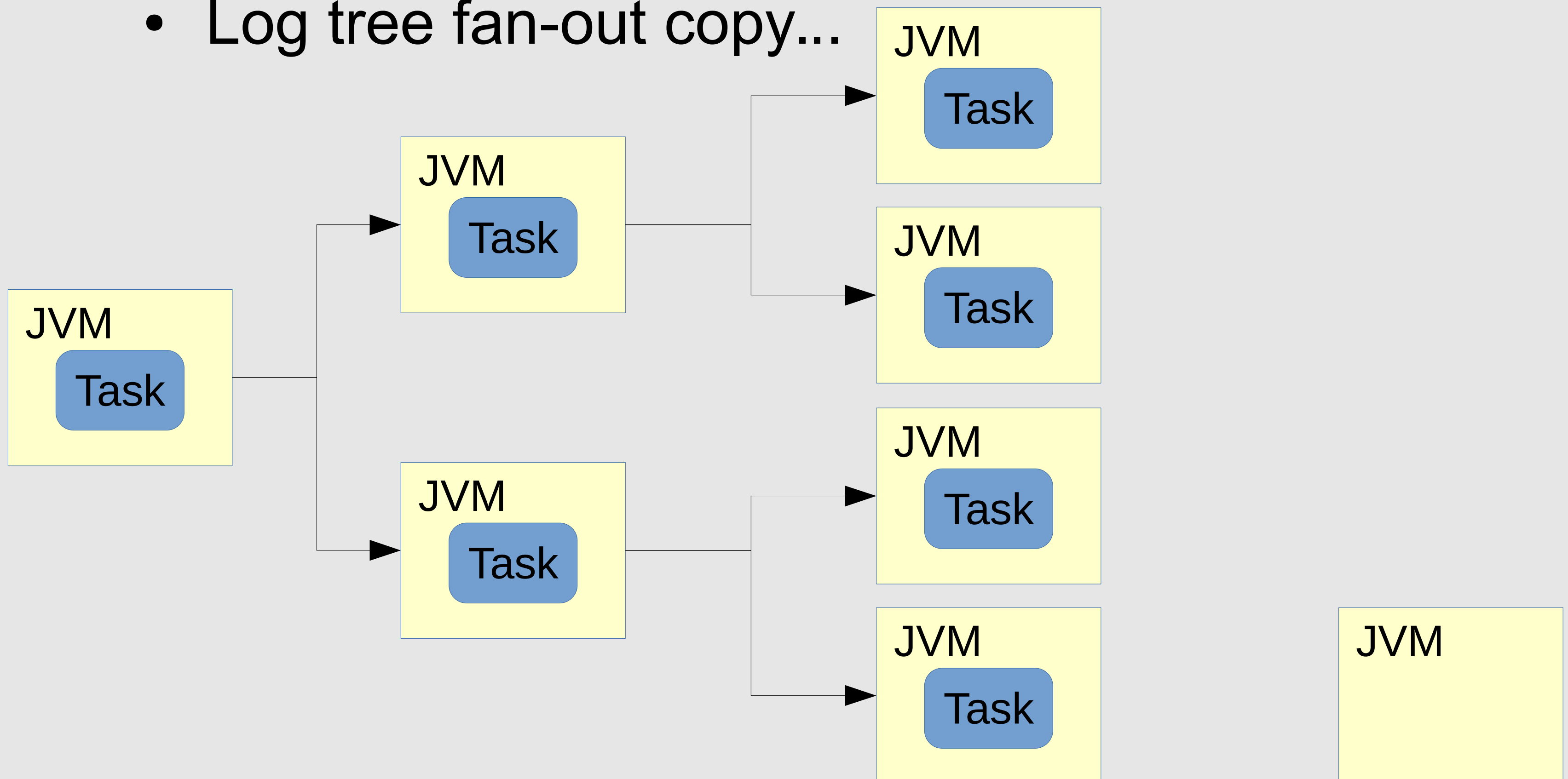
Distributed Fork / Join

- `T = new MRtask().doAll(data);`
  - Log tree fan-out copy...



