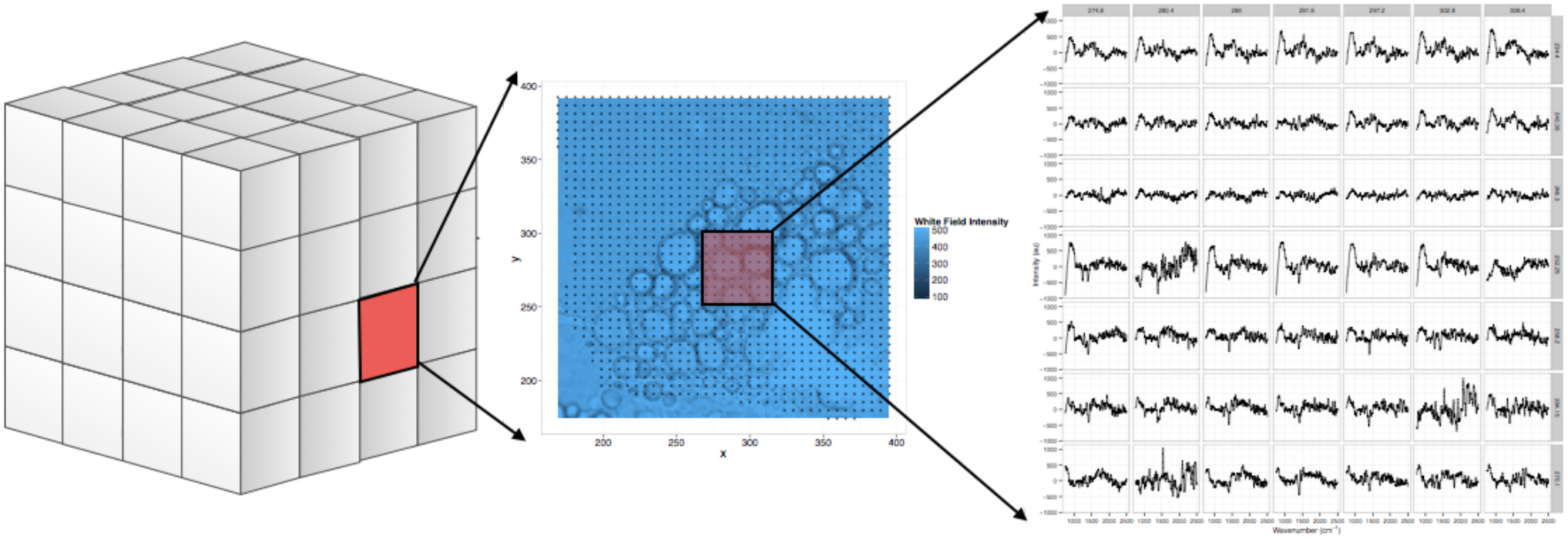
+/- R Code

Making Scaling Simple: Using Big Data to Improve Imaging

author: Kevin Mader (slides: http://bit.ly/1xpbUSf (http://bit.ly/1xpbUSf)) date: January 2015, Swiss Big Data User Group Meeting width: 1440 height: 900 transition: rotate css: beigeplus.css navigation: slide



4Quant

PAUL SCHERRER INSTITUT

PSI

Outline

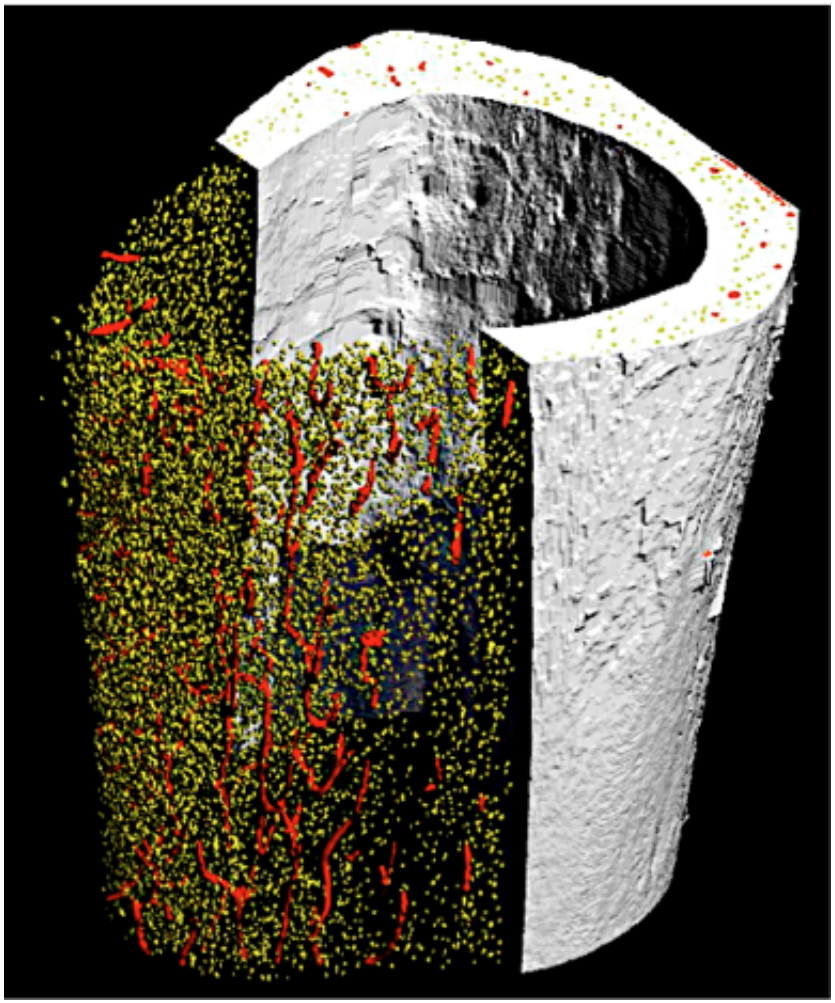
- Background: Our Technique (why we have big data)
  - X-Ray Tomographic Microscopy
- Microscopy in 2014
- The Problem(s)

The Tools

- Spark Imaging Layer
- 3D Imaging
- Hyperspectral Imaging
- Real-Time Streaming

The Science

- Genome Scale Studies
- Large Datasets / Beyond
- Outlook

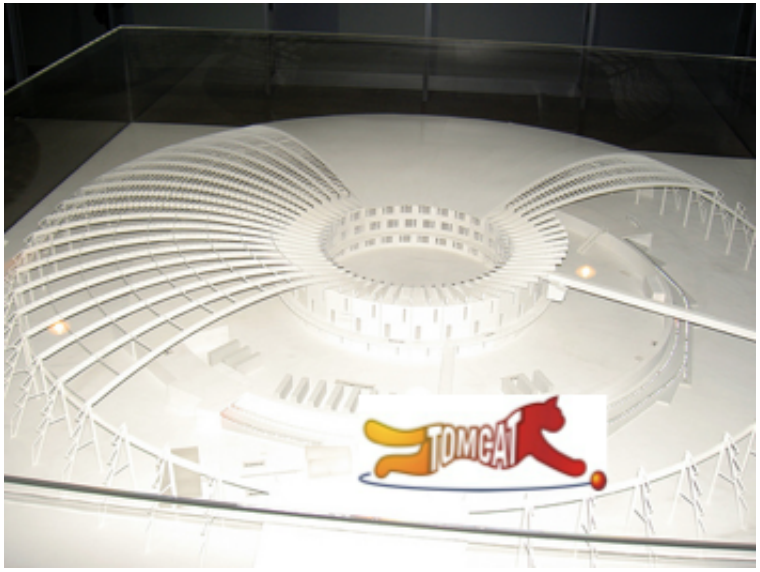


Synchrotron-based X-Ray Tomographic Microscopy

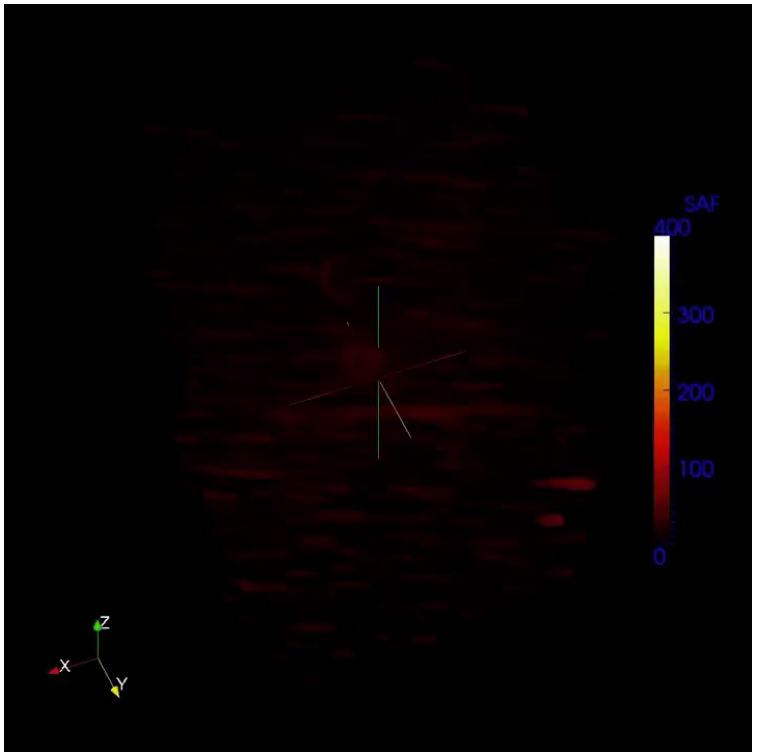
The only technique which can do **all**

- peer **deep** into large samples
- achieve $< \mathbf{1}\mu\mathbf{m}$ isotropic spatial resolution

    - with **1.8mm** field of view
- achieve **>10 Hz** temporal resolution
- **8GB/s** of images



[1] Mokso et al., J. Phys. D, 46(49),2013



Courtesy of M. Pistone at U. Bristol

Microscopy in 2014

Ever more sensitive and faster hardware means image acquisition possibilities are growing rapidly

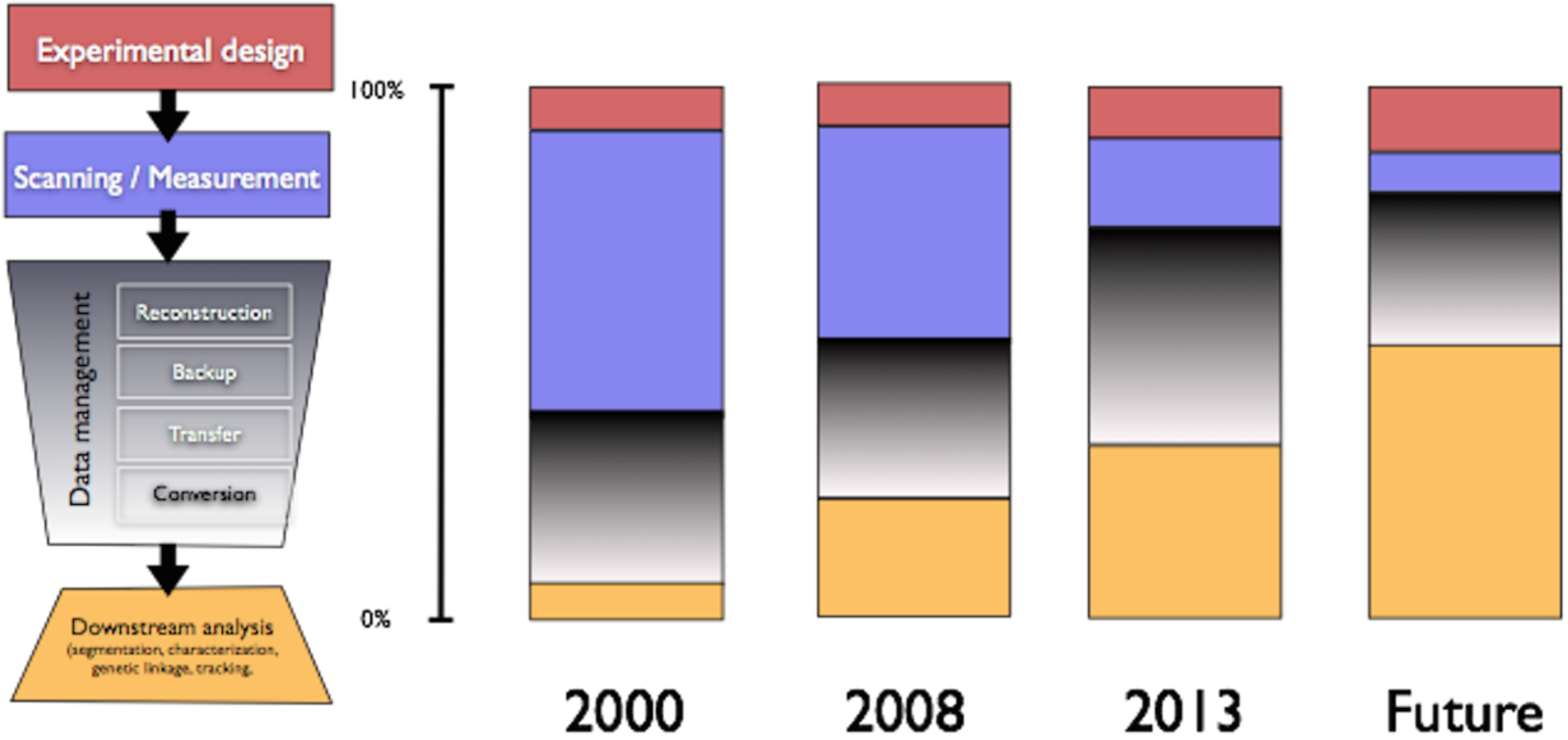
X-Ray

- SRXTM images at (>1000fps) → 8GB/s
- cSAXS thousands of high-resolution diffraction patterns in minutes, acquisition rates might soon reach 30GB/s
- Nanoscopium, 10TB/day, 10-500GB file sizes, multiple pieces of information acquired in parallel

Optical

- Light-sheet microscopy (see talk (http://spark-summit.org/2014/talk/A-platform-for-large-scale-neuroscience) of Jeremy Freeman) produces images → 500MB/s
- High-speed confocal images at (>200fps) → 78Mb/s

