Blue Brain Project

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Accelerating the Pipeline of Brain Tissue Simulations with Apache Spark

Fernando Pereira, Judit Planas, Matthias Wolf

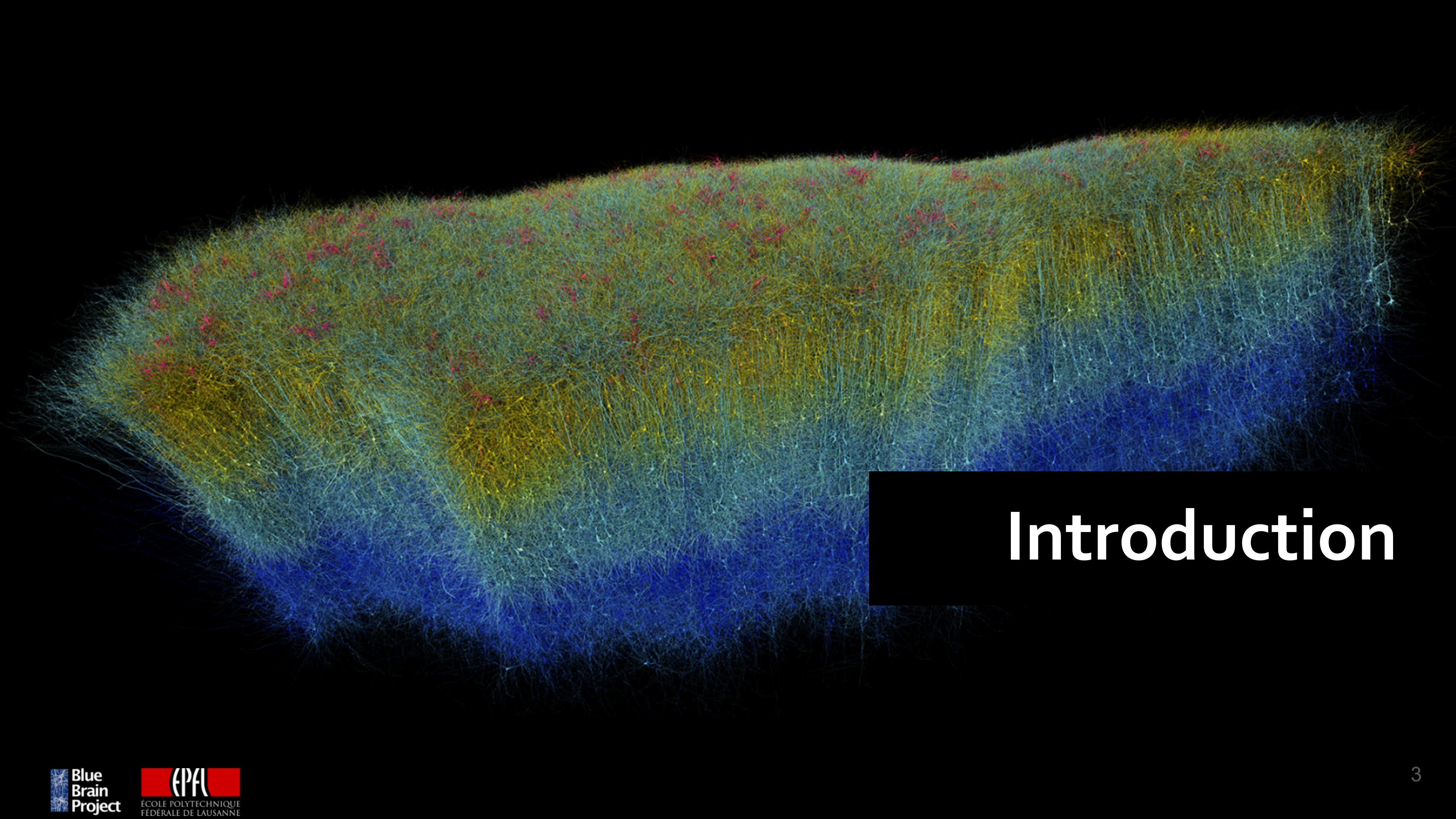
Blue Brain Project, EPFL

Outline

- Introduction
- Motivation
- Circuit building process
    - Functionalizer filtering
    - Partitioning
    - Reproducibility
    - Running at scale
- Simulation output analysis
    - Workflow & data structure
    - Evaluation
- Conclusions