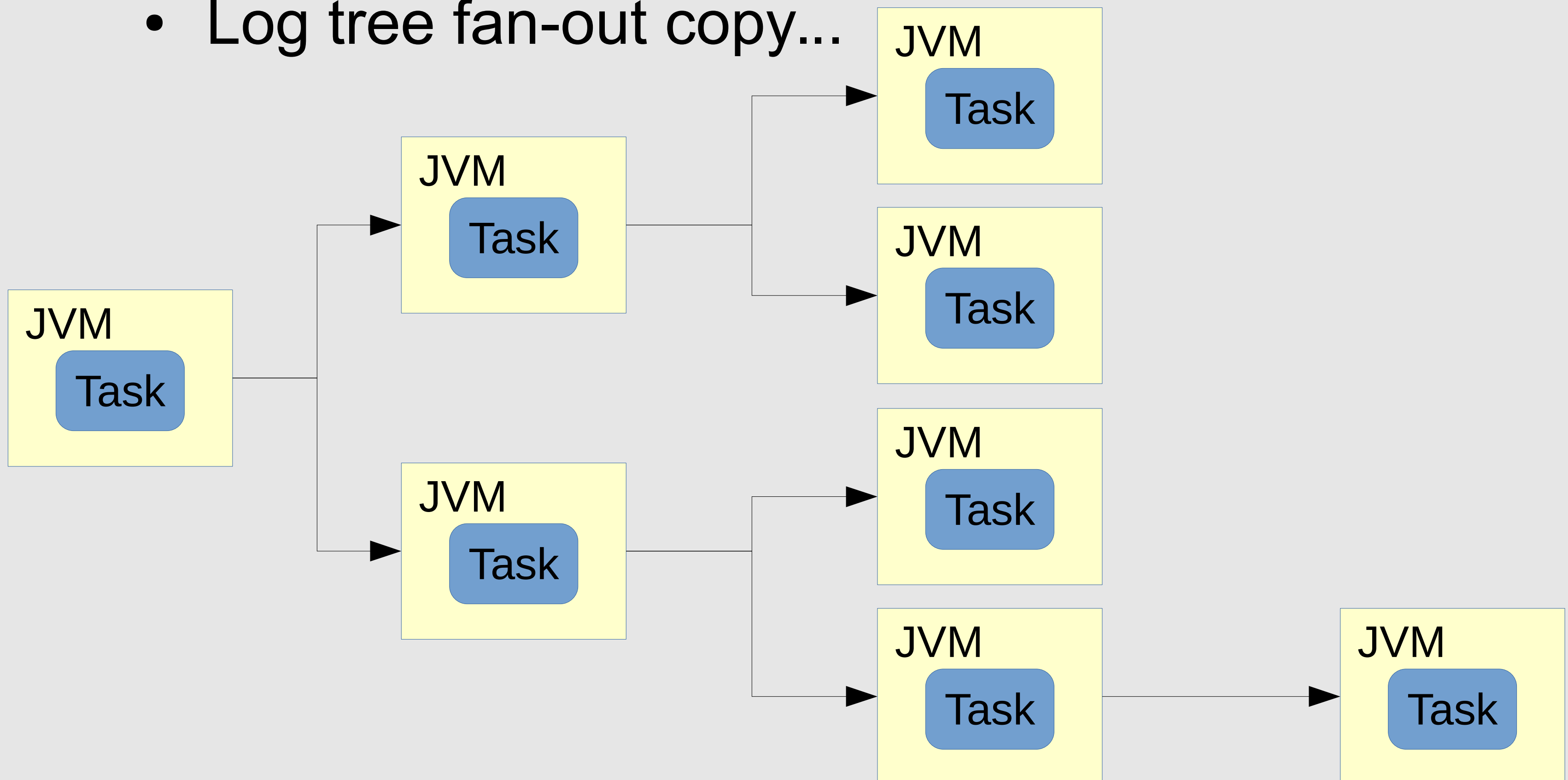
Distributed Fork / Join

- **`T = new MRtask().doAll(data);`**
  - Log tree fan-out copy...



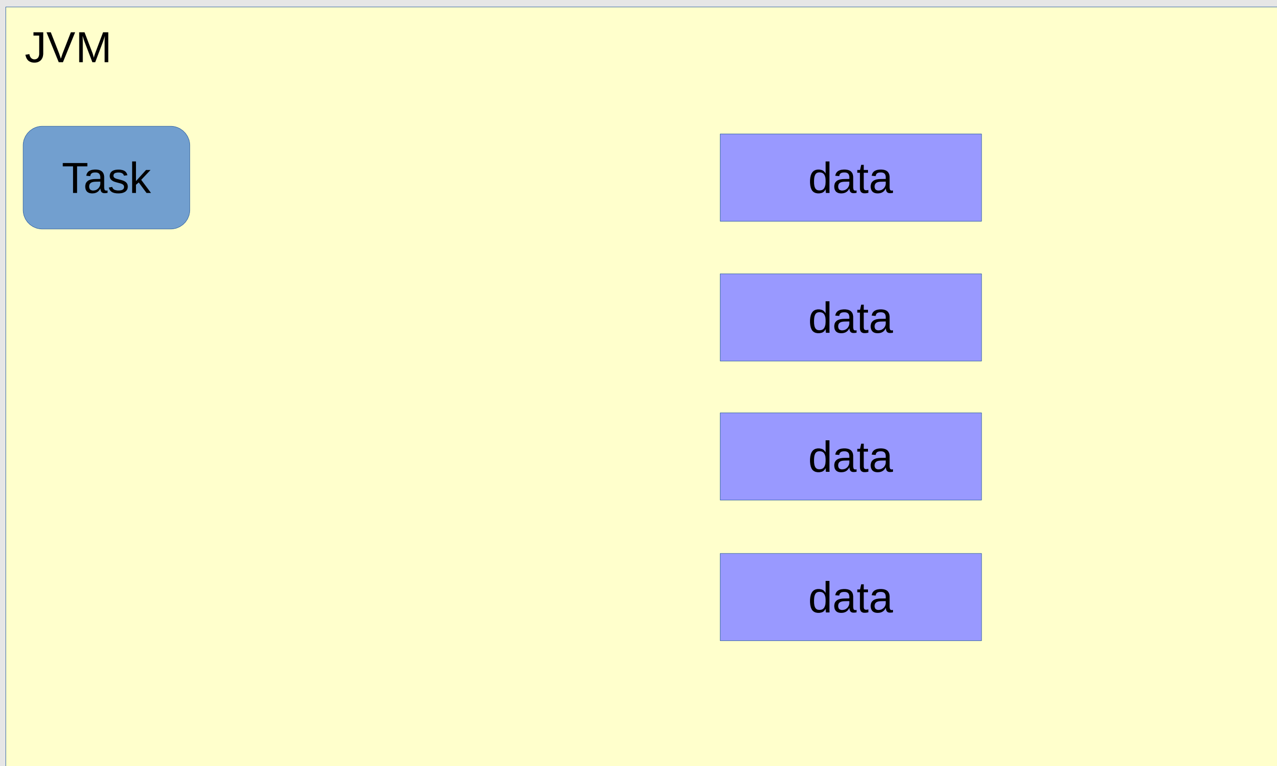
Distributed Fork / Join

- `T = new MRtask().doAll(data);`
  - Log tree fan-out copy...