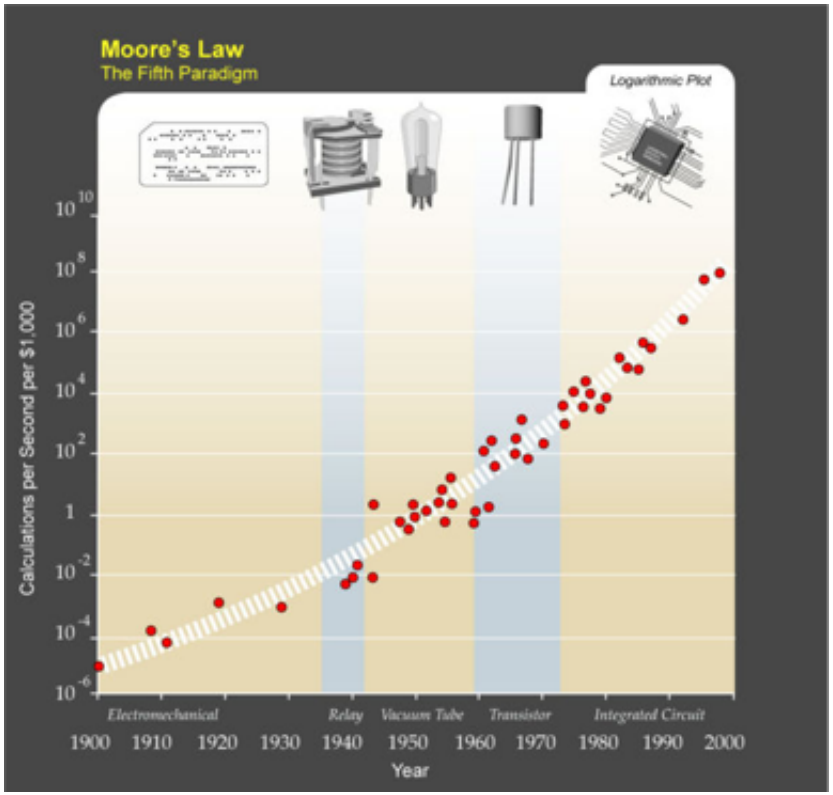
Images are only as useful as what you can do with them, the bottleneck isn't measurement speed, but analysis



Adapted from: Sboner A,et. al. Genome Biology, 2011

The changing world

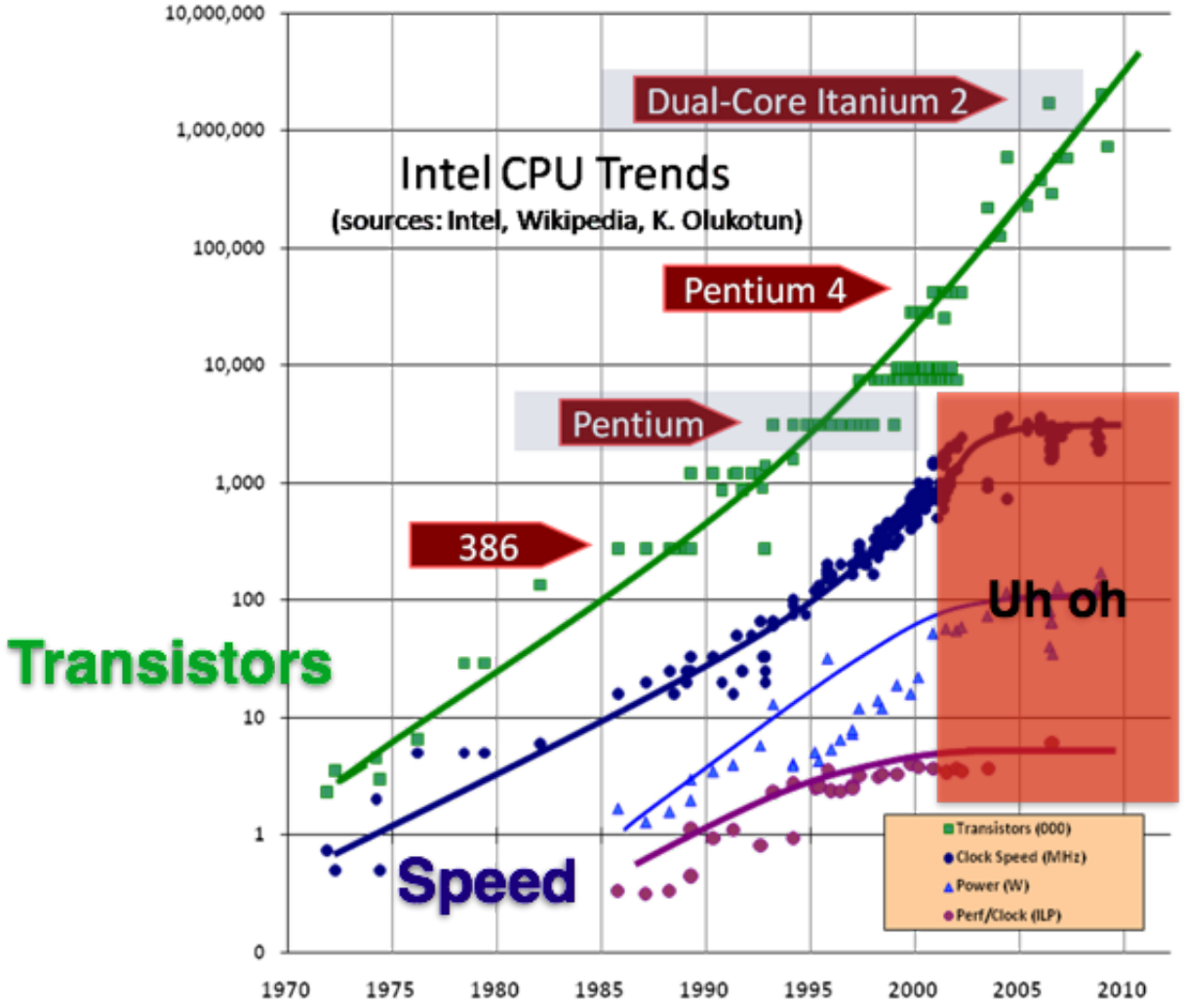
Moore's law is still alive, number of transitors are doubling every 18 months. Traditional solution: Bigger problem means a more expensive computer



- http://en.wikipedia.org/wiki/Moore's_law (http://en.wikipedia.org/wiki/Moore's_law)

But, not how it used to be

Clock cycles per second (speed) and performance per clock cycle has saturated, so *parallelization* is needed for further benefits.



- http://www.gotw.ca/publications/concurrency-ddj.htm (http://www.gotw.ca/publications/concurrency-ddj.htm)

The Problem

There is a flood of new data

What took an entire PhD 3-4 years ago, can now be measured in a weekend, or even several seconds. Analysis tools have not kept up, are difficult to customize, and usually highly specific.

Optimized Data-Structures do not fit

Data-structures that were fast and efficient for computers with 640kb of memory do not make sense anymore

Single-core computing is too slow

CPU's are not getting that much faster but there are a lot more of them. Iterating through a huge array takes almost as long on 2014 hardware as 2006 hardware

The reality

X-Ray Microscopy isn't the first

Google, Facebook, Yahoo, and Amazon had these, or very similar problems years ago. They hired a lot of very competent Computer Science PhDs to solve it.

Cloud computing is ubiquitious and cheap

Cloud computing infrastructures are 10-100X cheaper than labor costs, usually more reliable internal systems, and easily accessible from **almost** anywhere.

The big why!

- You don't ask the questions you **should**, you ask the questions you **can**
- None of these approaches are robust or deal with the data **flood**

- 8 GB/s is more data than Facebook processes a day and 3 times as many images as Instagram (*and they are all 5Mpx 14-bit and we don't compress them*)
- http://news.cnet.com/8301-1023_3-57498531-93/facebook-processes-more-than-500-tb-of-data-daily/ (http://news.cnet.com/8301-1023_3-57498531-93/facebook-processes-more-than-500-tb-of-data-daily/)
- http://techcrunch.com/2013/01/17/instagram-reports-90m-monthly-active-users-40m-photos-per-day-and-8500-likes-per-second/ (http://techcrunch.com/2013/01/17/instagram-reports-90m-monthly-active-users-40m-photos-per-day-and-8500-likes-per-second/)

What we want?

Fast, scalable computing

Run it on your laptop, local cluster, or 5000 machines at Amazon without rewriting anything

- No management of dead-locks, mutability, and blocking
- Test exact same code as you run on the cluster

Reliable

- One machine crashes, the next picks up where it left off
- Run of out memory, just write to disk

Easy

- **Readable**, small code bases which focus on analysis rather than data management details
- No dealing with thread management, message passing, memory allocation

Big Data: Standard Definition

Velocity, Volume, Variety

When a ton of heterogeneous data is coming in fast.

Performant, scalable, and flexible

Save **Everything**

- HDFS and cloud storage make keeping all of the data easy
- Immutable / append only storage makes it the most natural approach

Tons of Hype!

What is big data really?

- Automating everything
  - Leave nothing to 'click'
  - remove mistakes for tired, biased humans to make
  - 10X, 100X, 1000X is the same amount of effort
- Connected, more flexible tools
  - Pipelined analysis
  - Combining and tuning many simple operations
- Higher-level of abstraction
  - Code for the problem
  - Do not code for the logistics
  - Abstract away CPUs, memory, storage, and networks

- Algorithms over People
  - People are expensive, machines are cheap The **right** approach for many small problems.
- Code is very problem oriented and readable
- Typically portable to different underlying platforms (laptop, HPC Cluster, cloud)
- Large collection of complex analyses (graph, machine learning, ...)

- Scalability if/when problems get big

Big Data: A brief introduction

Google ran head-first into *Big Data* trying to create an index of the internet. Single computers cannot do everything and writing parallel and distributed code is complicated, **difficult to test**, easy to break, and often **non-deterministic**. The resulting code is a tangled inseperable mess of computation, data, and processing management.

MapReduce

A robust, fault-tolerant framework for handling distributed processing and data storage on which complicated analyses could be run **declaritively** by specifying **what** not **how**. Reduced data processing vocabulary to two words *Map* and *Reduce*. Recreated and open-sourced by Yahoo as Hadoop.

Apache Spark

MapReduce can be applied to many tasks, but it is often very tricky and time-consuming to describe complicated, interative workflows into just map and reduce. Spark expanded the vocabulary of Hadoop to *filter*, *flatMap*, *aggregate*, *join*, *groupBy*, and *fold*, incorporated many performance improvements through caching (very important for images) and interactive (REPL) shell

- **World Record Holder** for Terabyte and Petabyte Sort
  - 100TB of data on 206 machines in 23 minutes
  - 1PB of data in 190 machines in <4 hours
- Sustained data rates of 3Gb/s/node (map) and 1.1Gb/s/node (reduce)

The Framework First

- Rather than building an analysis as quickly as possible and then trying to hack it to scale up to large datasets
  - chose the framework **first**
  - then start making the necessary tools.
- Google, Amazon, Yahoo, and many other companies have made huge in-roads into these problems

- The real need is a fast, flexible framework for robustly, scalably performing complicated analyses, a sort of Excel for big imaging data.

Apache Spark and Hadoop 2

The two frameworks provide a free out of the box solution for

- scaling to >10000 computers
- storing and processing exabytes of data
- faul tolerance
  - 2/3rds of computers can crash and a request still accurately finishes
- hardware and software platform indpendence (Mac, Windows, Linux)

Spark -> Microscopy?

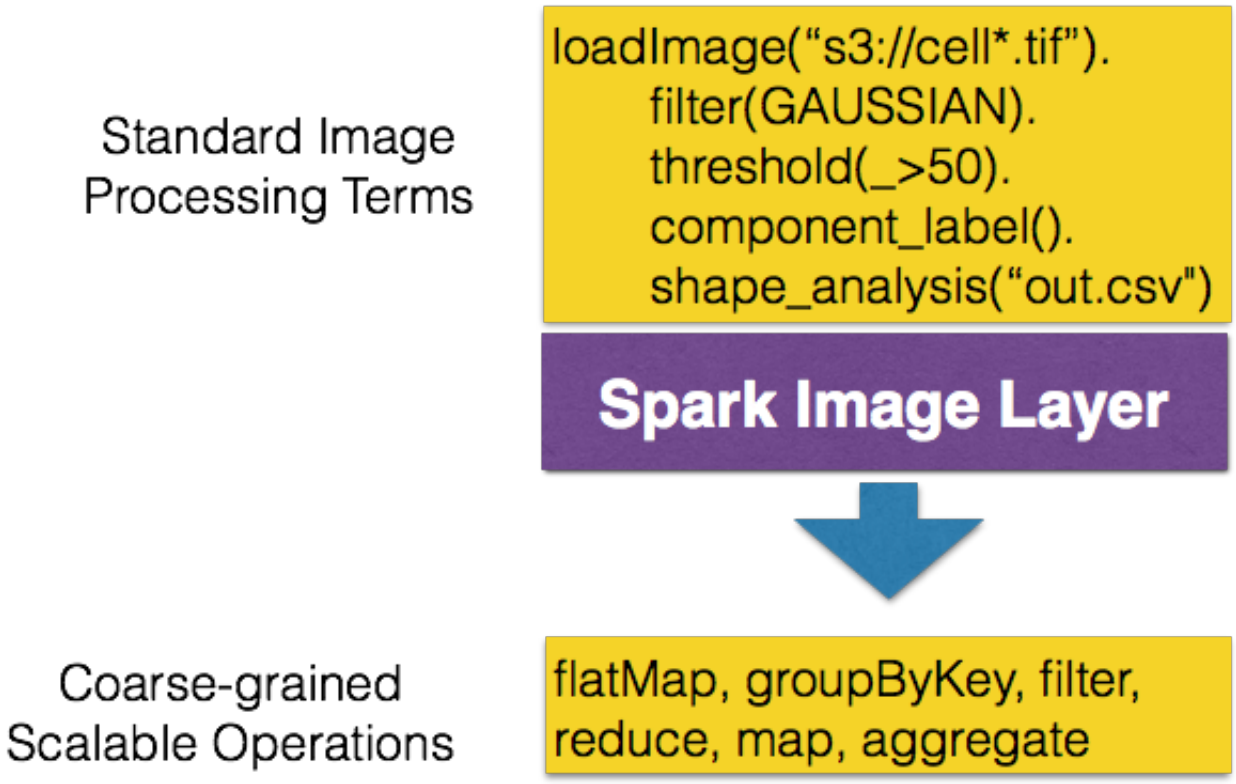
These frameworks are really cool and Spark has a big vocabulary, **but** *flatMap*, *filter*, *aggregate*, *join*, *groupBy*, and *fold* still do not sound like anything I want to do to an image.

I want to

- filter out noise, segment, choose regions of interest
- contour, component label
- measure, count, and analyze
- ...