Blue Brain Project

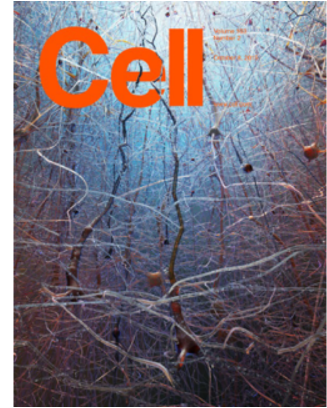
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE


Introduction
Blue Brain Project
EPFL
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

The Blue Brain Project

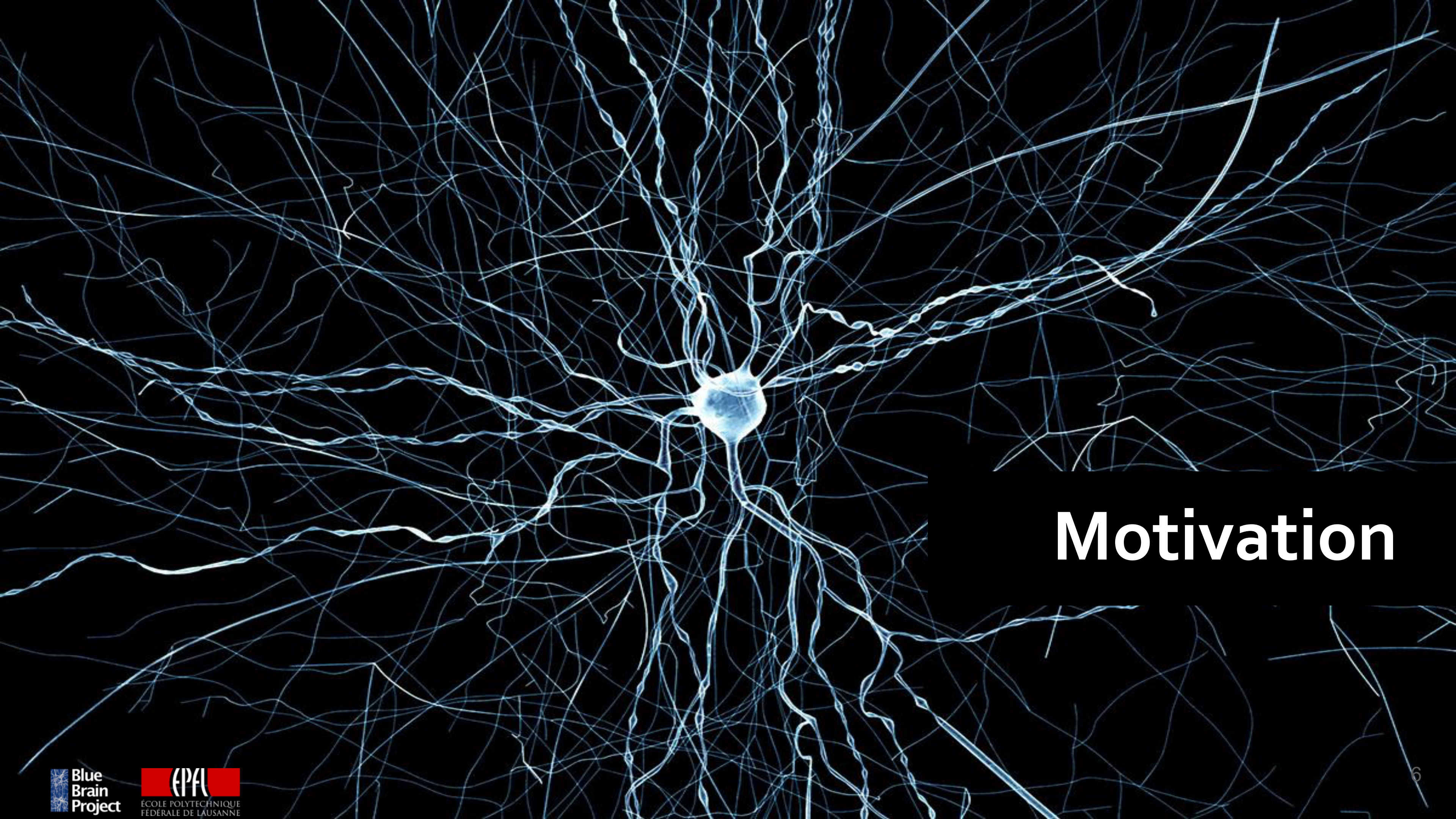
- The Blue Brain Project (BBP) is a Swiss initiative that targets the **reconstruction and simulation of the brain**, hosted in Geneva
- BBP is a **multidisciplinary team** that brings together people from a wide variety of backgrounds, like neuroscience, computer engineering, physics, maths or chemists (~120 people and growing!)