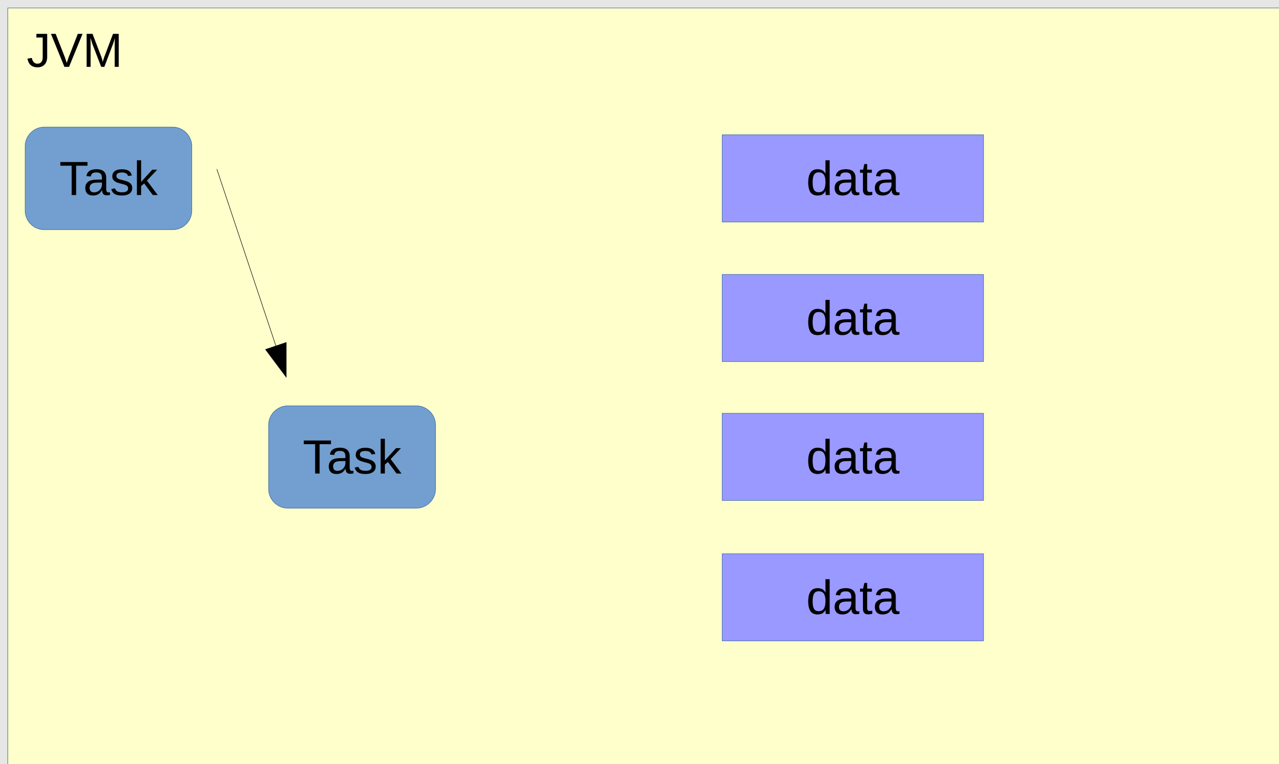
Distributed Fork / Join

- Within a single node, classic Fork/Join
  - Divide & Conquer



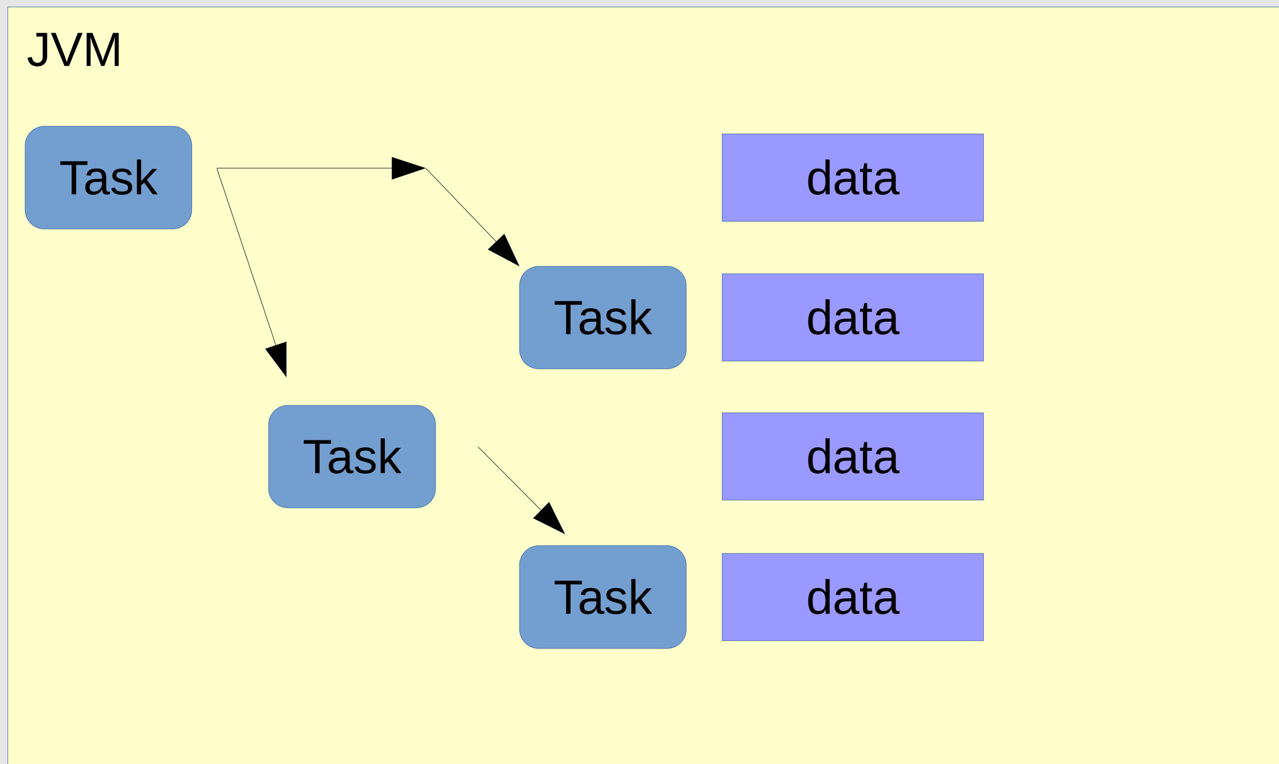
Distributed Fork / Join

- Within a single node, classic Fork/Join
  - Divide & Conquer



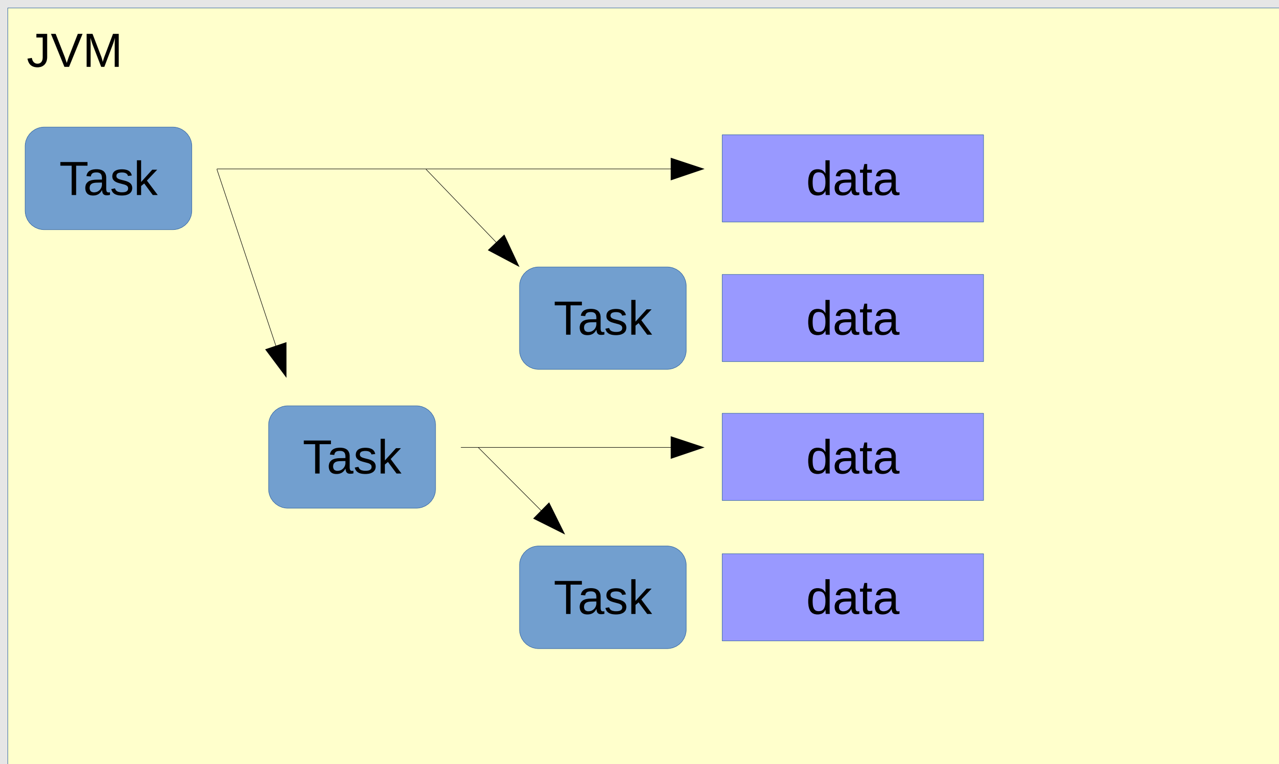
Distributed Fork / Join