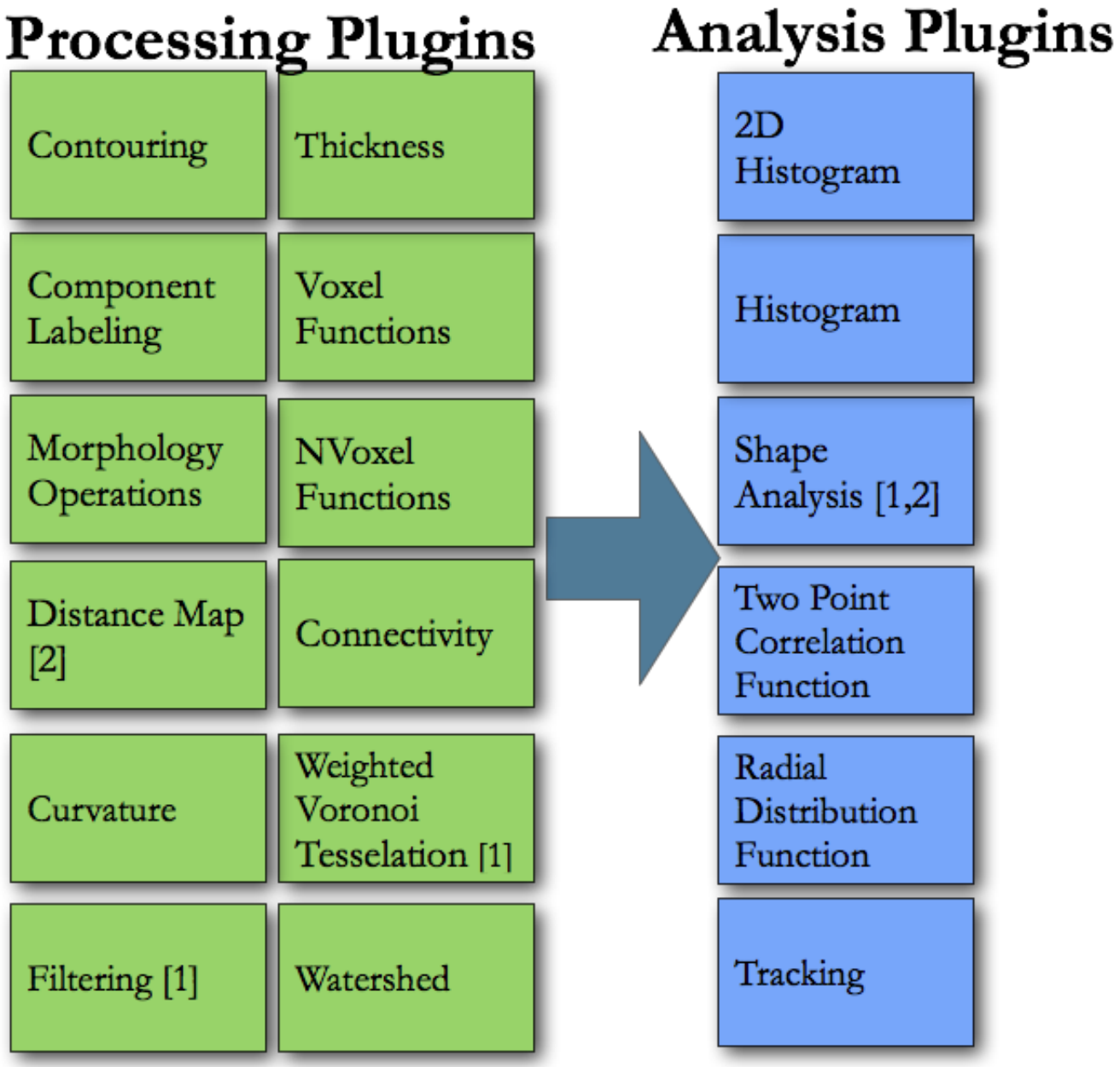
Spark Image Layer

- Developed at 4Quant (http://www.4quant.com), ETH Zurich (http://www.ethz.ch), and Paul Scherrer Institut (http://www.psi.ch)
- The Spark Image Layer is a *Domain Specific Language* for Microscopy for Spark.
- It converts common imaging tasks into coarse-grained Spark operations
- Spark handles processing, fault-tolerance, and resource management

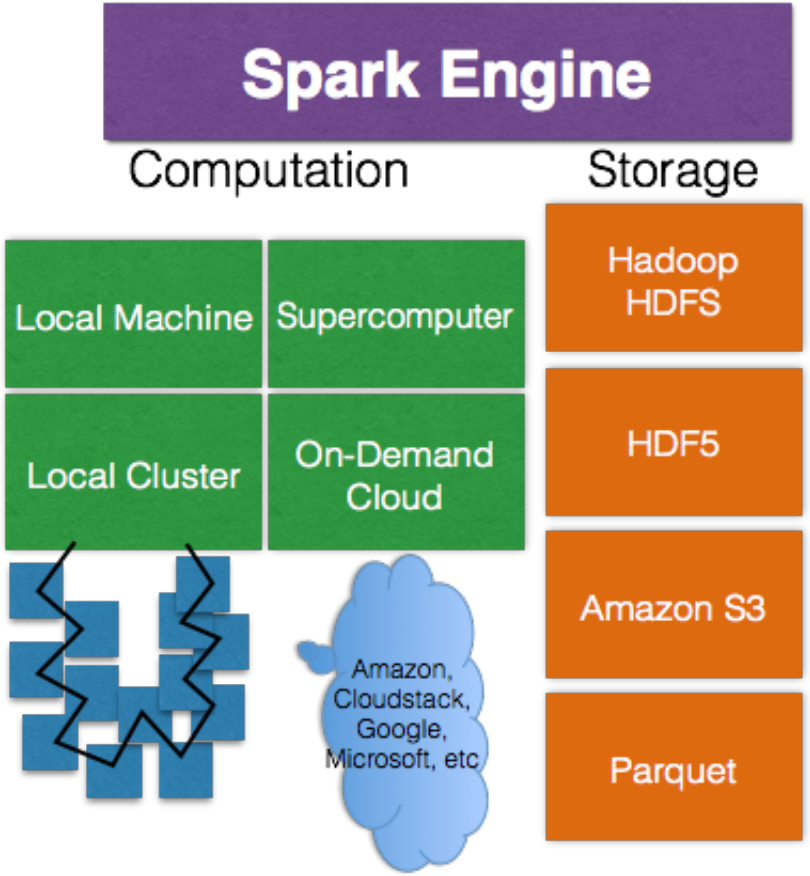


Spark Image Layer

We have developed a number of commands for SIL handling standard image processing tasks


Processing Plugins
Contouring
Thickness
Component Labeling
Voxel Functions
Morphology Operations
NVoxel Functions
Distance Map [2]
Connectivity
Curvature
Weighted Voronoi Tesselation [1]
Filtering [1]
Watershed
Analysis Plugins
2D Histogram
Histogram
Shape Analysis [1,2]
Two Point Correlation Function
Radial Distribution Function
Tracking

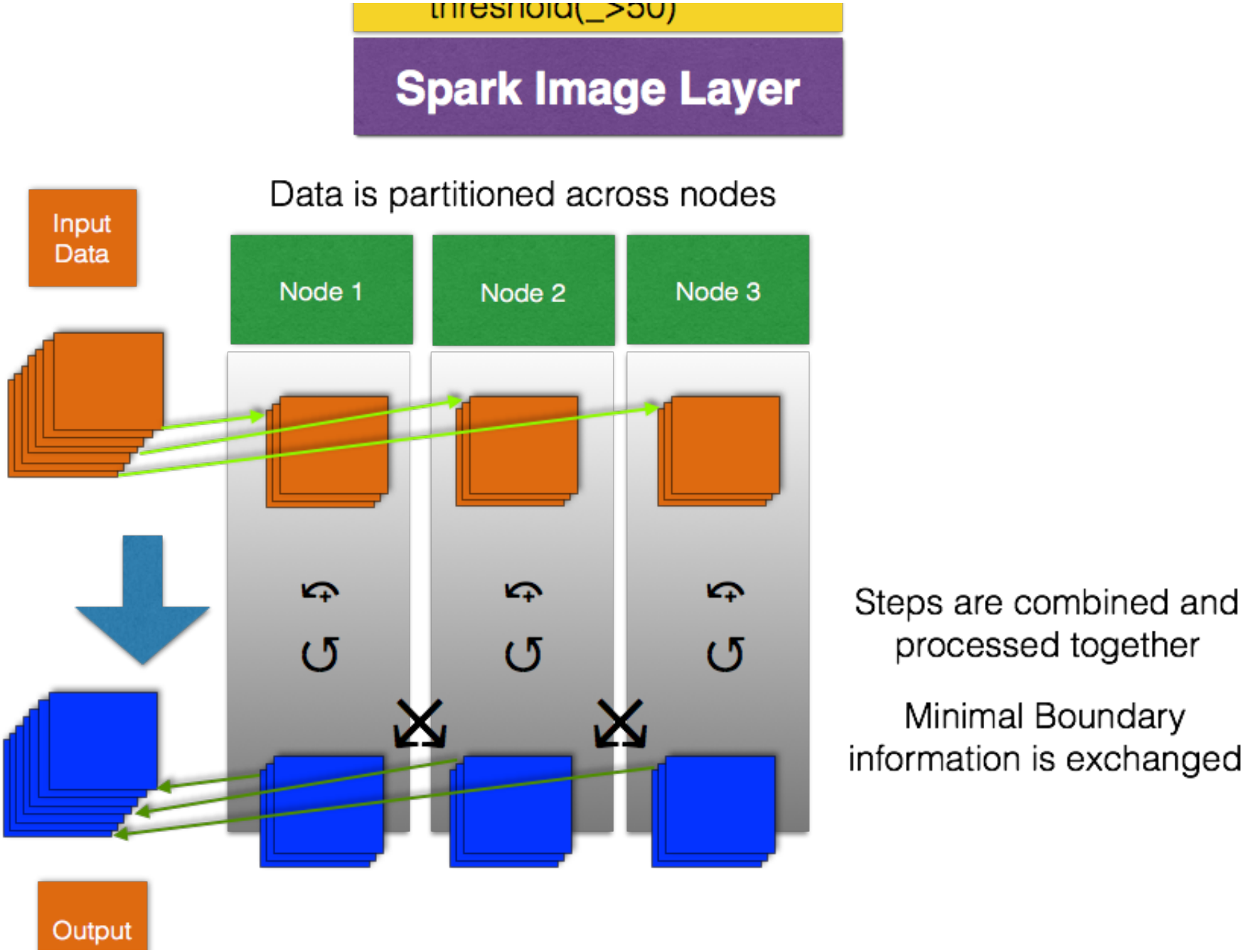
New components can be added using Scala python Java or imported from ImageJ directly. The resulting analyses are then *parallelized* by the Spark Engine and run on multiple cores/CPUs locally, on a cluster, supercomputer, or even a virtual in the cluster in the cloud which can be started in seconds.



How does scaling look?

Input Command

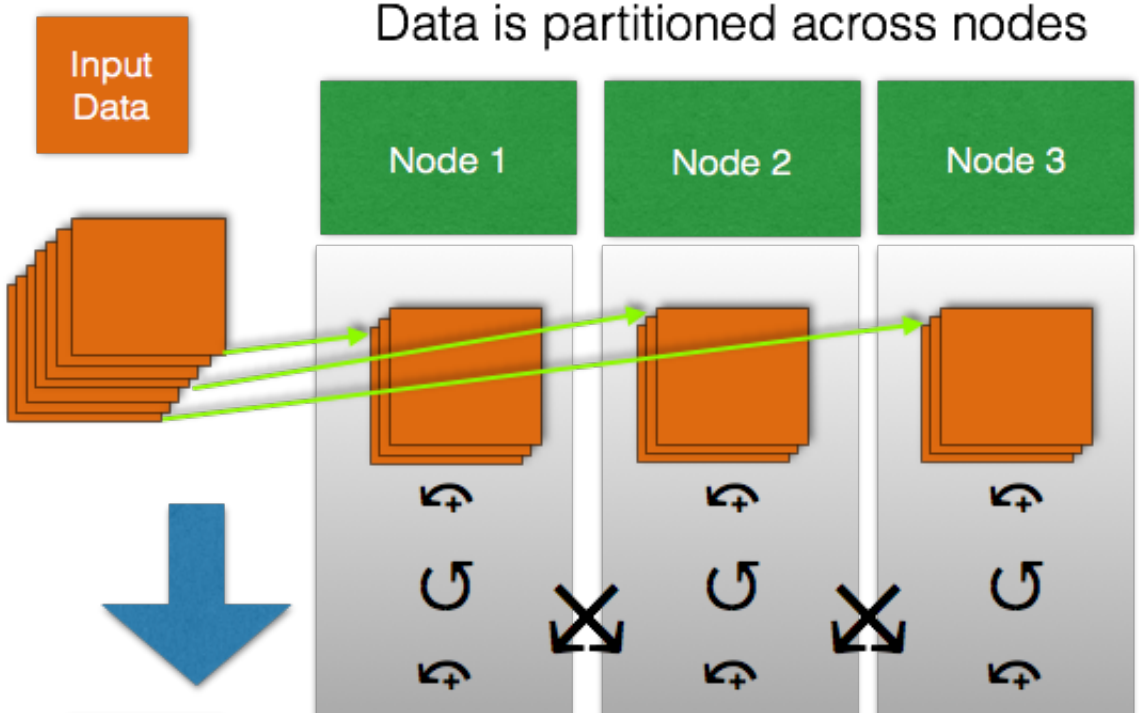
```
loadImage("s3://cell*.tif").
    filter(GAUSSIAN).
```


threshold(_>50)
Spark Image Layer
Data is partitioned across nodes
Input Data
Node 1
Node 2
Node 3
Output
Steps are combined and processed together
Minimal Boundary information is exchanged

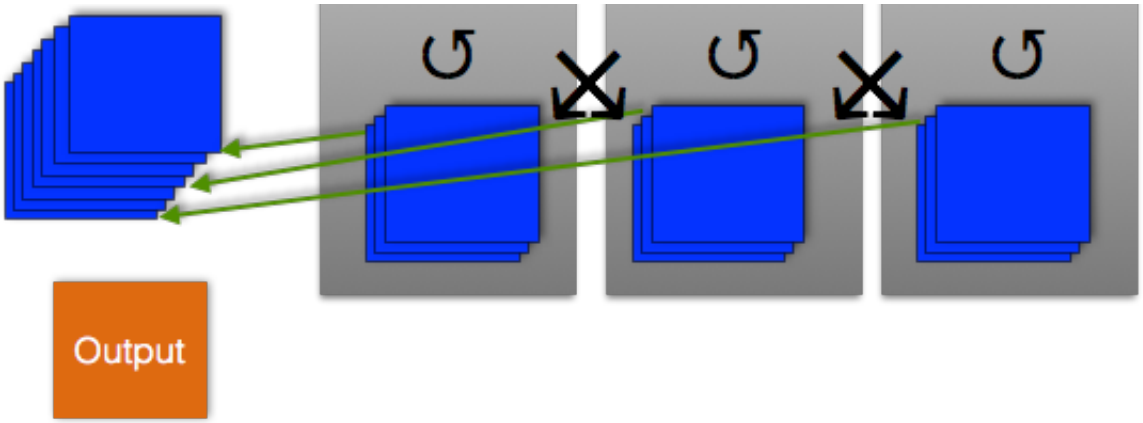
```
loadImage("s3://cell*.tif").
    filter(GAUSSIAN).
    threshold(_>50).
    component_label().
    shape_analysis("out.csv")
```

