# Spark and Hail: An Introduction

Robert M. Porsch

May 18, 2018

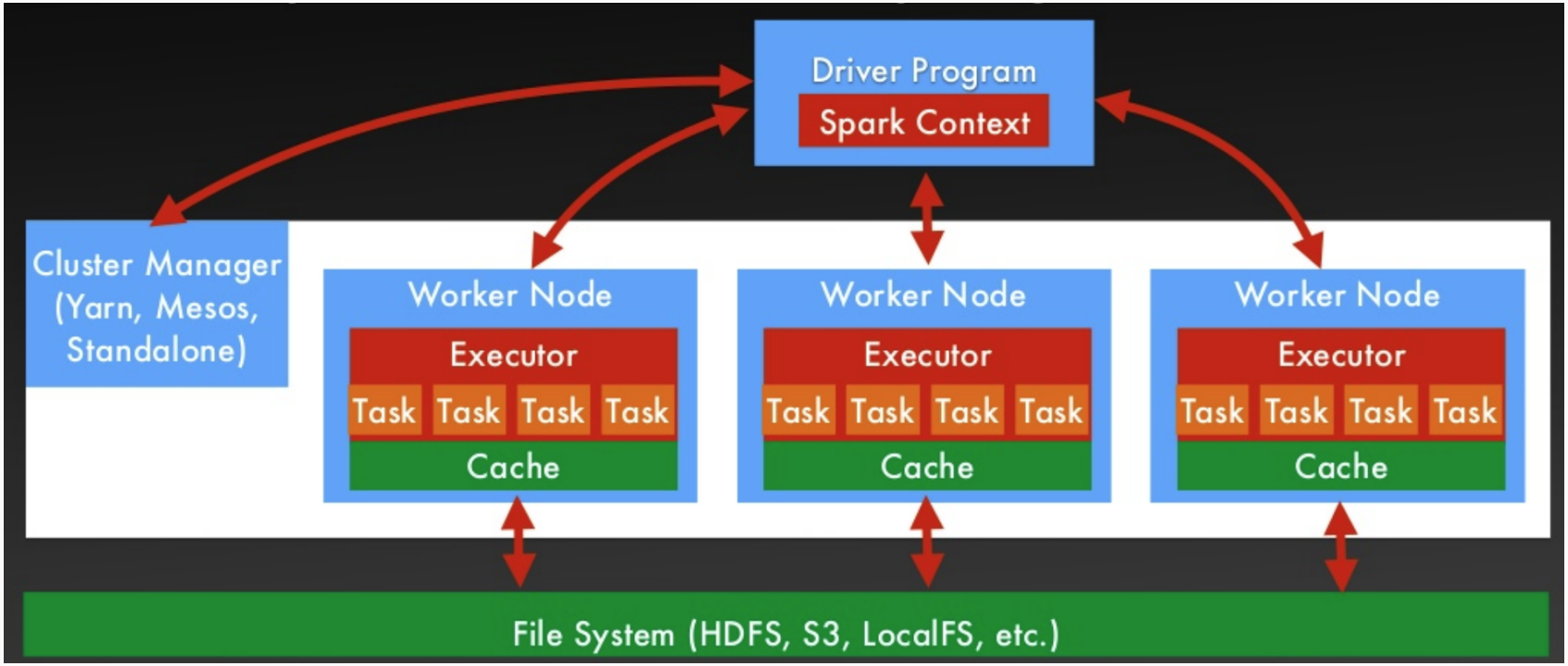
Center of Genomic Science

Main aim of this presentation

- What is Spark and why is it useful?
- Hail: Spark for genetic data analysis
- The cloud: Experiences using Hail after 2 weeks