- Within a single node, classic Fork/Join
    - Divide & Conquer



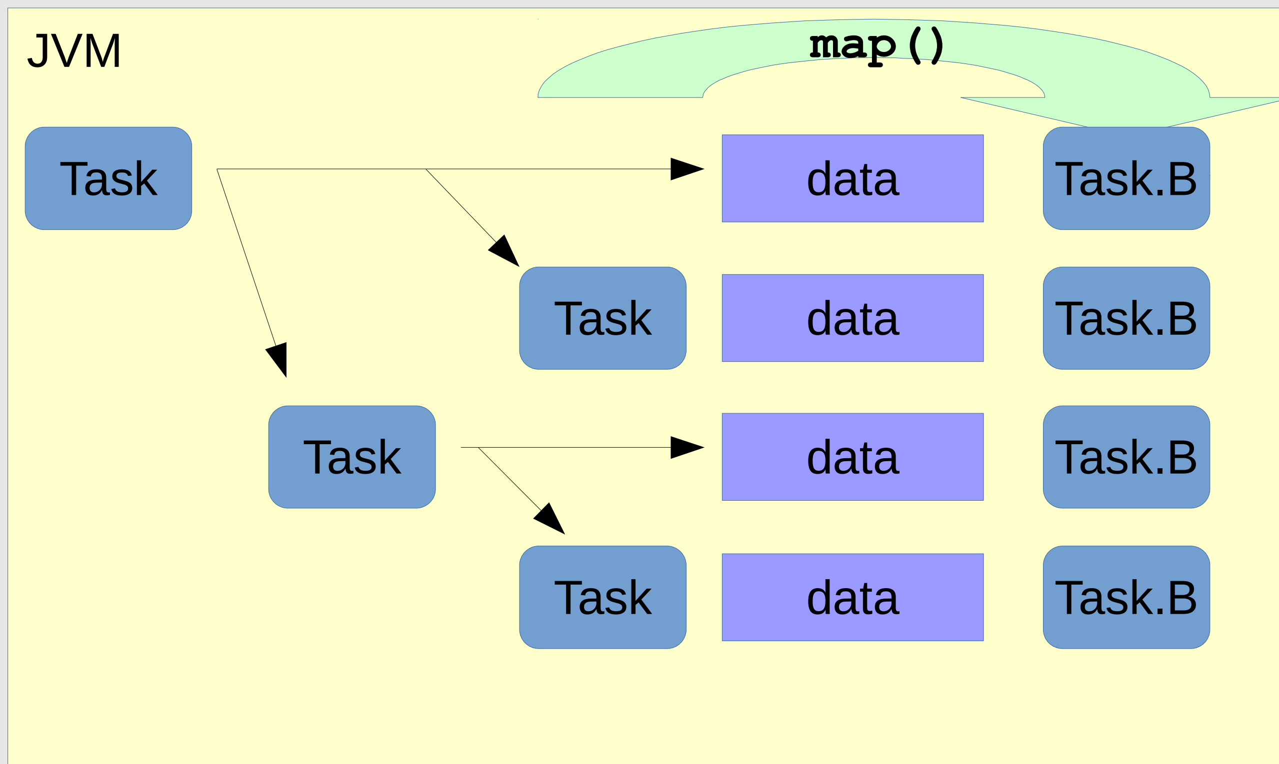
Distributed Fork / Join

- Within a single node, classic Fork/Join
  - Divide & Conquer



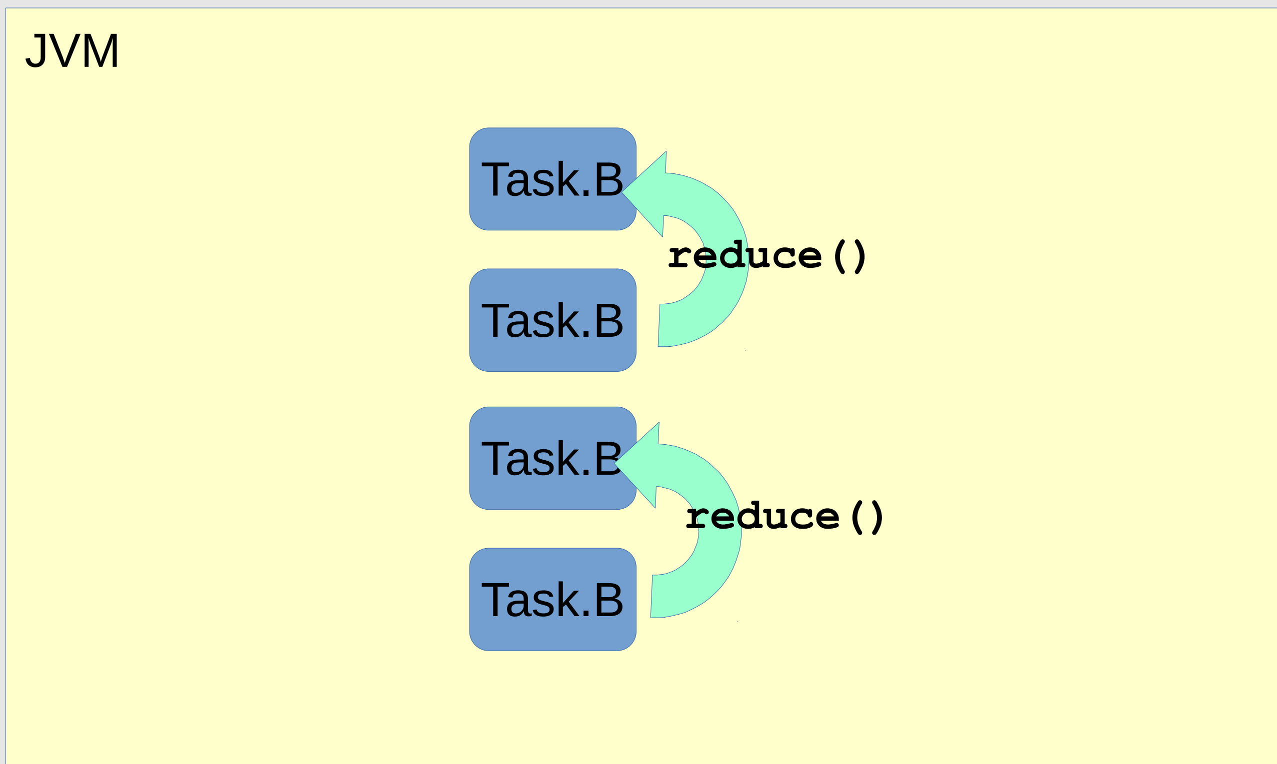
Distributed Fork / Join

- Within a single node, classic Fork/Join
  - Divide & Conquer, parallel map over local data