Spark Image Layer

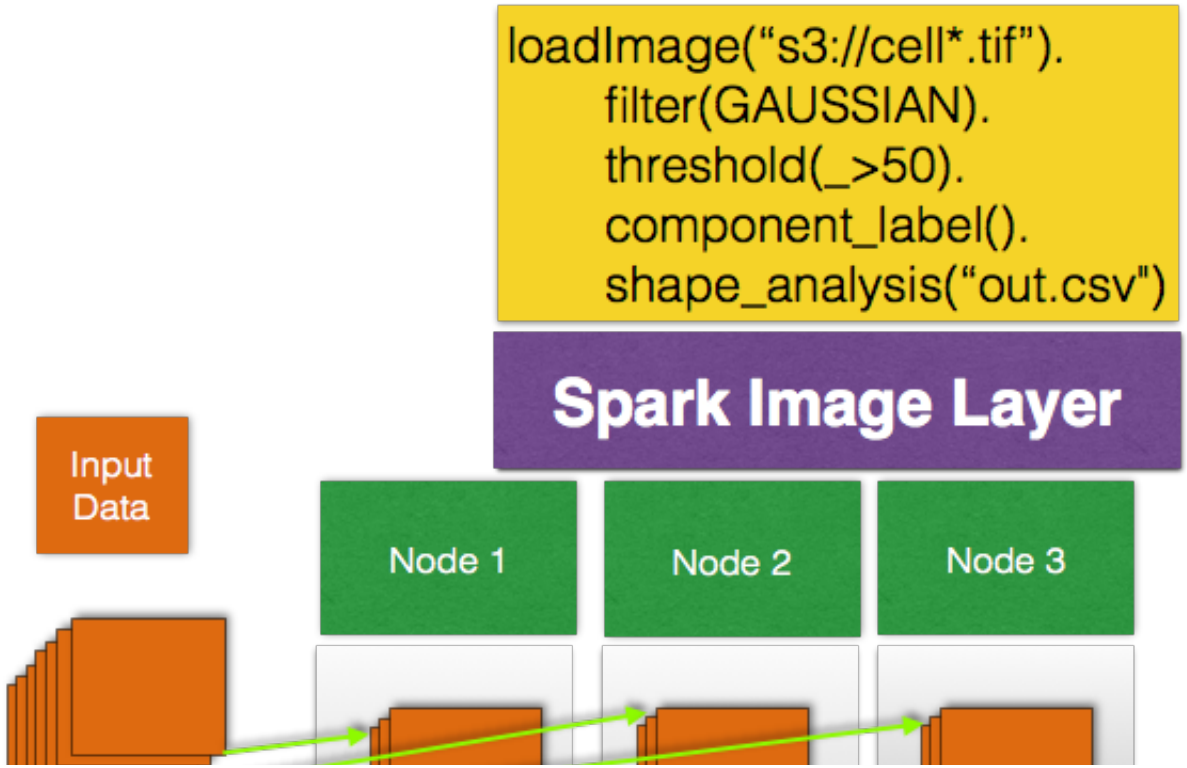
Data is partitioned across nodes

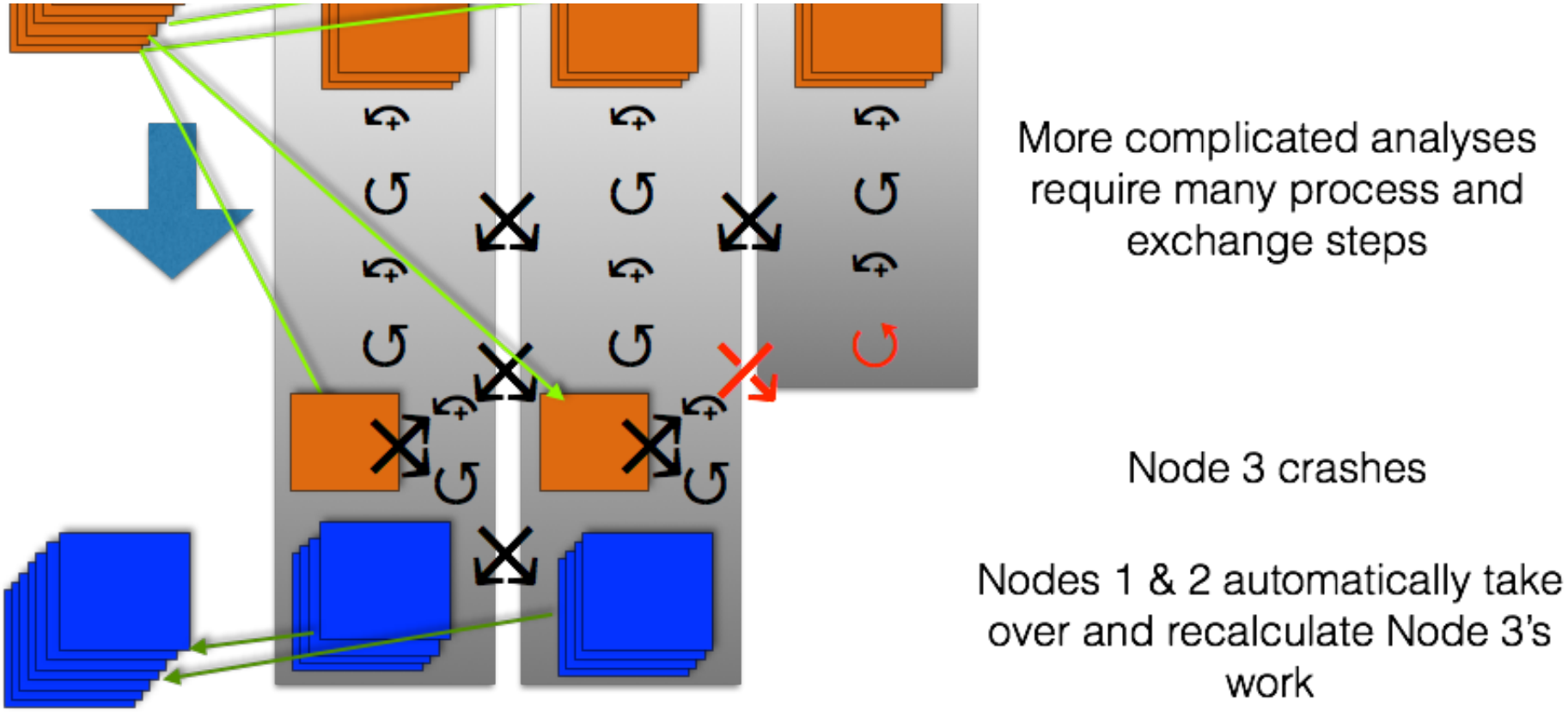


More complicated analyses require many process and exchange steps



Fault-Tolerance





Fault-tolerance is particularly tedious to add into existing tools, it must be done from the first step. Spark Imaging Layer is fault-toleranct and allows checkpointing with single `result.checkpoint()` commands enabling long running analyses to continue

Flexibility through Types

Developing in Scala brings additional flexibility through types[1], with microscopy the standard formats are 2-, 3- and even 4- or more dimensional arrays or matrices which can be iterated through quickly using CPU and GPU code. While still possible in Scala, there is a great deal more flexibility for data types allowing anything to be stored as an image and then processed as long as basic functions make sense.

[1] Fighting Bit Rot with Types (Experience Report: Scala Collections), M Odersky, FSTTCS 2009, December 2009

What is an image?

A collection of positions and values, maybe more (not an array of double). Arrays are efficient for storing in computer memory, but often a poor way of expressing scientific ideas and analyses.

- Filter Noise?

```
combine information from nearby pixels
```

- Find objects

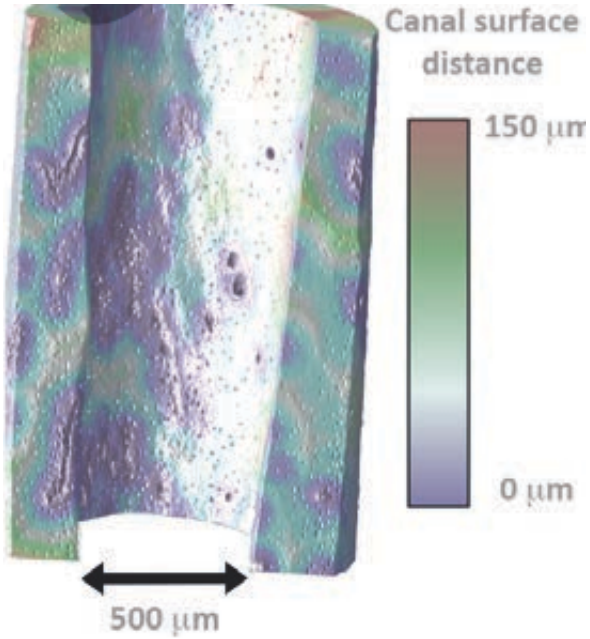
```
determine groups of pixels which are very similar to desired result
```

Practical Meaning

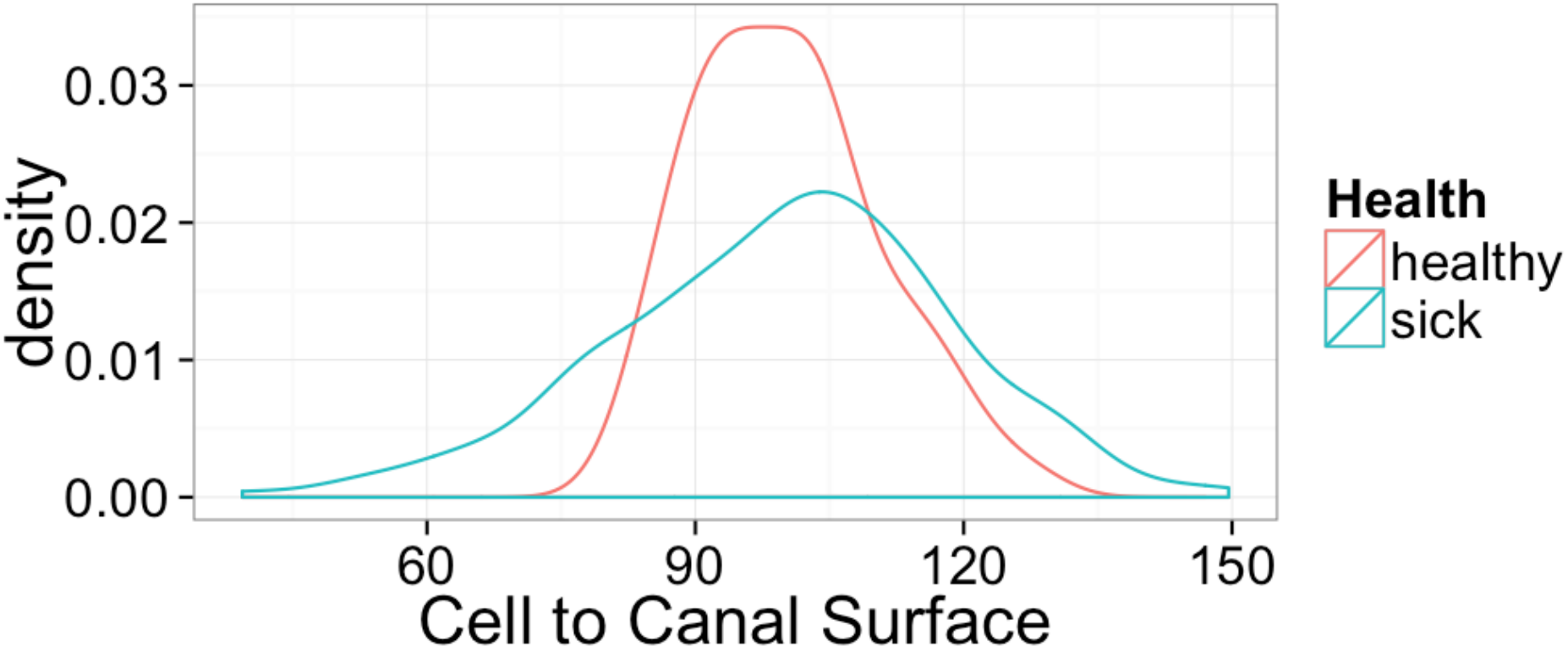
Much simplier code and combining different pieces of information is trivial. For example calculating the distance to the nearest vessel and then determining the average distance of each cell

```
cellDist = vesselImage.invert.
  calculateDistance
cellLabel = cellThreshold.makeLabel
avgCellDist = cellLabel.join(cellDist).
  groupBy(LABEL).reduce(MEAN)
```