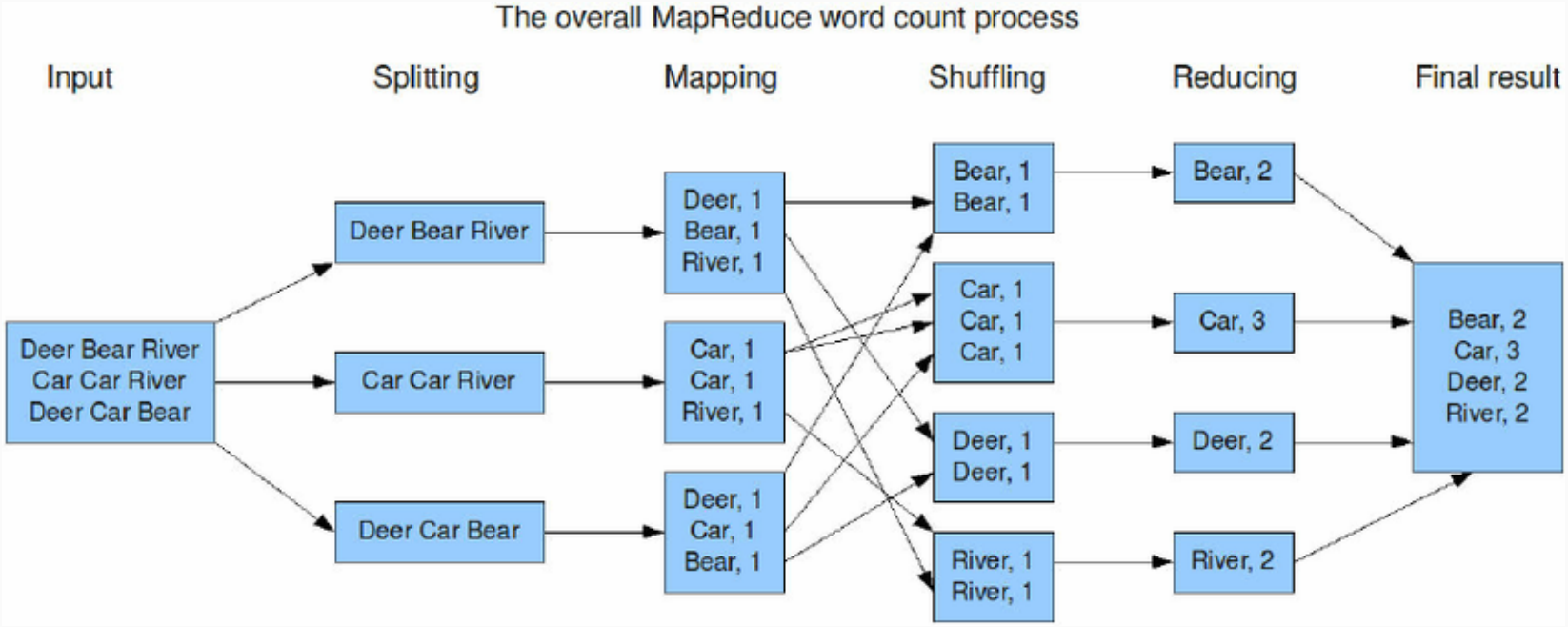
Bonus: Distributed statistical learning

# Basic Architecture of Spark




- Spark splits the input data-set into independent chunks
- Chunks are processed by the workers in a completely parallel manner.
- This is also called MapReduce