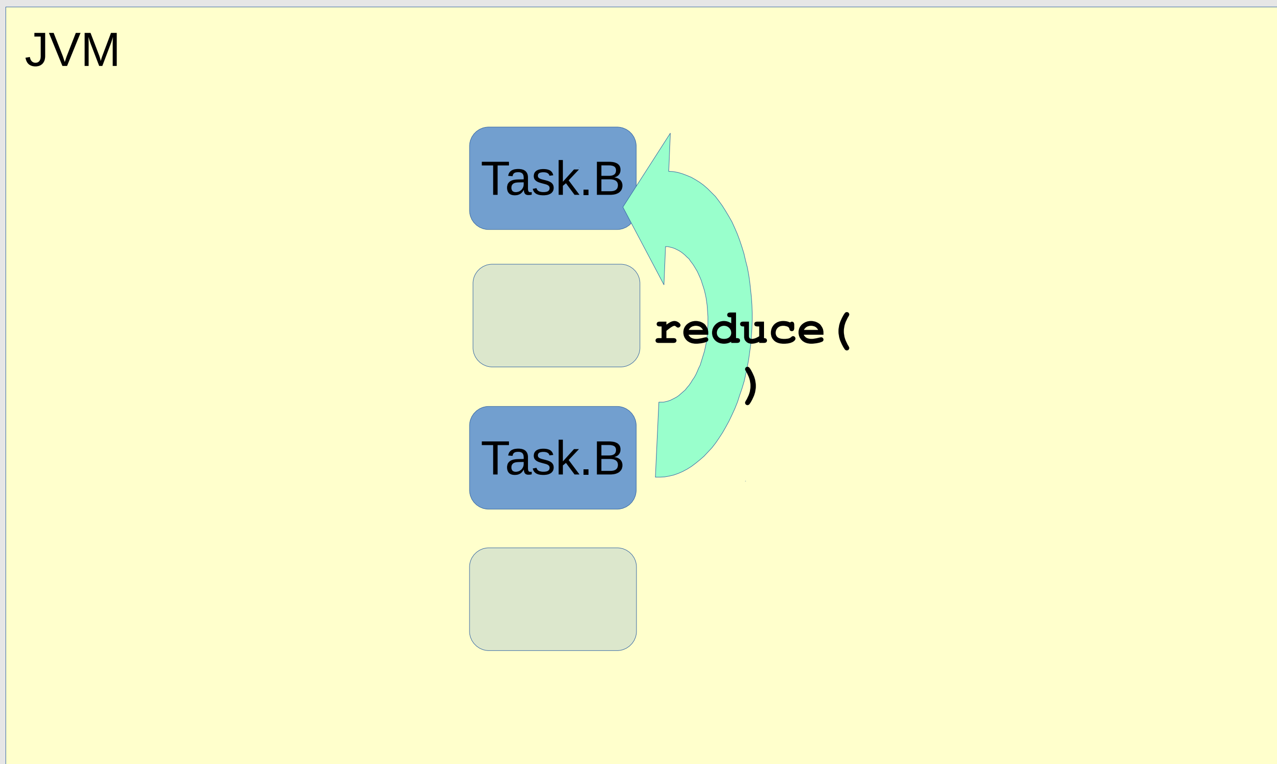
Distributed Fork / Join

- Within a single node, classic Fork/Join
  - Parallel, eager reduces



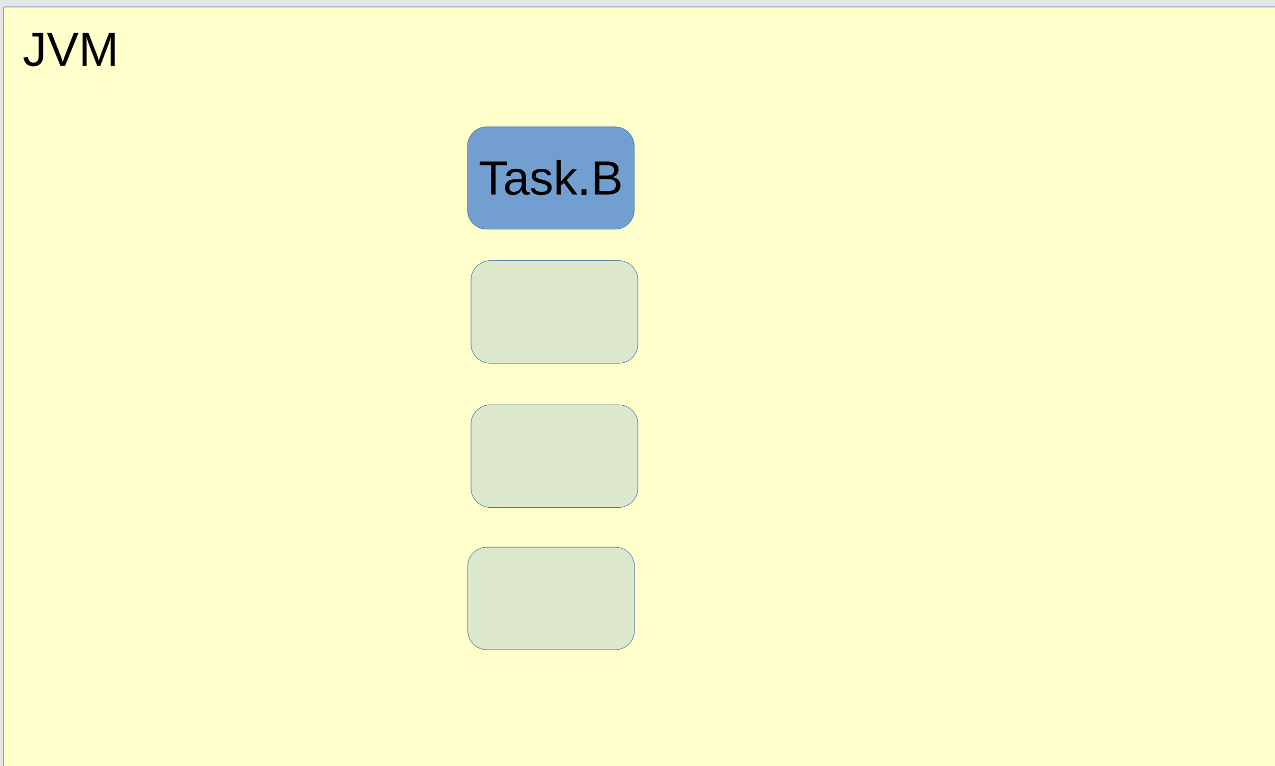
Distributed Fork / Join