This analysis can be done easily since the types are flexible



+/- R Code

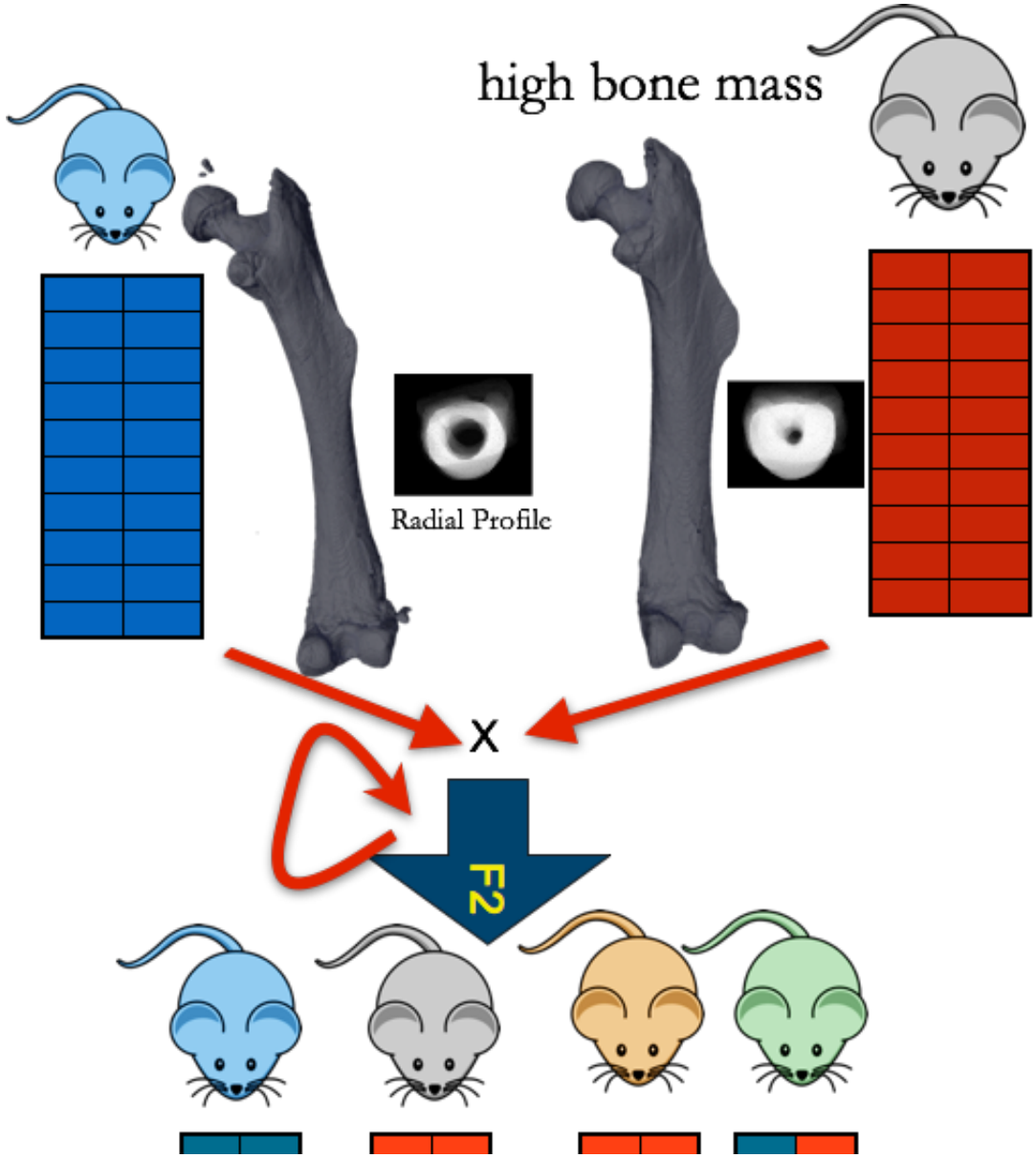


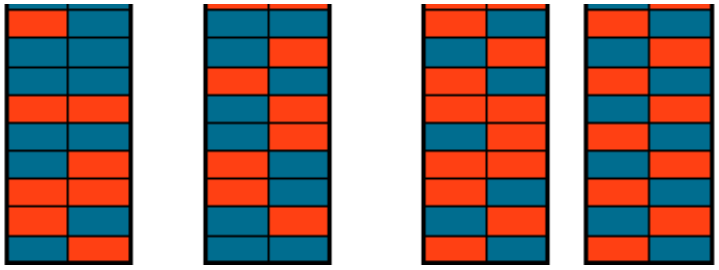
How does this produce better science?

We want to understand the relationship between genetic background and bone structure

- With existing tools, analysis is possible and a number of publications have been made, even ones that show differences between strains of mice
- But
  - n<12
  - time-consuming (years between measurement and publication)
  - not flexible or reproducible

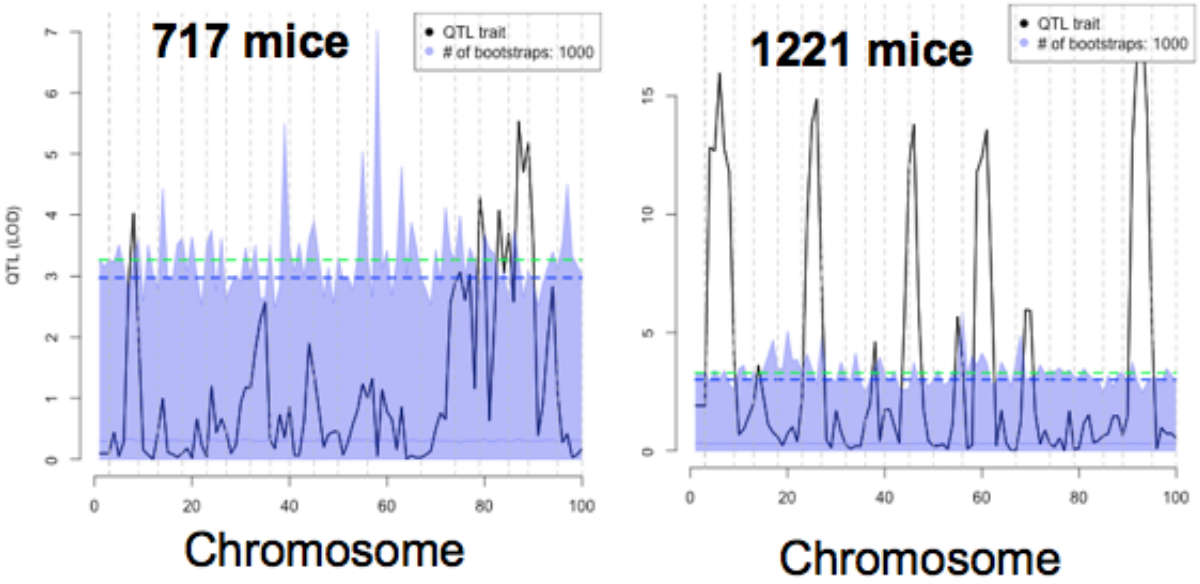
- not cloud-based





How does this produce better science?

Genetic studies require hundreds to thousands of samples



Standard approach (2008)

- Hand Identification -> 30s / object
- 30-40k objects per sample
- One Sample in 6.25 weeks
- One Genome-scale study in **120 years**

Today's approach (2014)

- ImageJ macro for segmentation (2-4 hours / sample)
- Python script for shape analysis (3 hours / sample)
- Paraview macro for network and connectivity (2 hours / sample)
- Python script to pool results (3-4 hours)
- MySQL Database storing results (5 minutes / query)
- One study in 1.5 years

Genetic Studies using Big Data

- Analysis could be completed in several months (instead of 120 years, could now be completed in days in the cloud)
- Data can be freely explored and analyzed
  - `val bones = sc.loadImages("work/f2_bones/*/bone.tif")`
  - `val cells = sc.textFile("work/f2_bones/*/lacuna.csv")`
- Collaborators / Competitors can verify results and extend on analyses
- New hypotheses and analyses can be done in seconds / minutes

+/- R Code

Task	Single Core Time	Spark Time (40 cores)
Load and Preprocess	360 minutes	10 minutes
Single Column Average	4.6s	400ms

1 K-means Iteration	2 minutes	1s

The science beyond one study

A framework is only as useful as the scientific studies it enables. Here we show the new types of science that are possible using the Spark Imaging Layer. In particular

- genome-science
- whole-brain imaging
- 4D rheology
- reproducible parameter selection

Our Large Scale Imaging Projects

Zebra fish Phenotyping Project

Collaboration with Keith Cheng, Ian Foster, Xuying Xin, Patrick La Raviere, Steve Wang

1000s of samples of full adult animals, imaged at 0.74 μm resolution: Images 11500 x 2800 x 628 $\longrightarrow$ 20-40GVx / sample

- Identification of single cells (no down-sampling)
- Cell networks and connectivity
- Classification of cell type
- Registration with histology

Brain Project

Collaboration with Alberto Astolfo, Matthias Schneider, Bruno Weber, Marco Stampanoni

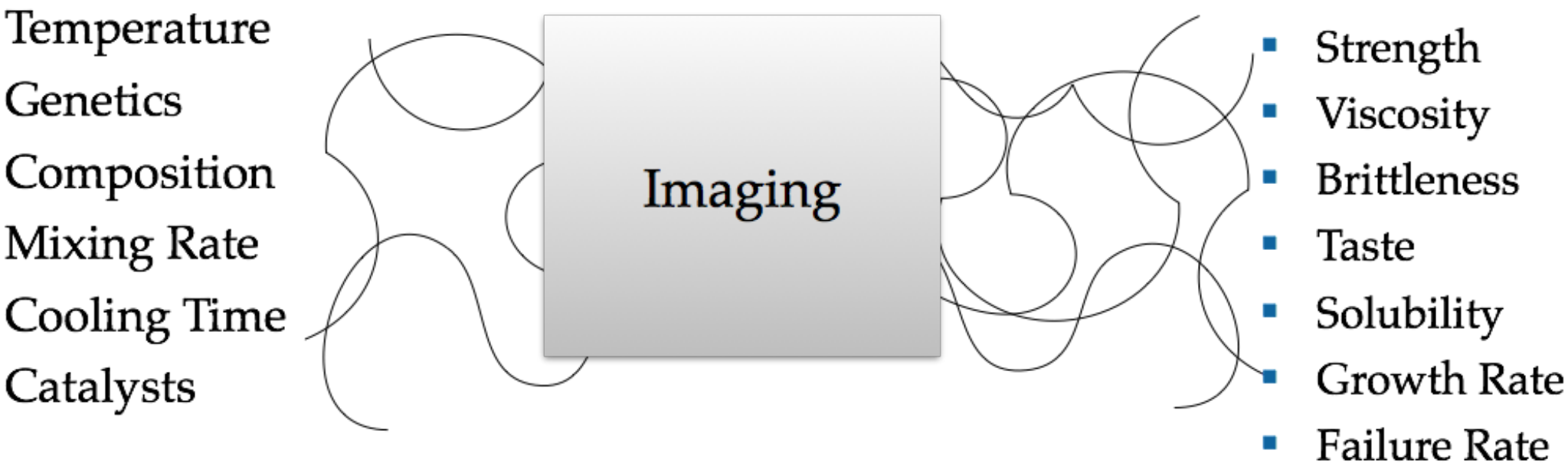
1 cm^3 scanned at 1 μm resolution: Images $\longrightarrow$ 1000 GVx / sample

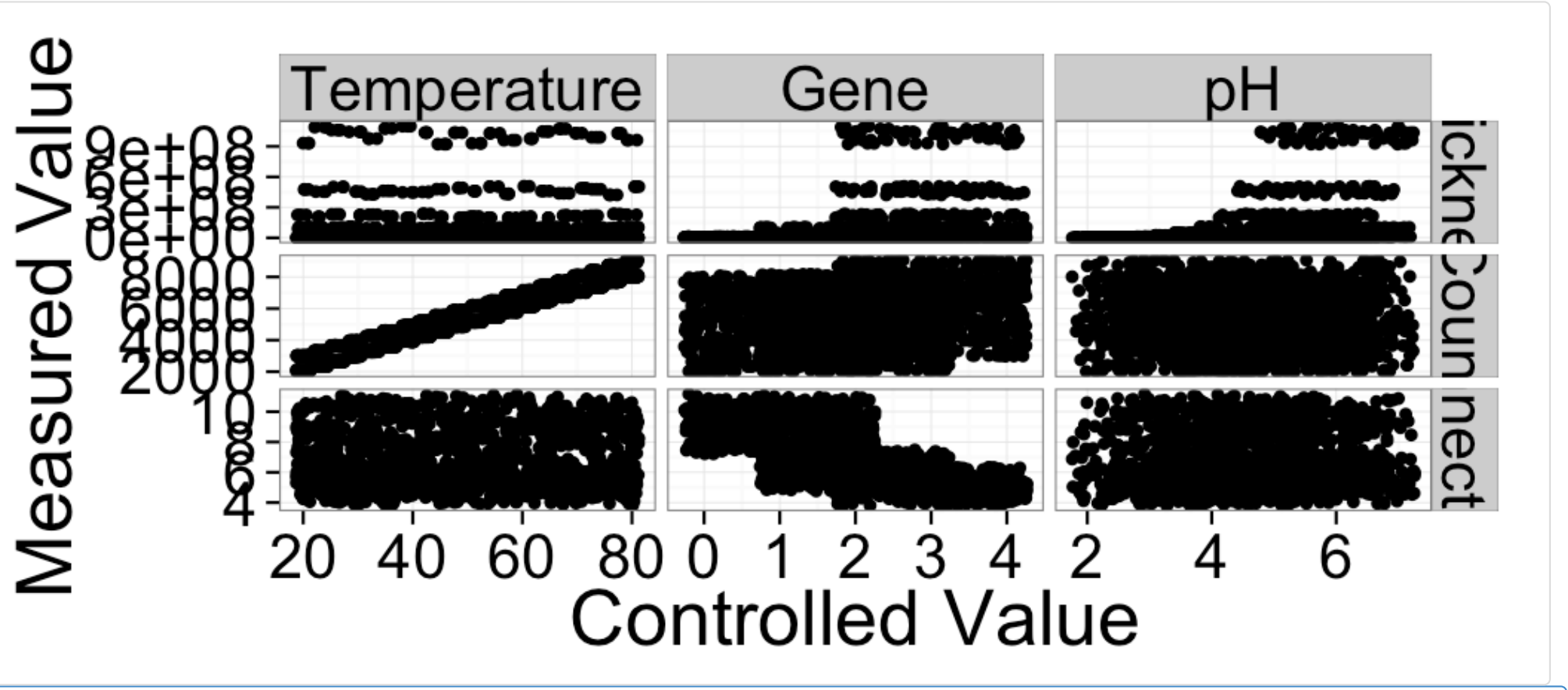
- Registration separate scans together
- Blood vessel structure and networks
- Registration with fMRI, histology

Imaging as Machine Learning

Scalable high-throughput imaging enables new realms of investigation and analysis

- input
  - sample: genetic background, composition, etc
  - environment: treatments, temperature, mechanical testing, etc
- output
  - structure: cell count, shape, distribution, thickness
  - function: marker/tag localization, etc
- prediction
  - identify how a single input affects the outputs