BBP’s Key Target Contributions to Neuroscience

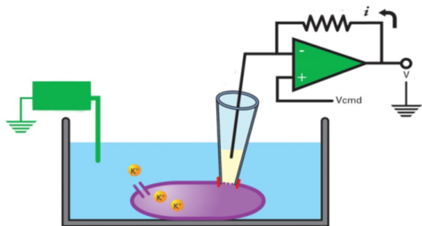


- **Help scientists understand how the brain functions internally**
- BBP has been able to reproduce the electrical behavior of a neocortex fragment by means of a computer [1]
  - Revealed novel insights into the functioning of the neocortex
- Supercomputer-based simulations of the brain:
  - Enables experiments that are impossible in a laboratory
- Understanding the brain can contribute in different fields: understanding of brain diseases, neurobotics, neuromorphic computing, AI, …

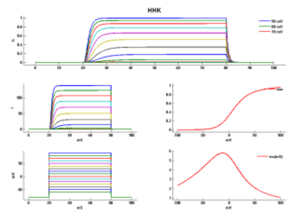
[1] Markram *et al*. **Reconstruction and Simulation of Neocortical Microcircuitry**. Cell, Vol. 163, Issue 2, pp 456 - 492


Motivation
Blue
Brain
Project
EPFL
ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

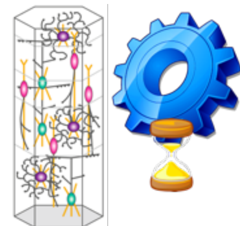
Brain Tissue Simulation Pipeline



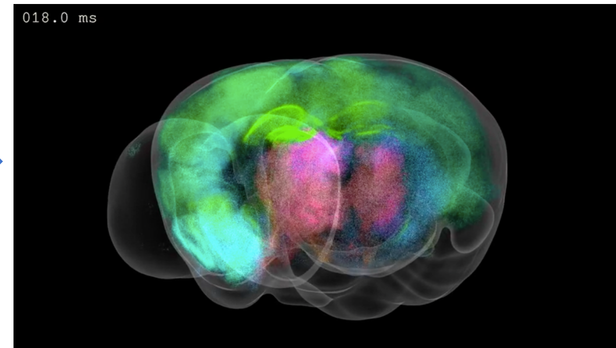
In vivo / in vitro experiments



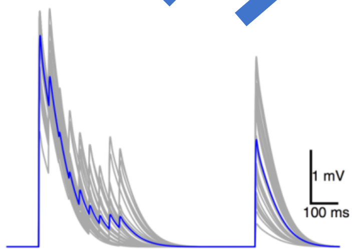
Model creation



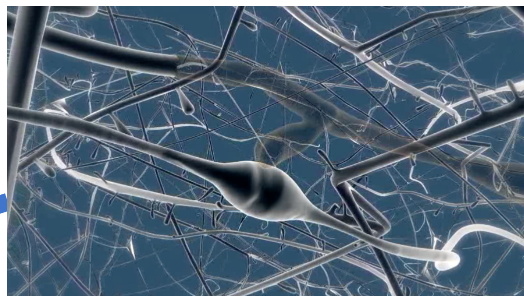
Circuit building



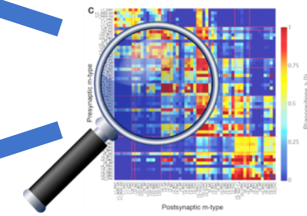
Simulation
(*in sillico* experiments)



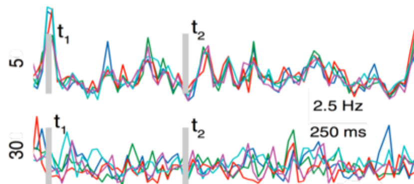
Model validation



Visualization



Data analysis



Scripting analysis

Blue Brain Project

EPFL ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Larger Circuits, Larger Data