- Within a single node, classic Fork/Join
  - Log-tree reduce



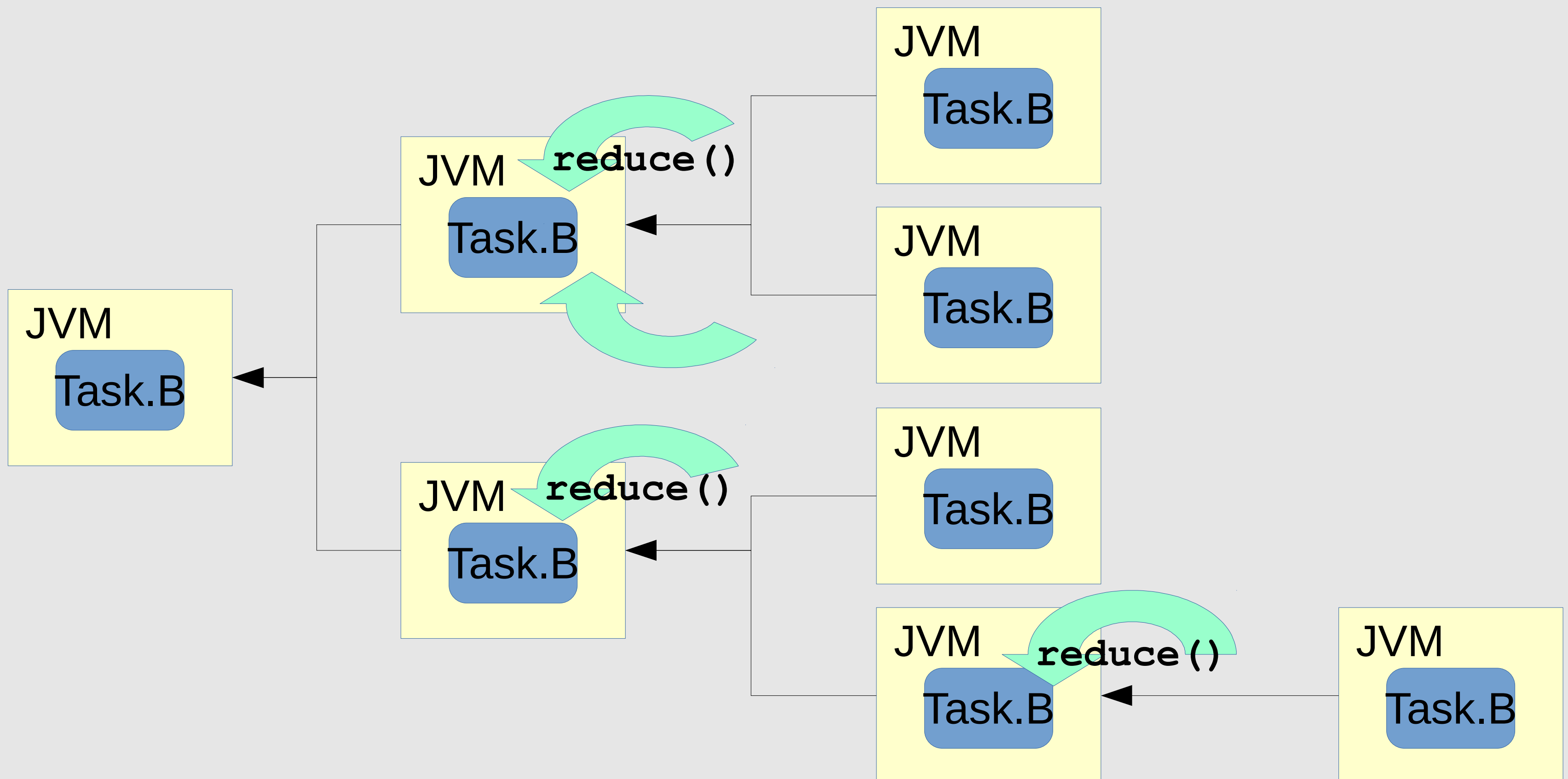
Distributed Fork / Join

- Within a single node, classic Fork/Join
  - Reduce to the same top-level instance