+/- R Code

Real-time with Spark Streaming

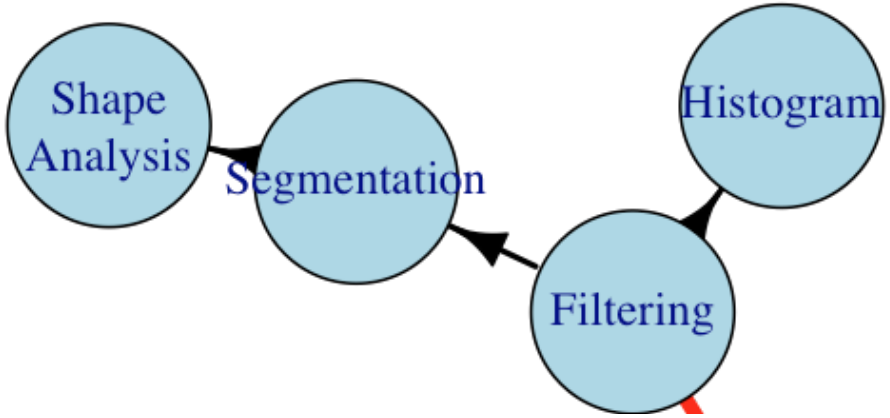
Before

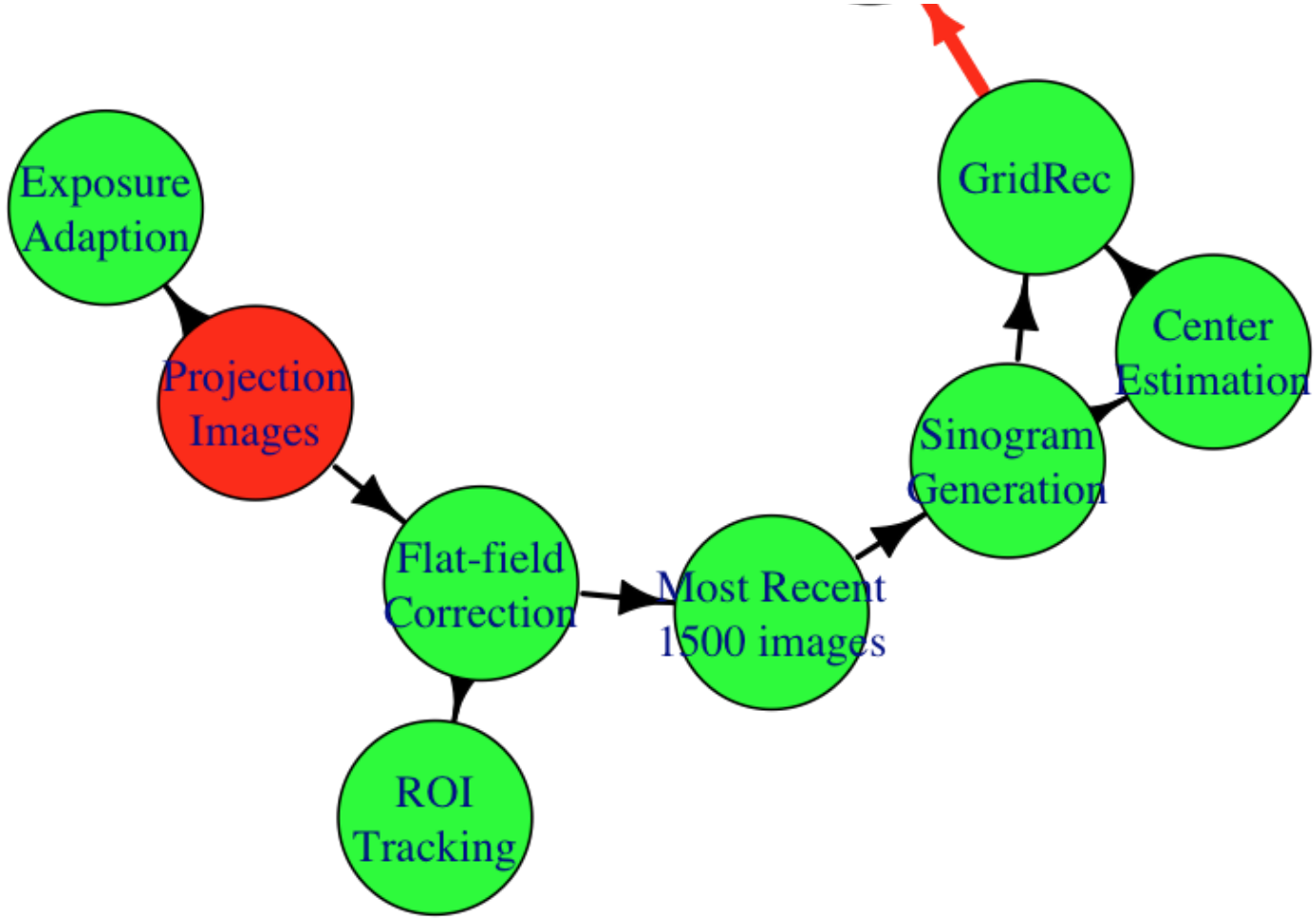
- Spec Macro to control acquisition

- C program to copy data to network drive
- Python-wrapped C program to locally create sinograms
- Highly optimized C program called from Python script through PHP website
- Bash scripts to copy data
- DCL Scripts to format data properly
- Proprietary image processing tools (with clicking)
- Output statistics with Excel

With Spark

- Spec Macro to control acquisition
- Data into a ZeroMQ pipe
- Spark Streaming on ZeroMQ pipe




Exposure Adaption
Projection Images
Flat-field Correction
ROI Tracking
Most Recent 1500 images
Sinogram Generation
Center Estimation
GridRec

+/- R Code

Real-time with Spark Streaming: Webcam

```
val wr = new WebcamReceiver(StorageLevel.MEMORY_ONLY,wrDelay,wrThreads)
val ssc = sc.toStreaming(strTime)
val imgList = ssc.receiverStream(wr)
```

Filter images

```
val filtImgs = allImgs.mapValues(_.run("Median...","radius=3"))
```

Identify Outliers in Streams

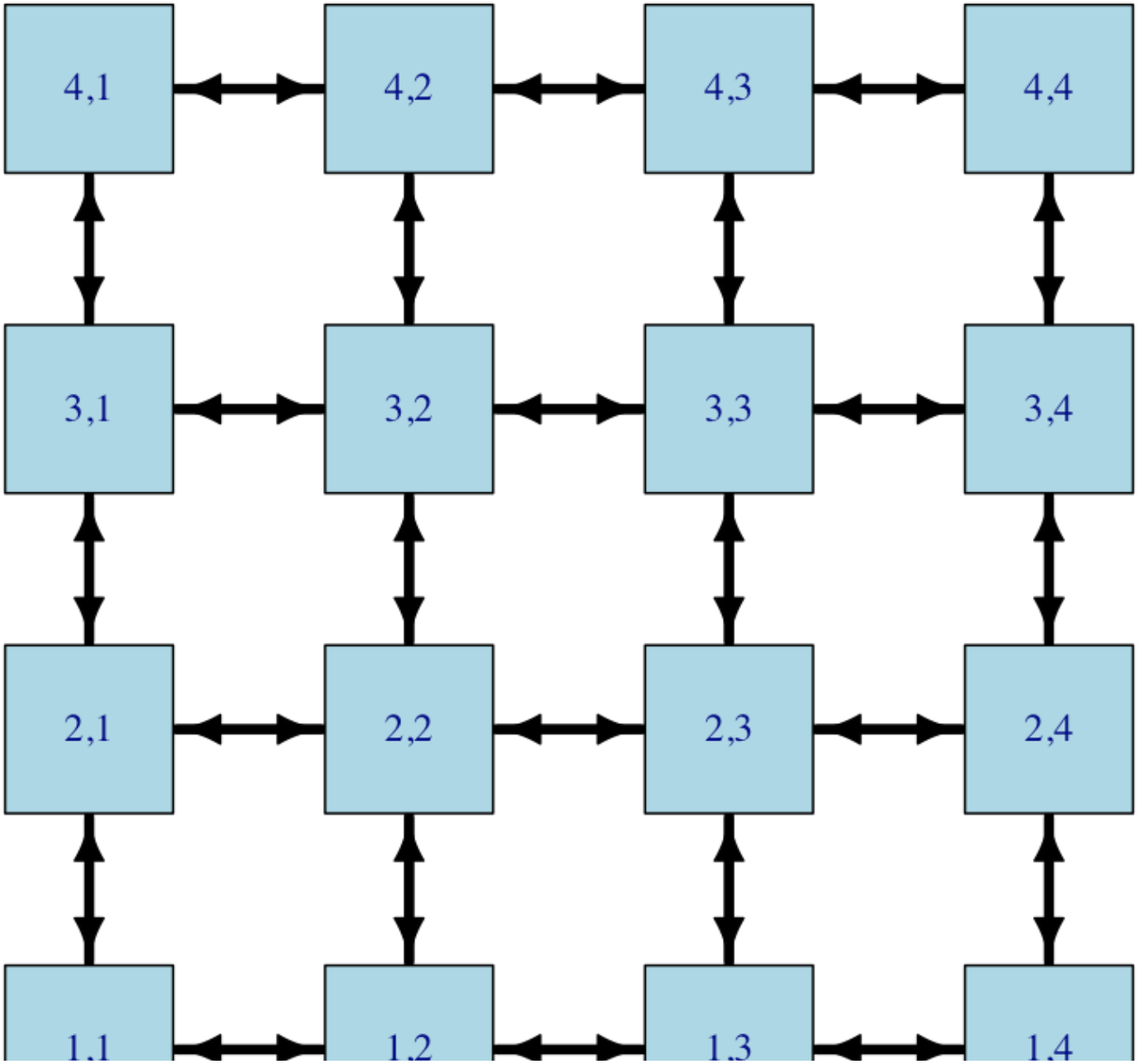
```
val eventImages = filtImgs.
    transform{
    inImages =>
      val totImgs = inImages.count()
      val bgImage = inImages.values.reduce(_.average(_,1.0f)).multiply(1.0/totImgs)
      val corImage = inImages.map {
        case (inTime,inImage) =>
          val corImage = inImage.subtract(bgImage)
          (corImage.getImageStatistics().mean,(inTime,corImage))
      }
      corImage
  }
```

Beyond Image Processing

For many datasets processing, segmentation, and morphological analysis is all the information needed to be extracted. For many systems like bone tissue, cellular tissues, cellular materials and many others, the structure is just the beginning and the most interesting results come from the application to physical, chemical, or biological rules inside of these structures.

$$\sum_j \vec{F}_{ij} = m\ddot{x}_i$$

Such systems can be easily represented by a graph, and analyzed using GraphX in a distributed, fault tolerant manner.


4,1
4,2
4,3
4,4
3,1
3,2
3,3
3,4
2,1
2,2
2,3
2,4
1,1
1,2
1,3
1,4

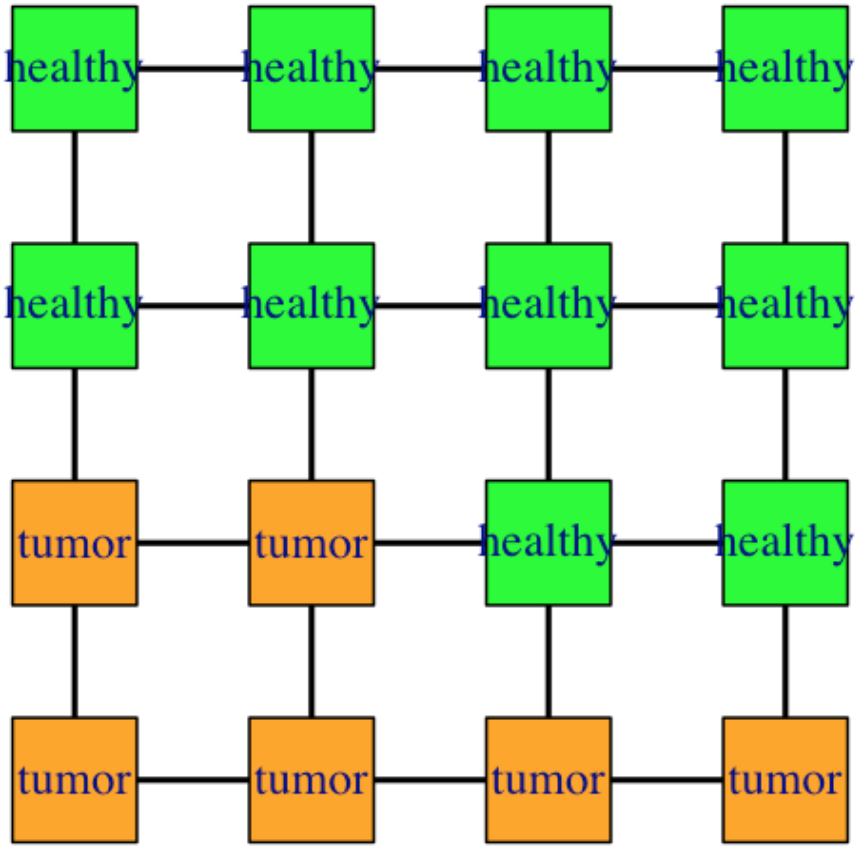
+/- R Code

Cellular Potts Simulations

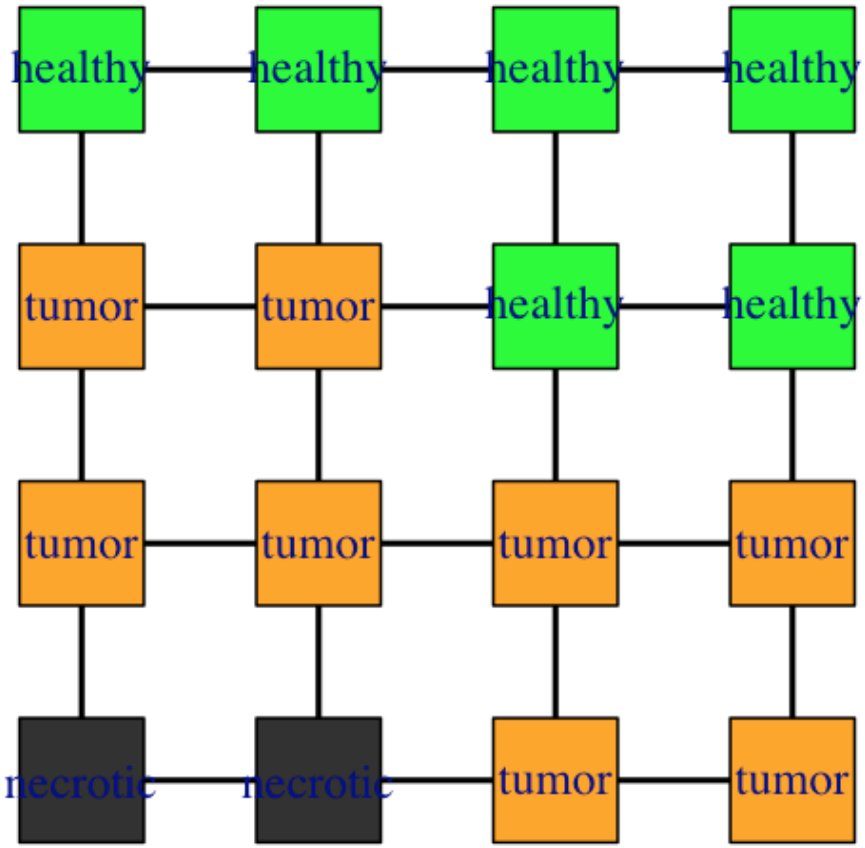
For cellular systems, a model called the Cellular Potts Model (CPM) is used to simulate growth, differentiation, coarsening, phase changes, and a number of similar properties for complex systems. Implementing such algorithms has traditionally been very tedious requiring thousands of lines of optimized MPI code (which is difficult to program well and in a stable manner). For real imaging data, the number of elements in these simulations can exceed 14 billion elements with 378 billion edges run over thousands of iterations

$$E = \frac{1}{2}\sum_{i}\sum_{\langle j \rangle_i} J[1-\delta(S_i - S_j)]$$

Thomas, G., de Almeida, R., & Graner, F. (2006). Coarsening of three-dimensional grains in crystals, or bubbles in dry foams, tends towards a universal, statistically scale-invariant regime. Physical Review E, 74(2), 021407. doi:10.1103/PhysRevE.74.021407


healthy
healthy
healthy
healthy
healthy
healthy
healthy
healthy
tumor
tumor
healthy
healthy
tumor
tumor
tumor
tumor

+/- R Code


healthy
healthy
healthy
healthy
tumor
tumor
healthy
healthy
tumor
tumor
tumor
tumor
necrotic
necrotic
tumor
tumor

+/- R Code

Beyond: Approximate Results

Extensions of Spark out of AMPLab like BlinkDB are moving towards approximate results.