# Map Reduce



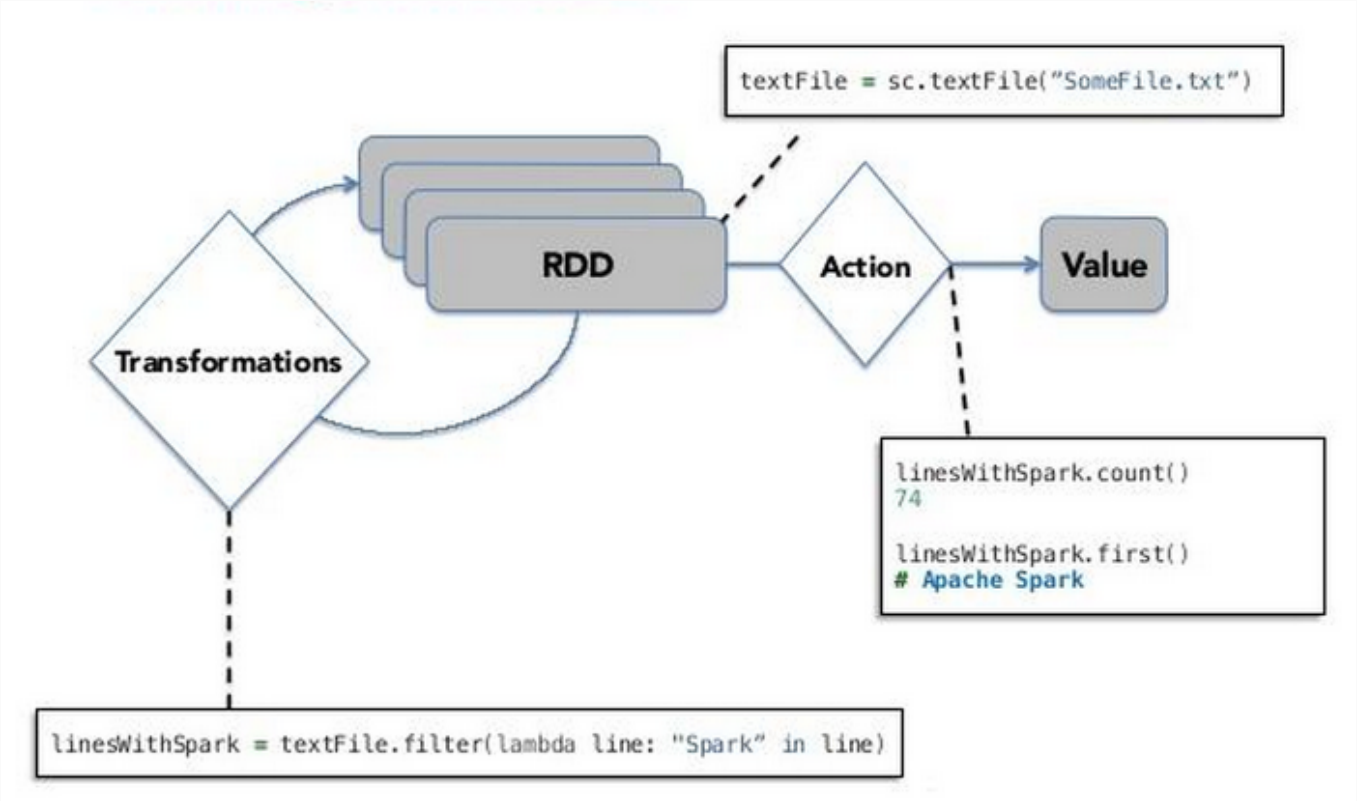

Spark retains the general idea of MapReduce from Hadoop, just not the implementation

# Spark and MapReduce

Aim of Spark: *generalize* MapReduce

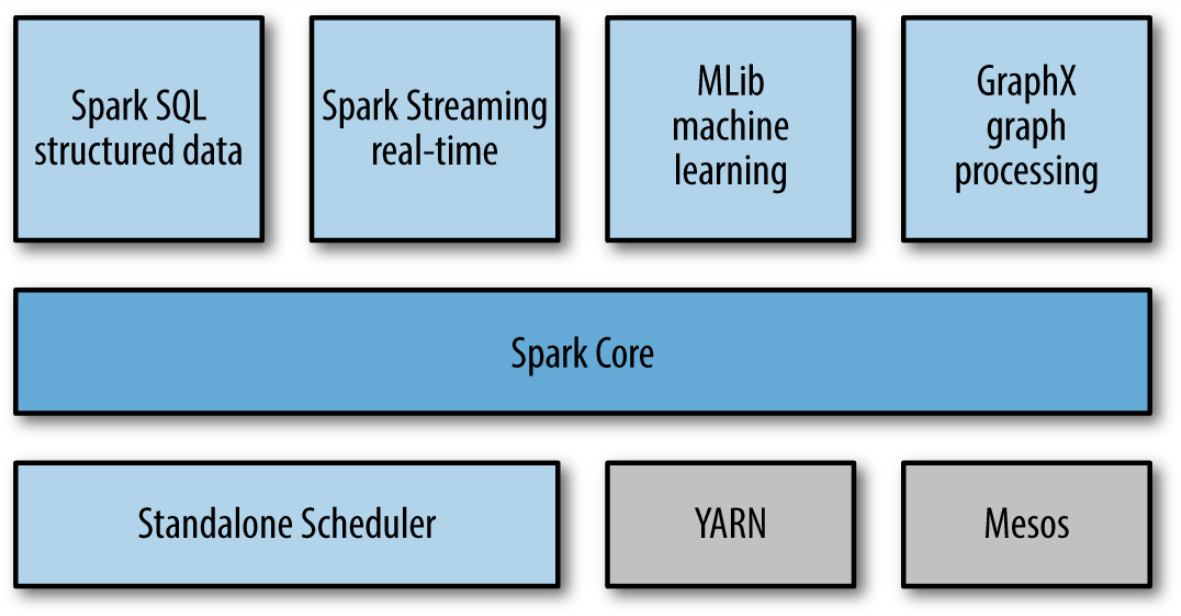
- handles batch, interactive and real-time data
- native integration with python, Java and Scala
- programming at a higher level of abstraction
- additional constructor (next to MapReduce)
- more memory efficient
- fault tolerance