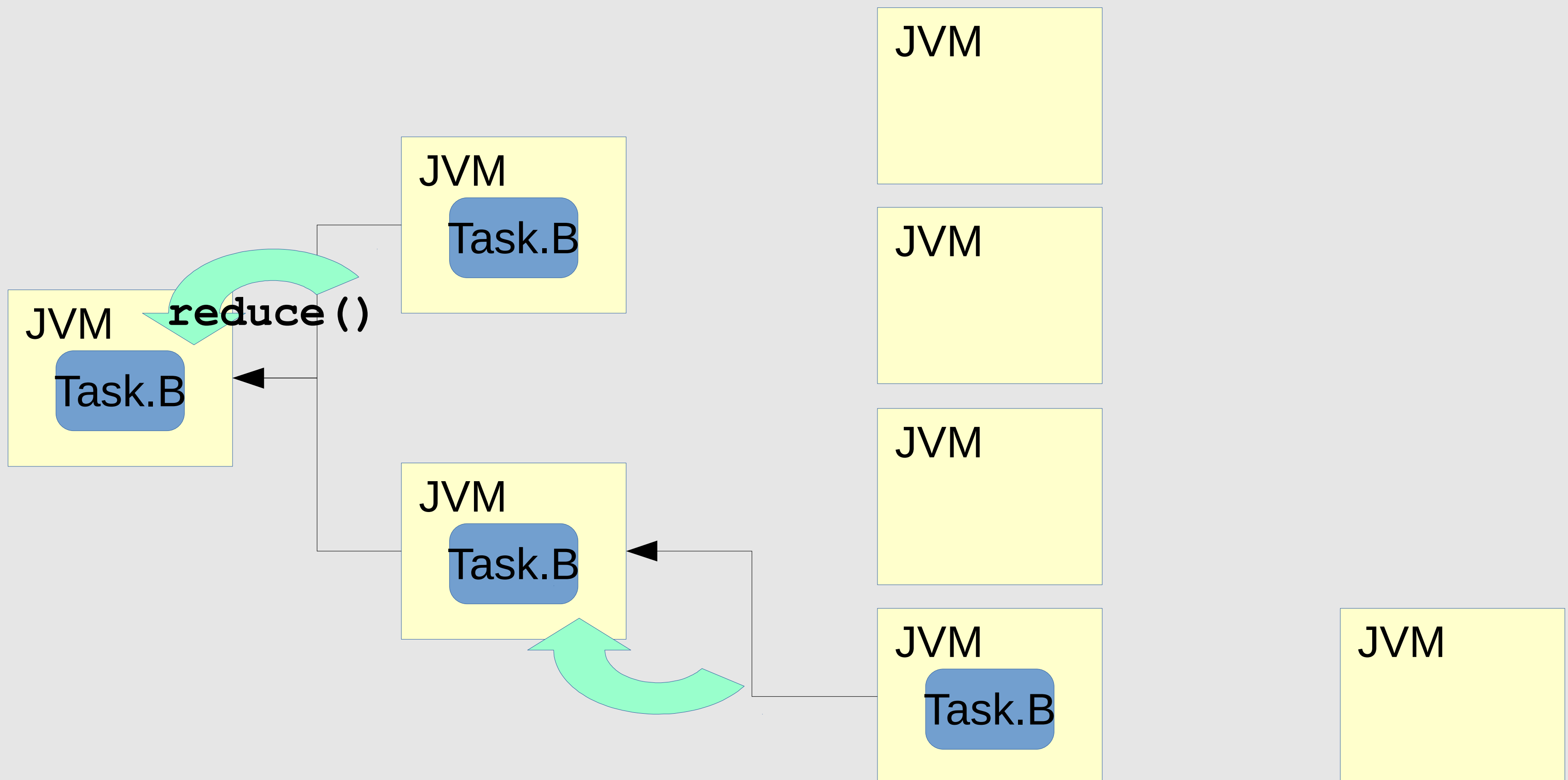
Distributed Fork / Join

- Reductions back up the log-tree



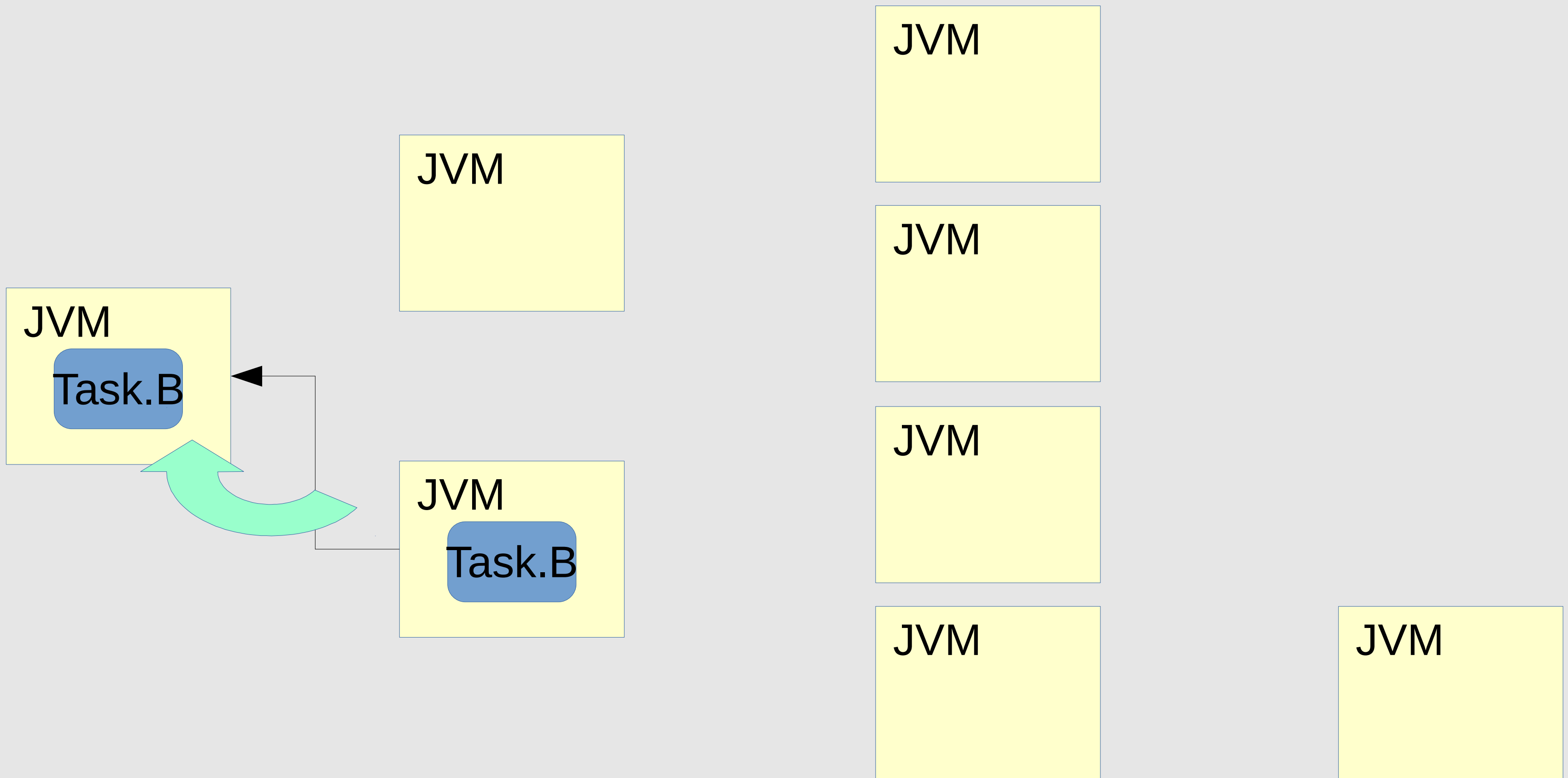
Distributed Fork / Join

- Reductions back up the log-tree



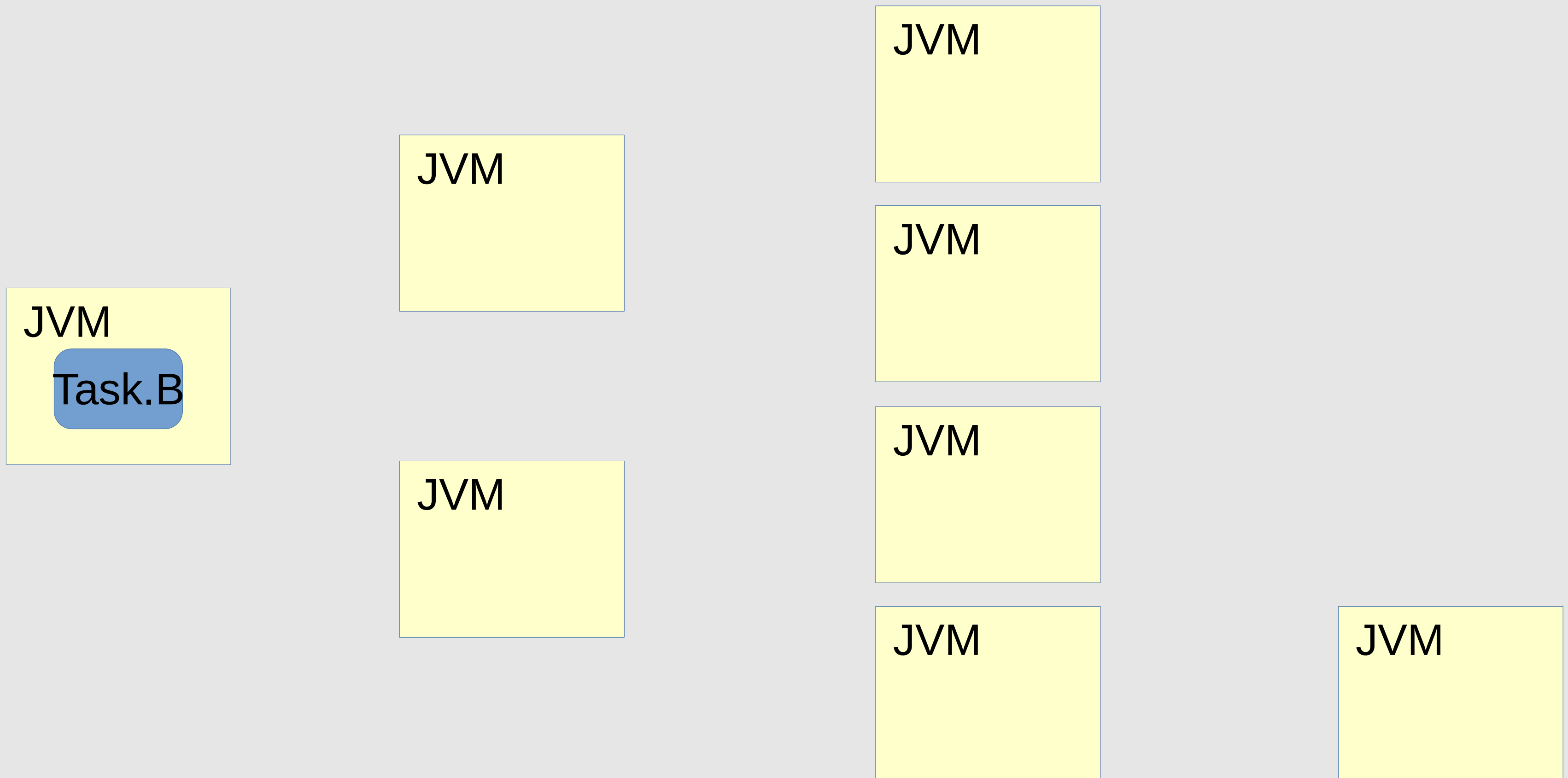
Distributed Fork / Join

- Final reduction into the original instance



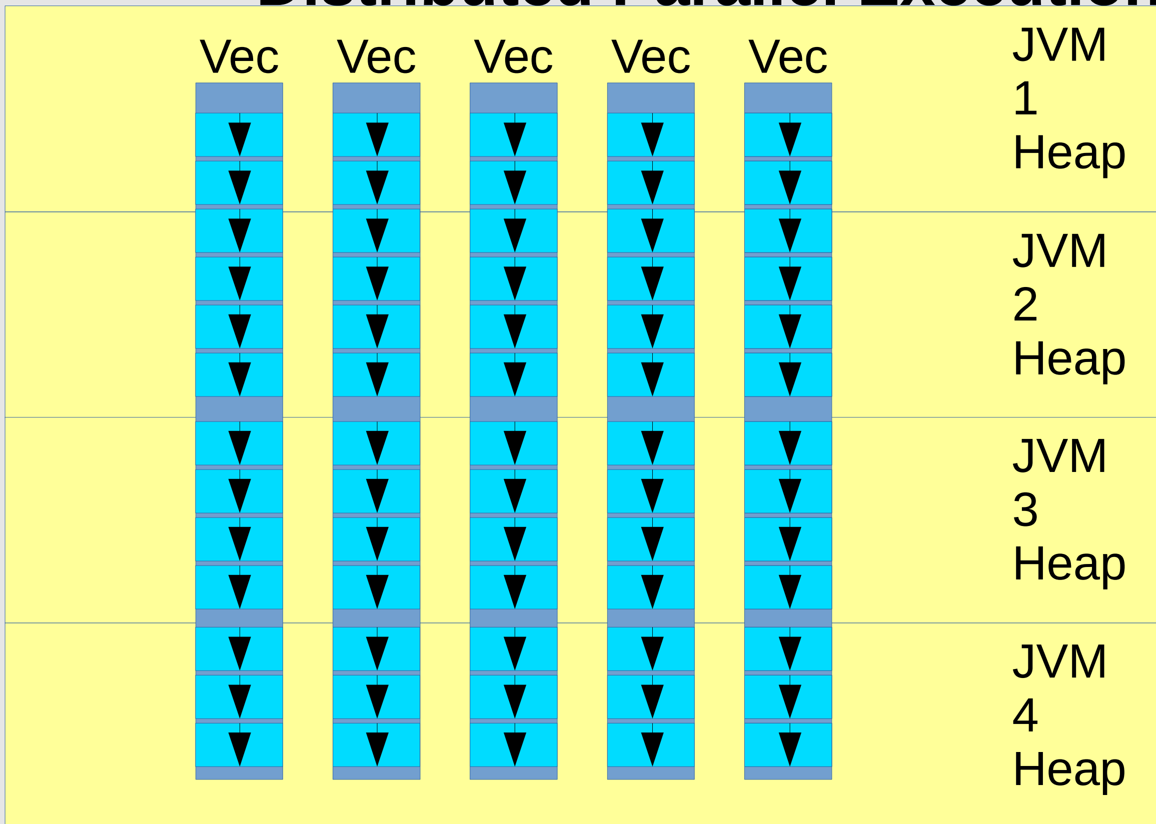
Distributed Fork / Join

- Final reduction into the original instance



How Does It Work?
(Data)

Distributed Data Taxonomy



- All CPUs grab Chunks in paralle
- F/J load balances

- Code moves to Da
- Map/Reduce & F/J handles all sync
- H2O handles all comm, data man

Distributed Data Taxonomy

```
Frame – a collection of Vecs
  Vec – a collection of Chunks
    Chunk – a collection of 1e3 to 1e6 elems
      elem – a java double
```

```
Row i – i'th elements of all the Vecs in a
Frame
```

Summary

Distributed Coding Taxonomy

- No Distribution Coding:
  - Whole Algorithms, Whole Vector-Math
  - REST + JSON: e.g. load data, GLM, get results
  - R, Python, Web, bash/curl
- Simple Data-Parallel Coding:
  - Map/Reduce-style: e.g. Any dense linear algebra
  - Java/Scala foreach* style
- Complex Data-Parallel Coding
  - K/V Store, Graph Algo's, e.g. PageRank

Summary: Writing (distributed) Java

- Most simple Java “just works”
  - Scala API is experimental, but will also "just work"
- **Fast**: parallel distributed reads, writes, appends
  - Reads same speed as plain Java array loads
  - Writes, appends: slightly slower (compression)
  - Typically memory bandwidth limited
    - (may be CPU limited in a few cases)
- **Slower**: conflicting writes (but follows strict JMM)
  - Also supports transactional updates

Summary: Writing Analytics

- We're writing Big Data Distributed Analytics
  - Deep Learning
  - Generalized Linear Modeling (ADMM, GLMNET)
    - Logistic Regression, Poisson, Gamma
  - Random Forest, GBM, KMeans, PCA, ...

- Come write your own (distributed) algorithm!!!

Further articles from H2O

Efficient Low Latency Java and GCs

A K/V Store For In-Memory Analytics: Part 1

A K/V Store For In-Memory Analytics, Part 2

H2O Architecture

http://h2o.ai/about/