- Instead of `mean(volume)`
  - `mean(volume).within_time(5)`
  - `mean(volume).within_ci(0.95)`

For real-time image processing, with tasks like component labeling and filtering it could work well

- are all 27 neighbor voxels needed to get a good idea of the result?
- must every component be analyzed to get an idea about the shape distribution?

It provides a much more robust solution than taking a small region of interest or scaling down

Beyond: Improving Performance

Scala code can be slow, but it is very high level and can be automatically translated to CPU or GPU code with projects like ScalaCL

```
val myData = new Array(1,2,3,4,5)
val doubleSum = myData.map(_*2).reduce(_+_)
```

Using **ScalaCL** to run on GPUs with 2-4X performance improvement

```
val clData = myData.cl
val doubleSum = clData.map(_*2).reduce(_+_)
```

Pipe

Spark Imaging Layer is fully compatible with the notion of 'pipes' replacing pieces of code with other languages and tools.

- Python scripts
- Compiled code
- GPU processing tools

We can also automatically choose among (and validate) different tools for the same analysis for increasing performance. Running on your laptop means you can profile code and see exactly where the bottlenecks are → `Premature Optimization is the source of all evil - Donald Knuth`

Reality Check

- Spark is **not** performant → dedicated, optimized CPU and GPU codes will perform slightly to much much better when evaulated by pixels per second per processing power unit
  - these codes will be wildly outperformed by dedicated hardware / FPGA solutions
- Serialization overhead and network congestion are not neglible for large datasets

---

But

- Scala / Python in Spark is substantially easier to write and test
  - Highly optimized codes are very inflexible
  - Human time is 400x more expensive than AWS time
  - Mistakes due to poor testing can be fatal
- Spark scales smoothly to enormous datasets
  - GPUs rarely have more than a few gigabytes
  - Writing code that pages to disk is painful
- Spark is hardware agnostic (no drivers or vendor lock-in)

We have a cool tool, but what does this mean for me?

A spinoff - **4Quant**: From images to insight

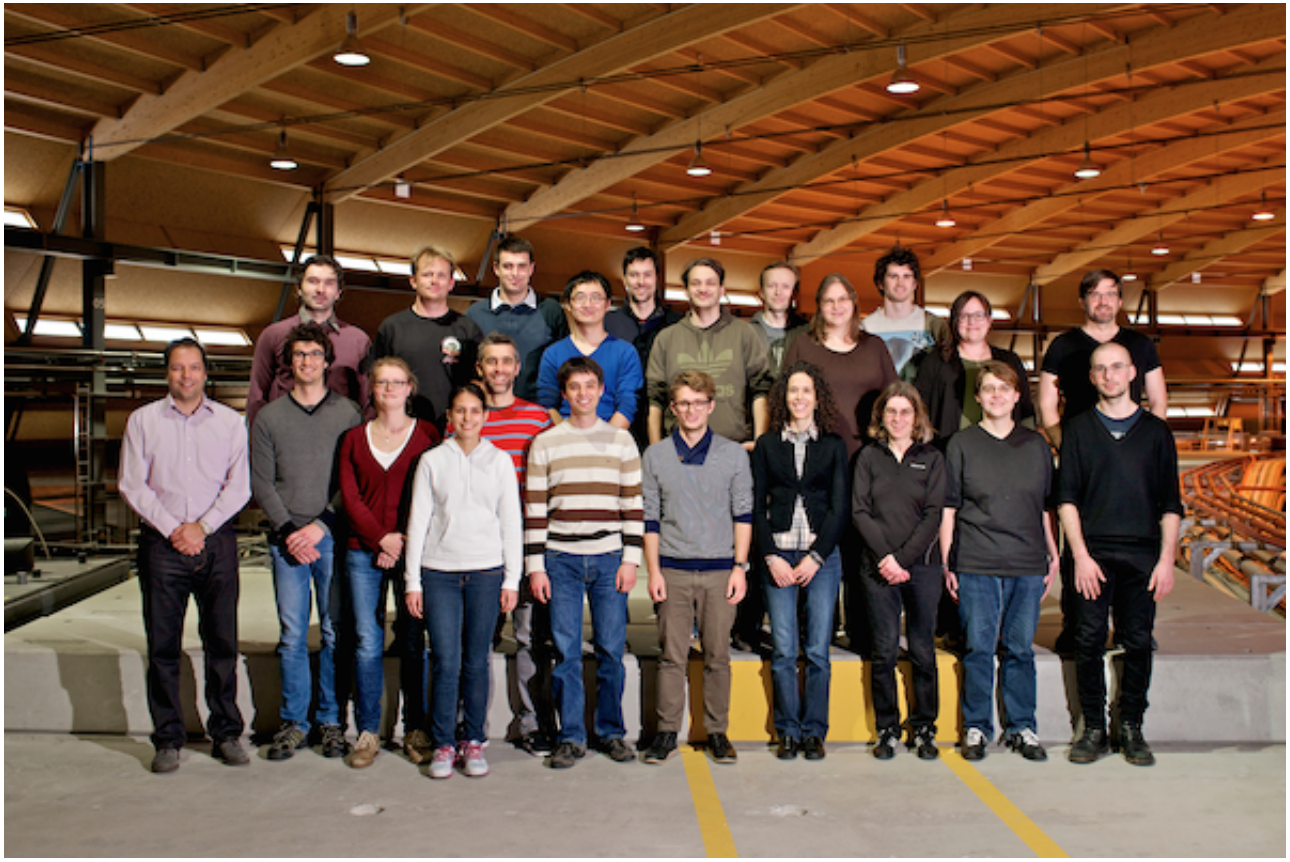
- Cloud Image Processing
  - Use our distributed version of ImageJ in the cloud to analyze thousands of remote datasets using your own (or community provided) processing routines
- Custom Analysis Solutions
  - Custom-tailored software to solve your problems
- One Stop Shop
  - Measurement, analysis, and statistical analysis

Education / Training

- Consulting
  - Advice on imaging techniques, analysis possibilities
- Education
  - Workshops on Image Analysis
  - Courses / Training
  - Quantitative Big Imaging (http://bit.ly/1kj9mnq)

Acknowledgements

- AIT at PSI
- TOMCAT Group



We are interested in partnerships and collaborations

Learn more at

- 4Quant: From Images to Statistics - http://www.4quant.com (http://www.4quant.com)

- X-Ray Imaging Group at ETH Zurich - http://bit.ly/1gD8wKb (http://bit.ly/1gD8wKb)
- Quantitative Big Imaging Course at ETH Zurich - http://bit.ly/1kj9mnq (http://bit.ly/1kj9mnq)
- These slides - http://bit.ly/1xpbUSf (http://bit.ly/1xpbUSf)

Using Amazon's Cloud

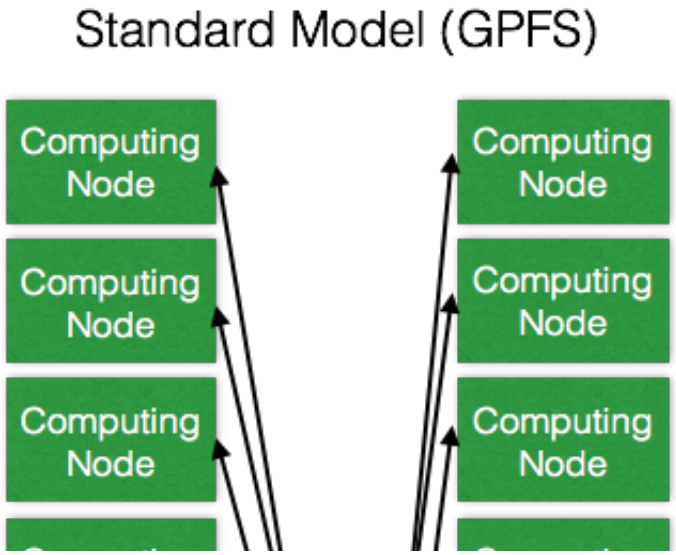
We have prepared an introduction to running Spark on the Amazon Cloud: http://4quant.github.io/aws-spark-introduction/ (http://4quant.github.io/aws-spark-introduction/)

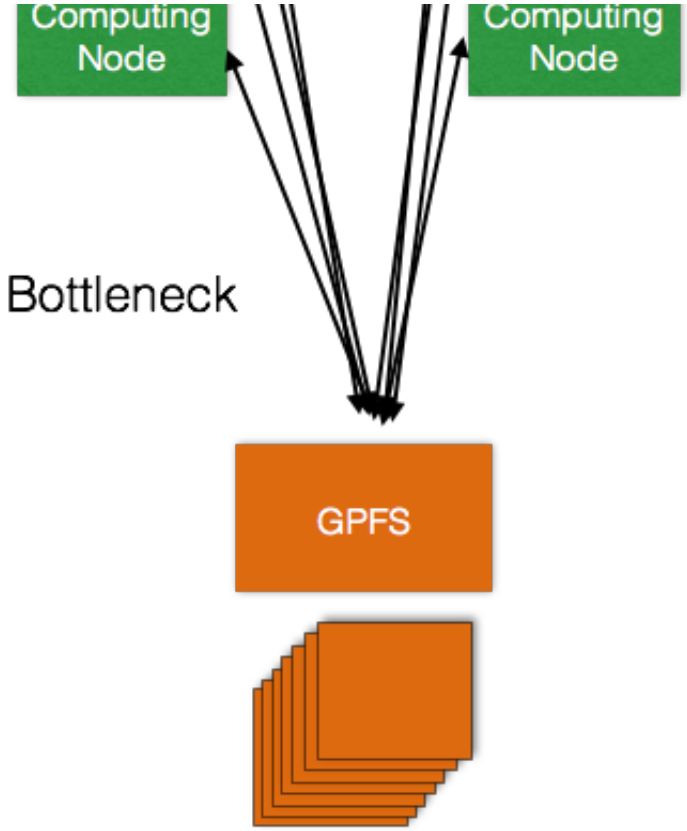
- Setup a cluster with 60 machines in Sydney

```
./spark-ec2 -k spark-key -i ~/Downloads/spark-key.pem -s 60 launch big-data-test-cluster --region=ap-southeast-2
```

Hadoop Filesystem (HDFS not HDF5)

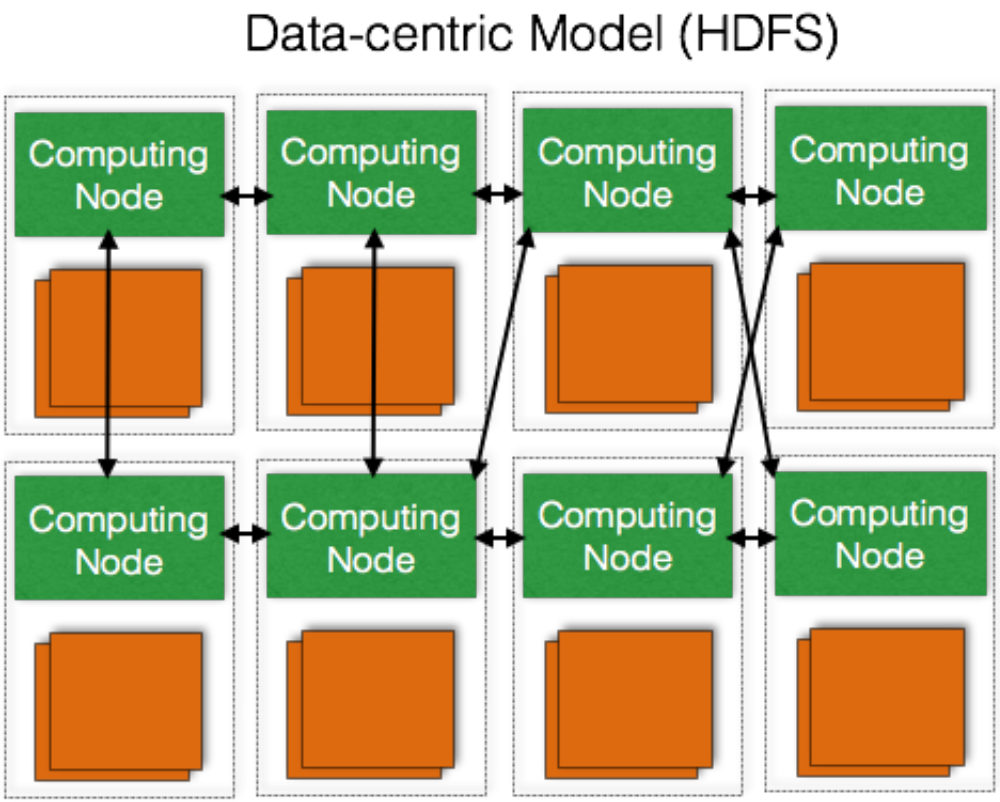
Bottleneck is filesystem connection, many nodes (10+) reading in parallel brings even GPFS-based fiber system to a crawl





One of the central tenants of MapReduce(tm) is data-centric computation → instead of data to computation, move the computation to the data.