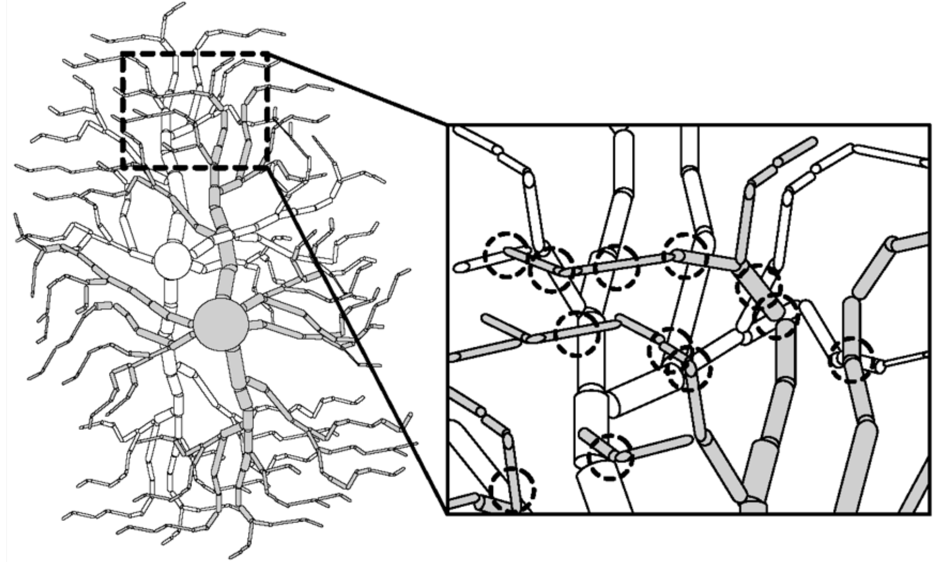
- Brain tissue simulations need & produce **massive amounts of data** very quickly
  - We cannot simulate the whole human brain with existing supercomputers
  - BBP focuses on the simulation of rodent brain regions
- Scientists use sequential scripts to build & analyze simulations
  - Python is preferred
  - They do not have time / expertise to improve their scripts
  - We are reaching the computational limit of our existing tools → **need for scalable solutions**
- Example: Plastic neocortex simulation
  - 31.000 neurons, 30 s biological time
  - Output: **~50 GB per recorded variable [x N]**

Circuit Building Process

Circuit Building Process

Touch Detector

- Which sections of axons/dendrites overlap?



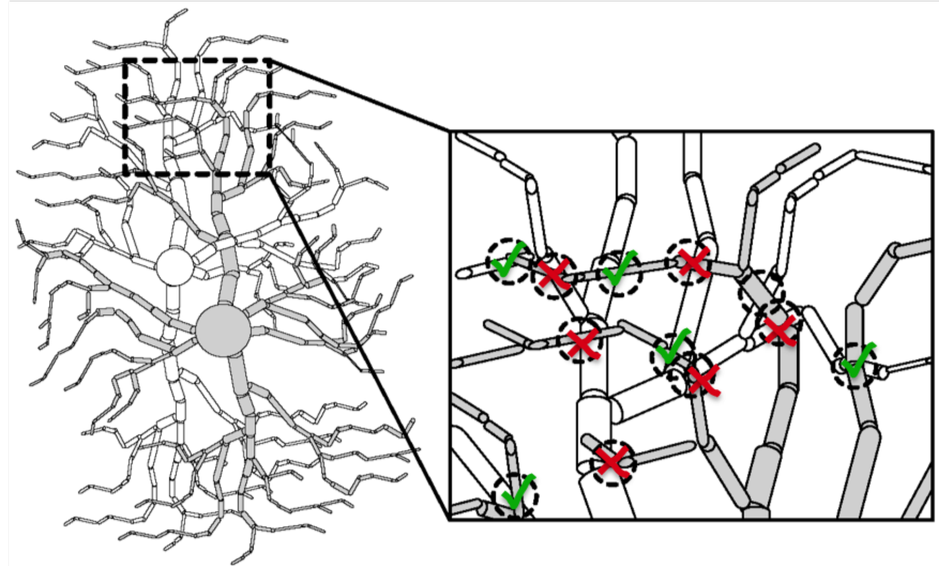
Circuit Building Process

Touch Detector

- Which sections of axons/dendrites overlap?