# General Spark Framework




RDD: Resilient Distributed Dataset

# Spark Technology Stack



# Application in genetics

Distribution of data across multiple nodes for processing by:

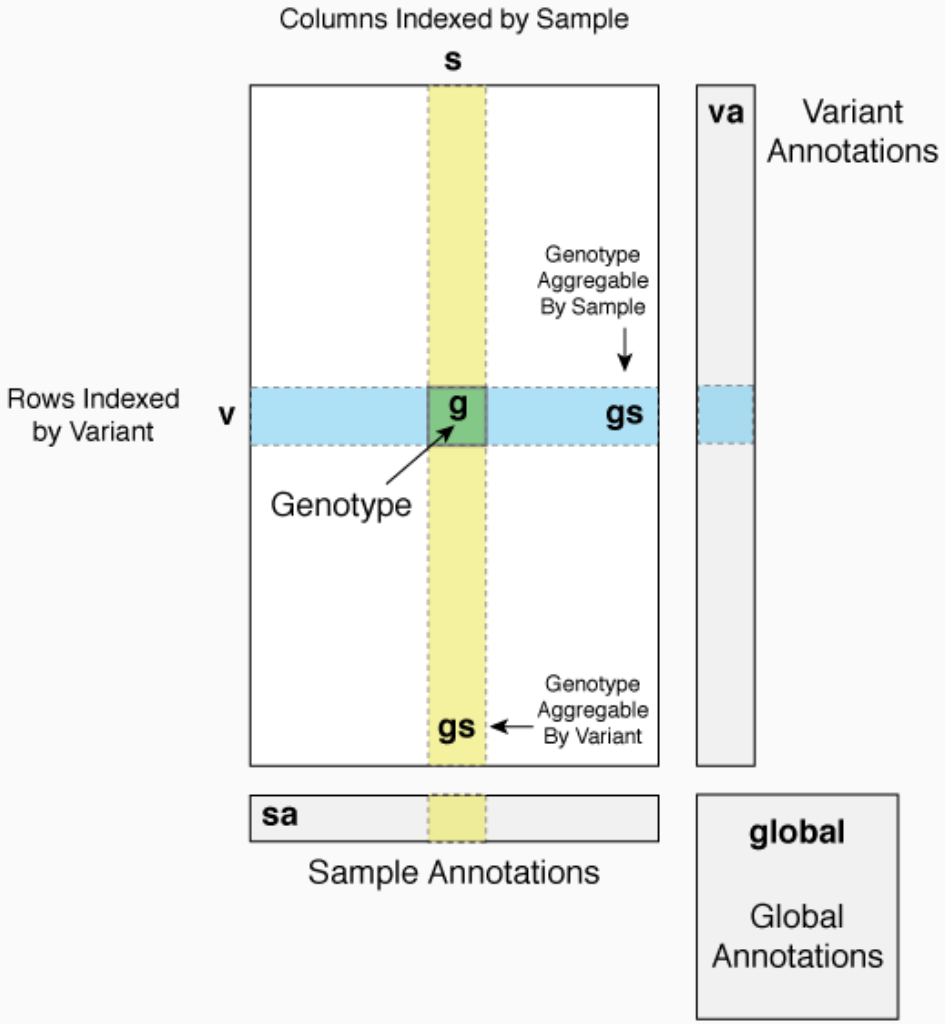
- Position
- Sample

The difference between Spark and our current approach is **scalability** and **automation**!

- Can scale to $X$ number of clusters automatically
- Data is dynamically chucked off and distributed
- Results are automatically feed back

# Hail: Spark for genetic data analysis

Hail comes from the Neal Lab (2015) with the aim to deliver distributed algorithm to the genetic community.