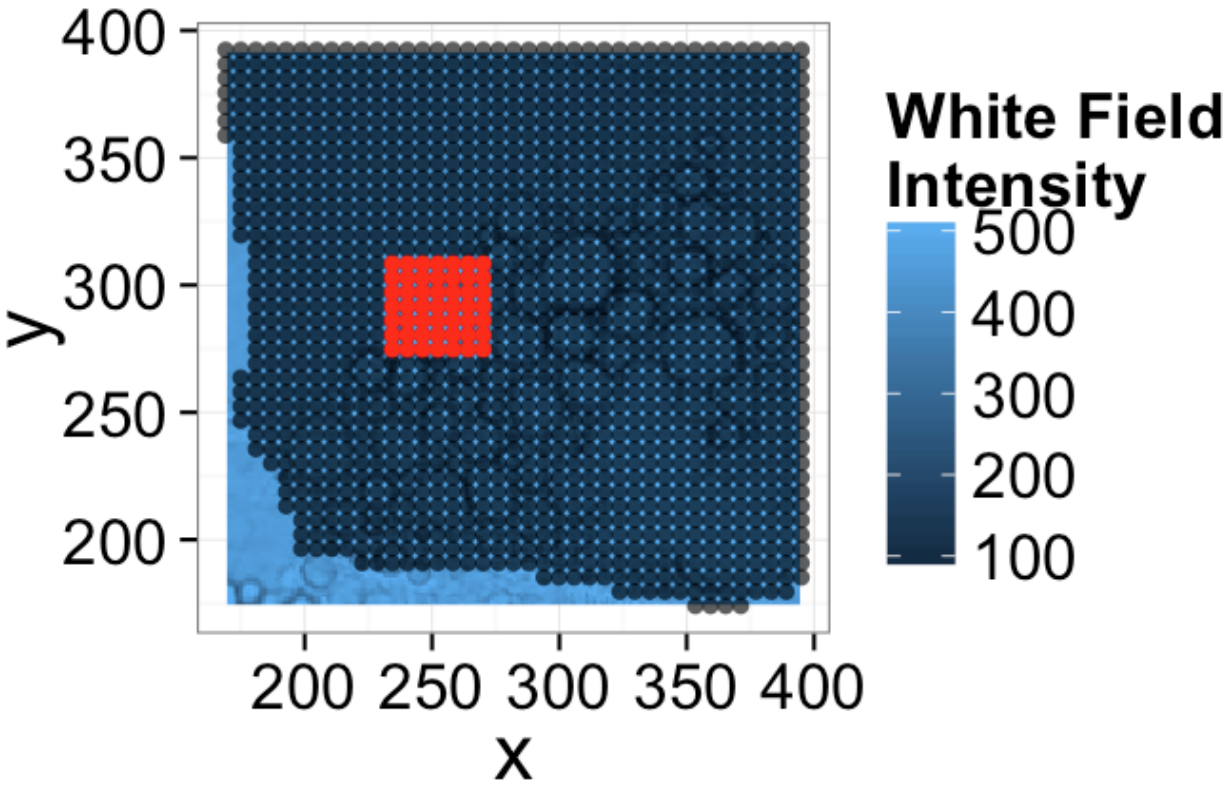
- Use fast local storage for storing everything redundantly → less transfer and fault-tolerance
- Largest file size: 512 yottabytes, Yahoo has 14 petabyte filesystem in use



Hyperspectral Imaging

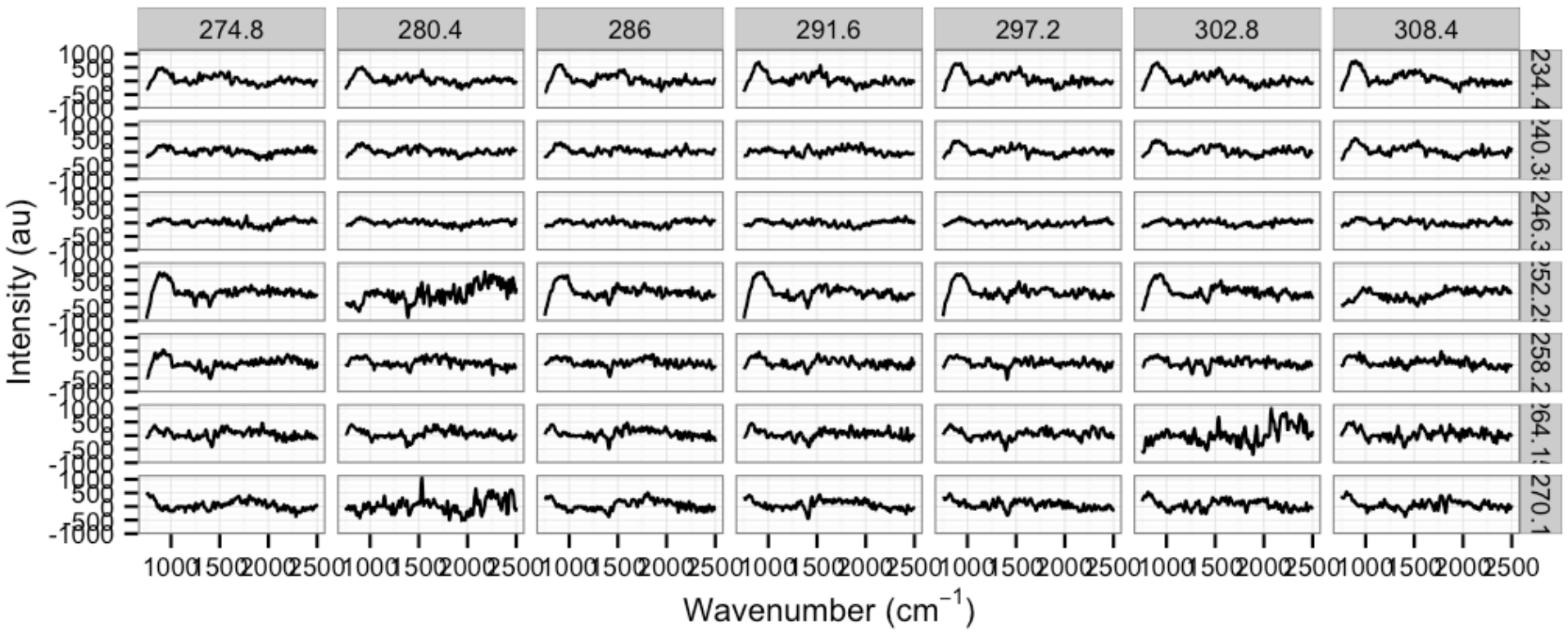
Hyperspectral imaging is a rapidly growing area with the potentially for massive datasets and a severe deficit of usuable tools.

+/- R Code



The scale of the data is large and standard image processing tools are ill-suited for handling them, although the ideas used in image processing are equally applicable to hyperspectral data (filtering, thresholding, segmentation,...) and distributed, parallel approaches make even more sense on such massive datasets

+/- R Code



+/- R Code

With SIL

Using SIL, Hyperspectral images are treated the same way as normal images and all of the operations you apply an a normal image are applicable, as long as the underyling operations are defined. A Gaussian filter might seem like a strange operation at first, but using information from each of a points neighbors to reduce noise makes sense.

- A Gaussian filter
  - Consists of addition, multiplication, and division
  - Define: operations as channel-wise
- K-Means Clustering
  - Similar points are grouped together
  - Define: a distance metric between two spectra

Identify cells from background by K-Means (2 groups) and then count the cells

```
clusters = KMeans.train(specImage,2)
labeledCells = specImage.map(clusters.identify(_)).
  makeLabels
print labeledCells.distinct.count+" cells"
```

K-Means Optimized (MPI/CUDA)

Taken from Serban's K-Means CUDA Project (https://github.com/serban/kmeans/blob/master/cuda_kmeans.cu (https://github.com/serban/kmeans/blob/master/cuda_kmeans.cu))

```
checkCuda(cudaMalloc(&deviceObjects, numObjs*numCoords*sizeof(float)));
checkCuda(cudaMalloc(&deviceClusters, numClusters*numCoords*sizeof(float)));
checkCuda(cudaMalloc(&deviceMembership, numObjs*sizeof(int)));
checkCuda(cudaMalloc(&deviceIntermediates, numReductionThreads*sizeof(unsigned int)));
checkCuda(cudaMemcpy(deviceObjects, dimObjects[0],
  numObjs*numCoords*sizeof(float), cudaMemcpyHostToDevice));
checkCuda(cudaMemcpy(deviceMembership, membership,
  numObjs*sizeof(int), cudaMemcpyHostToDevice));
do {
  checkCuda(cudaMemcpy(deviceClusters, dimClusters[0],
```

```
                numClusters*numCoords*sizeof(float), cudaMemcpyHostToDevice));
find_nearest_cluster
          <<< numClusterBlocks, numThreadsPerClusterBlock, clusterBlockSharedDataSize >>>
          (numCoords, numObjs, numClusters,
           deviceObjects, deviceClusters, deviceMembership, deviceIntermediates);
cudaDeviceSynchronize(); checkLastCudaError();
      compute_delta <<< 1, numReductionThreads, reductionBlockSharedDataSize >>>
          (deviceIntermediates, numClusterBlocks, numReductionThreads);
      cudaDeviceSynchronize(); checkLastCudaError();
      int d;
      checkCuda(cudaMemcpy(&d, deviceIntermediates,
                sizeof(int), cudaMemcpyDeviceToHost));
      delta = (float)d;
      checkCuda(cudaMemcpy(membership, deviceMembership,
                numObjs*sizeof(int), cudaMemcpyDeviceToHost));
      for (i=0; i<numObjs; i++) {
          /* find the array index of nestest cluster center */
          index = membership[i];

          /* update new cluster centers : sum of objects located within */
          newClusterSize[index]++;
          for (j=0; j<numCoords; j++)
              newClusters[j][index] += objects[i][j];
      }
```

K-Means Declarative

1. Starting list of points: **Points**
2. Take 4 random points from the list → **Centers**
    - Apply a *filter* operation with the function `(random()<nPoints/4)`
3. Create a function called nearest center which takes a point $\vec{x}$ and returns the nearest center to the point and the point itself

```
nearestCenter(x) = {
  for cCenter in Centers
    pointDist(cCenter) = dist(x,cCenter)
  end
  return (argmin(pointDist),x)
}
```

1. *Map* `nearestCenter` onto **Points** as **NearestCenterList**

Getting an image to Key-Value Format

+/- R Code

x	y	val
1	1	0.6274510
2	1	0.7803922
3	1	0.8862745
4	1	0.8980392
5	1	0.9098039
6	1	0.9215686

The key is position $\langle x, y \rangle$ and value is the intensity val

Loading the data into Spark (Scala)

```
val rawImage=sc.textFile("cell_colony.csv")
val imgAsColumns=rawImage.map(_.split(","))
val imgAsKV=imgAsColumns.map(point => ((point(0).toInt,point(1).toInt),point(2).toDouble))
```

- Count the number of pixels

```
imgAsKV.count
```

- Get the first value

```
imgAsKV.take(1)
```

- Sample 100 values from the data

```
imgAsKV.sample(true,0.1,0).collect
```

Perform a threshold

```
val threshVal=0.5
val labelImg=imgAsKV.filter(_._2<threshVal)
```

- Runs on 1 core on your laptop or 1000 cores in the cloud or on a local cluster resource.
- If one computer crashes or disconnects it **automatically** continues on another one.
- If one part of the computation is taking too long it will be sent to other computers to finish
- If a computer runs out of memory it writes the remaining results to disk and continues running (graceful dropoff in performance)

Get Volume Fraction

```
100.0*labelImg.count/(imgAsKV.count)
```

Region of Interest

Take a region of interest between 0 and 100 in X and Y

```
def roiFun(pvec: ((Int,Int),Double)) =
 {pvec._1._1>=0 & pvec._1._1<100 & // X
  pvec._1._2>=0 & pvec._1._2<100 } //Y
val roiImg=imgAsKV.filter(roiFun)
```

Perform a 3x3 box filter

```
def spread_voxels(pvec: ((Int,Int),Double), windSize: Int = 1) = {
  val wind=(-windSize to windSize)
  val pos=pvec._1
  val scalevalue=pvec._2/(wind.length*wind.length)
  for(x<-wind; y<-wind)
    yield ((pos._1+x,pos._2+y),scalevalue)
}

val filtImg=roiImg.
      flatMap(cvec => spread_voxels(cvec)).
      filter(roiFun).reduceByKey(_ + _)
```

Setting up Component Labeling

- Create the first labels from a thresheld image as a mutable type

```
val xWidth=100
var newLabels=labelImg.map(pvec => (pvec._1,(pvec._1._1.toLong*xWidth+pvec._1._2+1,true)))
```

- Spreading to Neighbor Function

```
def spread_voxels(pvec: ((Int,Int),(Long,Boolean)), windSize: Int = 1) = {
  val wind=(-windSize to windSize)
  val pos=pvec._1
  val label=pvec._2._1
  for(x<-wind; y<-wind)
    yield ((pos._1+x,pos._2+y),(label,(x==0 & y==0)))
}
```

Running Component Labeling

```
var groupList=Array((0L,0))
var running=true
var iterations=0
while (running) {
  newLabels=newLabels.
  flatMap(spread_voxels(_,1)).
    reduceByKey((a,b) => ((math.min(a._1,b._1),a._2 | b._2))).
    filter(_._2._2)
  // make a list of each label and how many voxels are in it
  val curGroupList=newLabels.map(pvec => (pvec._2._1,1)).
    reduceByKey(_ + _).sortByKey(true).collect
  // if the list isn't the same as before, continue running since we need to wait for swaps to stop
  running = (curGroupList.deep!=groupList.deep)
  groupList=curGroupList
  iterations+=1
  print("Iter #"+iterations+":"+groupList.mkString(","))
}
groupList
```

Calculating From Images

- Average Voxel Count

```
val labelSize = newLabels.
  map(pvec => (pvec._2._1,1)).
  reduceByKey((a,b) => (a+b)).
  map(_._2)
labelSize.reduce((a,b) => (a+b))*1.0/labelSize.count
```

- Center of Volume for Each Label

```
val labelPositions = newLabels.
  map(pvec => (pvec._2._1,pvec._1)).
  groupBy(_._1)
def posAvg(pvec: Seq[(Long,(Int,Int))]): (Double,Double) = {
val sumPt=pvec.map(_._2).reduce((a,b) => (a._1+b._1,a._2+b._2))
(sumPt._1*1.0/pvec.length,sumPt._2*1.0/pvec.length)
}
print(labelPositions.map(pvec=>posAvg(pvec._2)).mkString(","))
```

Lazy evaluation

- No execution starts until you save the file or require output
- Spark automatically deconstructs the pipeline and optimizes the jobs to run so computation is not wasted outside of the region of interest (even though we did it last)

Applying Machine Learning to Imaging

The needs in advertising, marketing, and recommendation engines have powered the creation of a many new machine learning tools capable of processing huge volumes of data. One of the most actively developed projects, MLLib, is a module of Spark and can be directly combined with the *Imaging Layer*