Functionalizer

- Which touches are biologically valid?
  - Filter out those which don’t fit
- What are the likely parameters?
  - Conductance, depression & facilitation time…
  - Sample from given distributions



11 M cells ~> 500 B touches

Blue Brain Project
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Functionalizer Filters

Soma-axon distance
Touch rules

Reduce

Cut

Deterministic filters

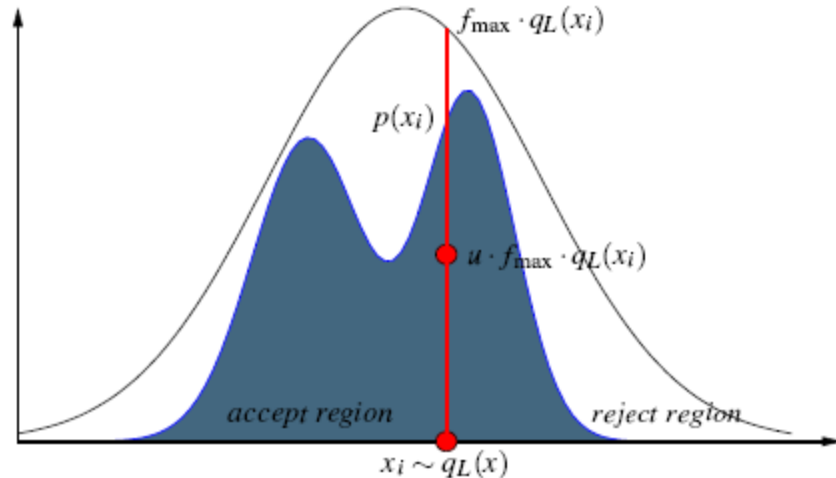
SQL filter expression applied to every touch

Limitation: Performance degrades as the number of rules increase

Improvement: Precalculate and broadcast a filtering table. Then merge by index lookup

Probabilistic filters

Same principle as Rejection Sampling



Blue Brain Project
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Functionalizer Filters

Soma-axon distance
Touch rules

Reduce

Cut

Deterministic filters

SQL filter expression applied to every touch

Limitation: Performance degrades as the number of rules increase

Improvement: Precalculate and broadcast a filtering table. Then merge by index lookup

Reduce:

1. Compute histogram
2. Filter accordingly

Cut:

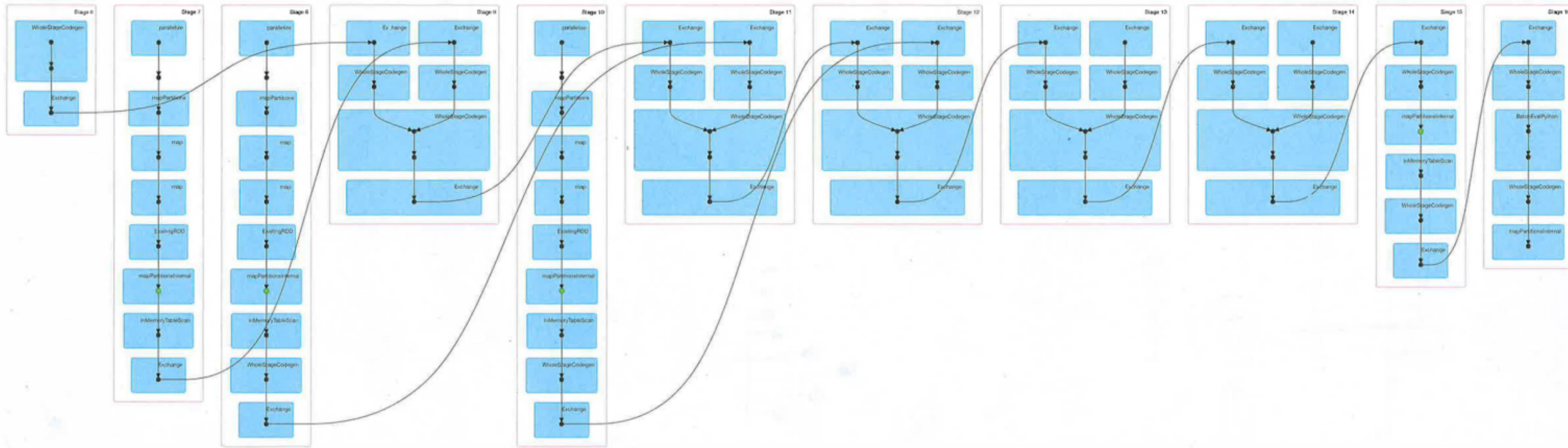
1. Compute histogram
2. Validity checks
3. Filter accordingly
4. Re-compute histogram
5. Validity checks
6. Filter accordingly

- Huge, impracticable, execution plans
- Lots of shuffles
- Sub-optimal partition sizes

Blue Brain Project

EPFL ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Really, Lots of Shuffles!