# How does it work?

- variants are distributed in packages across spark executors
  E.g. Variant $rs1, \ldots, rs1000$ goes to worker 1
- master organizes distribution of data packages
  (among current and new executors)
- executors perform requested computations and feed results back to master
- all of this is done in automation without user configuration

# Using Google's Cloud

**Why did we use Google?**

- we got 300USD credit for free
- Broad pushes new versions of hail onto Google
- Tools are available to rapidly start a spark cluster with hail (with one simple line of code)
- Variant annotations are available on Google and ready to be used

**What are some of the cluster configurations?**

- number of workers (flexible)
- number or preemptive workers (flexible)
- number of CPU, RAM, hard disk size for each worker and master machine (fixed)

# Start A Hail Cluster

```
gcloud dataproc clusters create performancetest1 \
        --image-version=1.2 \
        --master-machine-type=n1-standard-8 \
        --metadata="JAR=gs://hail-common/builds/devel/jars/hail-devel-b245f7d4af21-Spark-2.2.0.jar,
        ZIP=gs://hail-common/builds/devel/python/hail-devel-b245f7d4af21.zip,MINICONDA_VERSION=4.4.10" \
        --master-boot-disk-size=100GB \
        --num-master-local-ssds=0 \
        --num-preemptible-workers=0 \
        --num-worker-local-ssds=0 \
        --num-workers=8 \
        --preemptible-worker-boot-disk-size=40GB \
        --worker-boot-disk-size=40GB \
        --worker-machine-type=n1-standard-8 \
        --zone=asia-east1-a \
        --properties="spark:spark.driver.memory=24g,spark:spark.driver.maxResultSize=0,
        spark:spark.task.maxFailures=20,spark:spark.kryoserializer.buffer.max=1g,
        spark:spark.driver.extraJavaOptions=-Xss4M,spark:spark.executor.extraJavaOptions=-Xss4M,hdfs:dfs.replication=1" \
        --initialization-actions="gs://dataproc-initialization-actions/conda/bootstrap-conda.sh,
        gs://hail-common/cloudtools/init_notebook1.py"
```

- preemptive nodes: Nodes with 80% discount
- MINICONDA: Python packages for scientific computing
- We are using a pre-build Hail version from Broad
- Important: Region needs to be defined
- Usage of standard build scripts to install Python and other tools

# Example: GWAS in Hail

Hail provides a high level interface to perform various tasks.

```
3 import hail as hl
4
5 hl.init()
6 vds = hl.import_plink('gs://ukb_testdata/maf_0.01_10.bed',
7                       'gs://ukb_testdata/maf_0.01_10.bim',
8                       'gs://ukb_testdata/maf_0.01_10.fam')
9 gwas = hl.linear_regression(y=vds.phenotype, x=vds.GT.n_alt_alleles(),
10     covariates=[vds['pc'+str(i)] for i in range(1, 12)])
11 results = gwas.rows()
12 results.write('gs://ukb_testdata/GWAS_test_tim/GWAS_test_tim', overwrite=Tru
```

- perform QC
- compute relationship matrix
- Burden tests
- annotations
- and much more

# A case study

Lets run a simple GWAS:

- 300163 samples
- 369578 variant
- 8 workers, with each 8 CPU

Outcome:

- CPU usage: $\sim 83\%$
  Max. possible utilization:
  87.5%
- runtime: 1h 26min
- cost: 7.180090*USD*

| sku_description | cost |
|---|---|
| Licensing Fee for Google Cloud Dataproc (CPU cost) | 1.104667 |
| Network Egress via Carrier Peering Network - APAC Based | 0.000035 |
| Network Internet Egress from APAC to Americas | 0.000091 |
| Network Internet Egress from APAC to China | 0.000002 |
| Network Internet Egress from APAC to EMEA | 0.000001 |
| Standard Intel N1 8 VCPU running in APAC | 6.075294 |

Assuming $\leq 50\%$ of workers are preemptive, a GWAS would cost not more than 83USD (assuming 5 million SNPs)

# Overall cost


| PROJECT | PROJECT ID | COST BEFORE CREDIT | CREDIT | COST AFTER CREDIT |
|---|---|---|---|---|
| Hail | hail-201606 | $177.96 | -$177.96 | $0.00 |

- Cost control/monitoring is essential
- Costs are only updated every day
- There are a number of hidden costs (minor)

# Data Storage

- We are currently paying 0.02USD per GB per Month (Buckets)
- Variants can also be deposited into Google's Genetic Database for much lower cost (not accessible to Spark)
- Data written onto the disk of master or worker nodes will be lost after cluster is shut down