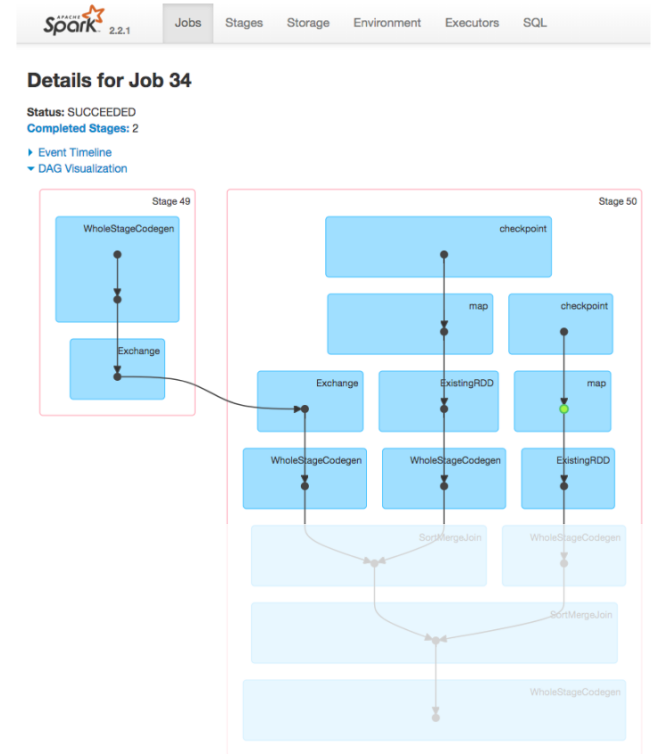
Breaking Execution Plans

➔ Hard time analyzing complex execution plans?

➔ Expensive calculations may end up being computed twice?

Consider checkpointing intermediate results to DISK

- Shuffles are written to disk anyway!
- Break the plan in strategic points
  - Invaluable for analysis (& execution!)
  - Reuse them whenever possible
- Stack computations to hide I/O time
  - Avoid shuffles → Keep partitioning!



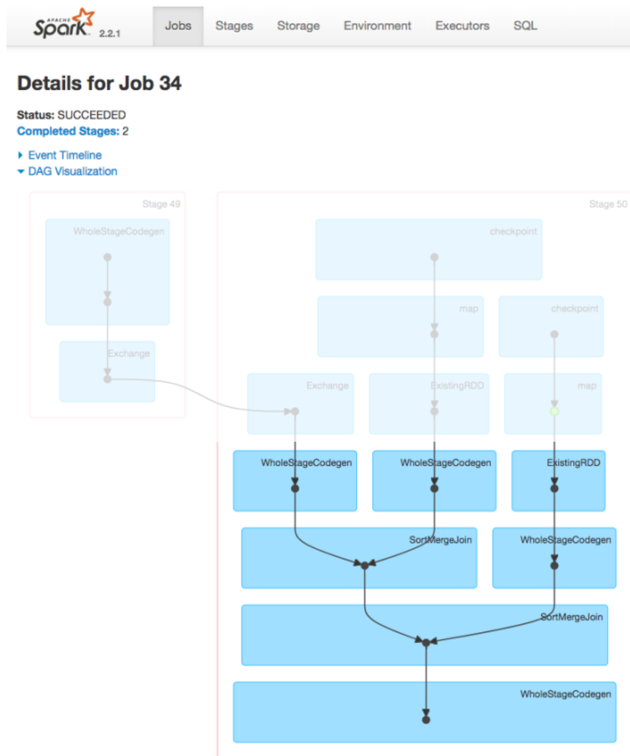
Blue Brain Project

EPFL ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Breaking Execution Plans

- BroadcastJoin [if one DF is small enough]
  - `"spark.sql.autoBroadcastJoinThreshold"` (def: 10MB)
  - Manually activate with `.broadcast()`

- SortMergeJoin
  - Requires the same exact partitioning:
    - **Columns** & **Number** of partitions
  - Otherwise does it for you → shuffle



Storing to disk with Partitioning:

- Use `saveAsTable()` or `localCheckpoint()` (sorry parquet!)

```
df.write.mode("overwrite")._jwrite
  .bucketBy(num_partitions, col1, _to_seq(sc, other_cols))
  .sortBy(col1, _to_seq(sc, other_cols))
  .saveAsTable(table_name))
```

Blue Brain Project
EPFL
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

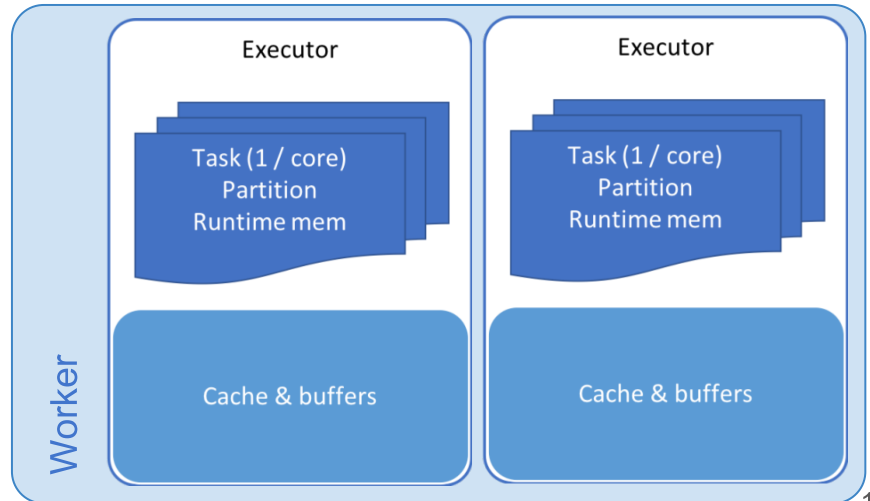
Partition Size Control

Avoid OOM

- Some operations (UDFs, sort) take N times the partition size in memory
- Executors heap mem < `64 GB`, high thread (task) number
  - `N_CPU x` **`Partition_Size`** `x Margin x 2 < 64 GB`
- Our configuration
  - Cap partitions to 256 MB
  - 2x 18-core executors per worker node

Spark features:

- `repartition(numPartitions, *cols)` ??



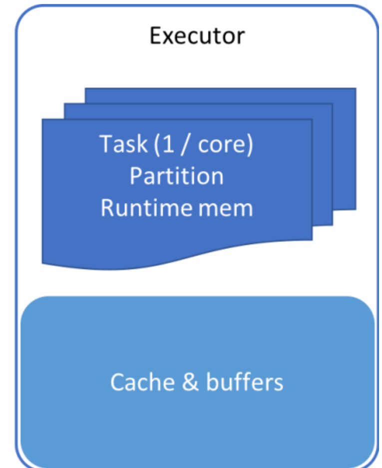
Blue Brain Project

EPFL ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Partition Size Control

Spark features:

- repartition(numPartitions, *cols) ⚠️ → shuffle, will change partitioning scheme → shuffle
    - `coalesce()`
    - `df.rdd.getNumPartitions()`
- Configuration entries:
    - Reading: `"spark.sql.files.maxPartitionBytes"`
    - Writing: `hadoopConfiguration().setInt("parquet.block.size",)`
    - Shuffles: `"spark.sql.shuffle.partitions"`



Blue Brain Project
EPFL ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Reproducibility & Scalability

Blue Brain Project
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Scientific Requirement: Reproducibility

We need random numbers to

- Sample touches to match statistical distributions
    - Using a “survival” probability depending on touch categorization
- Generate synapse properties from parameterized distributions
    - Following Poisson, Gamma, and Normal distributions
- Scientist want to have results reproducible on a binary level
    - Precludes relying on statistical equivalence
    - Need to use specific seeds in calculations

Scientific Requirement: Reproducibility

Seeding in the Spark documentation:

- `pyspark.sql.DataFrame.sampleBy(col, fractions, seed=None)`
- `pyspark.sql.functions.rand(seed=None)`

Discovered drawbacks:

- The current implementation in Spark itself uses the seed to re-seed per partition
- Introduces a **dependency on number of partitions**
- **Variance in number of cores used influences partitions**

No guaranteed reproducibility!