# Overall Experience

- More expensive than previously assumed
- Very, very flexible
- Spark allows for very fast computations
- Setup is less complicated than expected
- You get things done. Fast.
- Cost can be controlled by assigning fixed budgets for projects

# Distributed Statistical Learning

# Some refreshing

Lets use the simple one half mean squared error cost function:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 \tag{1}$$

and its partial derivative $\frac{d}{d\theta_j} J(\theta)$

$$\frac{d}{d\theta_j} J(\theta) = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} \tag{2}$$

which gives us the following update rule:

$$\theta_j := \theta_j - \alpha \frac{1}{400} \sum_{i=1}^{400} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} \tag{3}$$

How can we parallelize this process?

# Parallelization

Let's assume we have $m = 400$ samples. We could distribute these 400 across different machines

Machine 1: Use $(x^{(1)}, y^1), \ldots, (x^{(100)}, y^{100})$

$$temp_j^{(1)} = \sum_{i=1}^{100} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

Machine 2: Use $(x^{(101)}, y^{101}), \ldots, (x^{(200)}, y^{200})$

$$temp_j^{(2)} = \sum_{i=1}^{100} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

Machine 3: Use $(x^{(201)}, y^{201}), \ldots, (x^{(300)}, y^{300})$

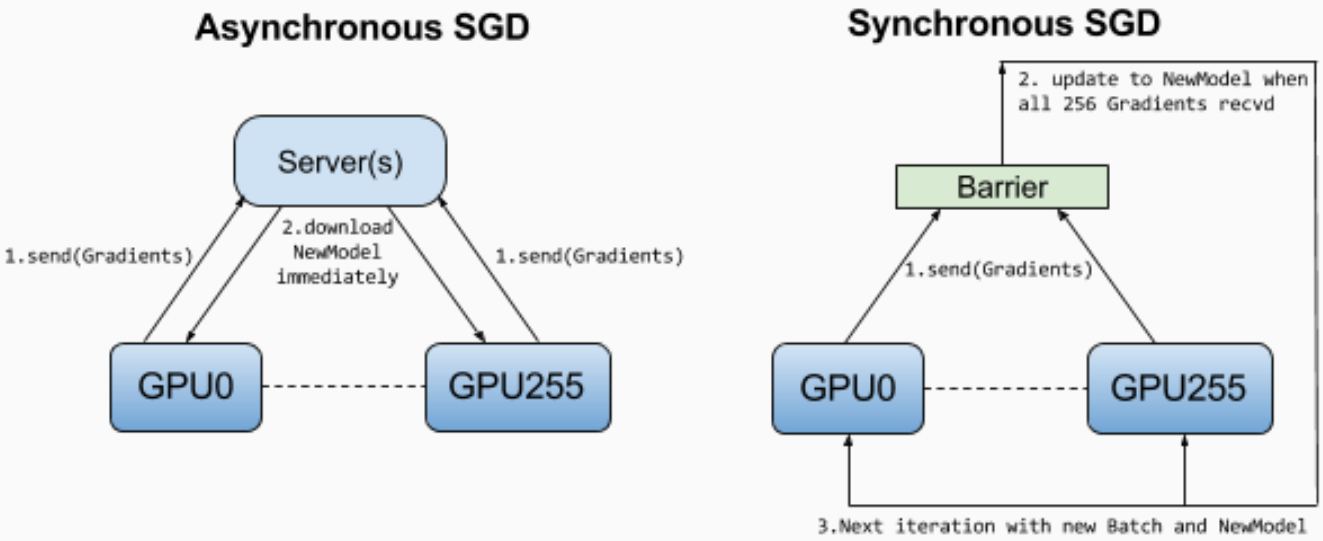
$$temp_j^{(3)} = \sum_{i=1}^{100} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

Machine 4: Use $(x^{(301)}, y^{(301)}), \ldots, (x^{(400)}, y^{400})$

$$temp_j^{(4)} = \sum_{i=1}^{100} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

Then $\theta_j := \theta_j - \alpha \frac{1}{400} (temp_j^{(1)} + temp_j^{(2)} + temp_j^{(3)} + temp_j^{(4)})$

# Synchronous versus asynchronous distributed training[1]




[1] Taken from https://www.oreilly.com/ideas/distributed-tensorflow

# Ring-allreduce