Blue Brain Project

EPFL ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Adding RNG via Java / PandasUDFs + Cython

Possible approach:

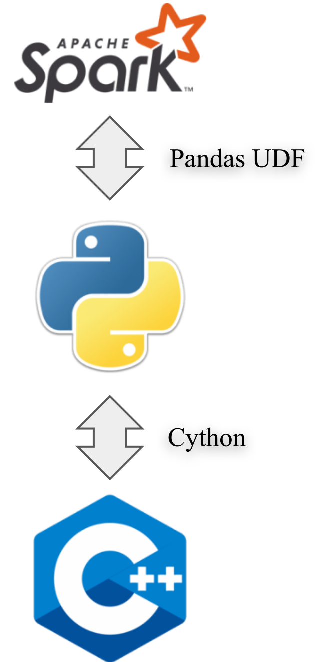
- Java UDF class, applied per row
- But Java RNG stack is very barebones, JNI integration difficult

Other software (libraries) in our stack:

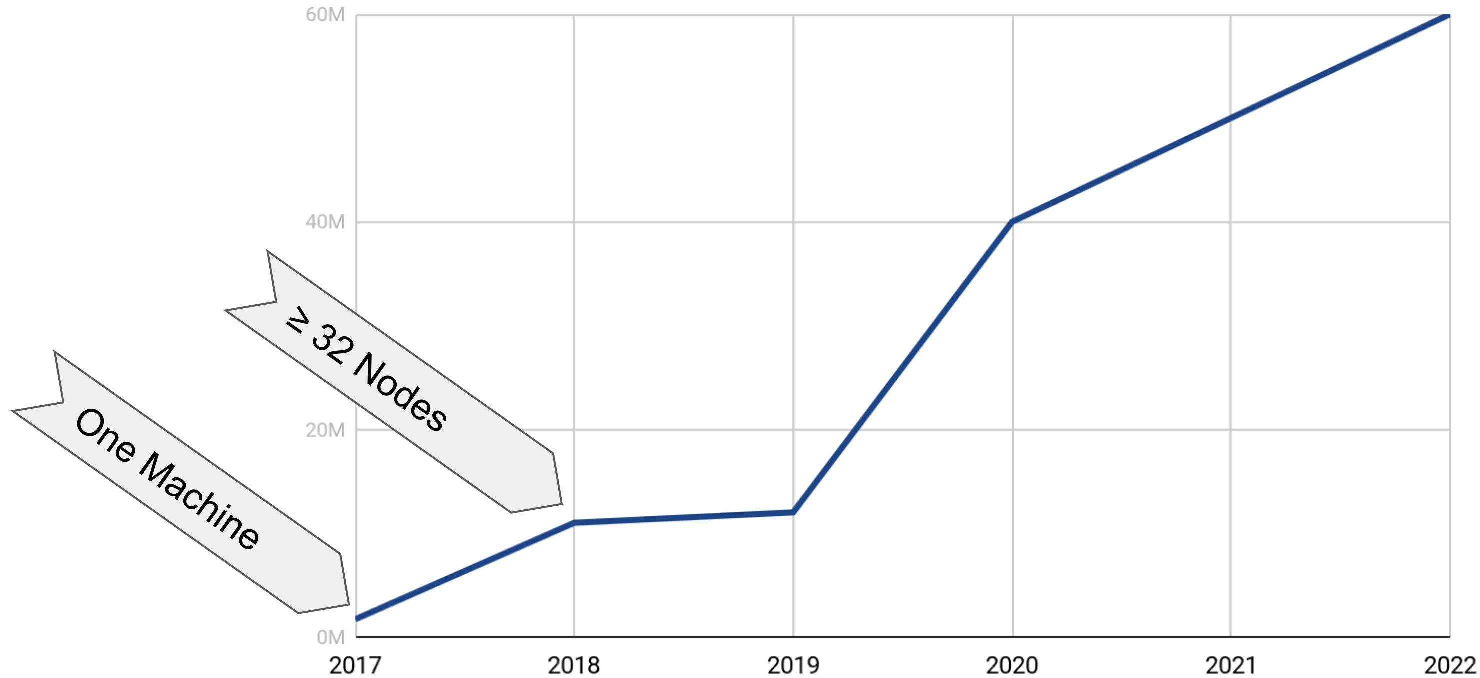
- Use Random123-based library (Threefry)
- Counter-based with keys, derivation creates new unique keys

How do we connect this with Spark?

- Easy to integrate with Cython into Python
- Call into Cython from columnar PandasUDF



Scientific Roadmap: Circuit Sizes



Blue Brain Project

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Interfacing with an HPC Cluster

Our environment:

- Spack / Nix for deployment and distribution (C libraries, Python packages)
- **No permanent or dedicated Spark cluster**
  - Use a shared HPC supercomputer (BlueBrain5)
  - Shared parallel file system: GPFS
  - 80 nodes with local disks: 2 TB NVME SSD per node
  - Launch software via SLURM

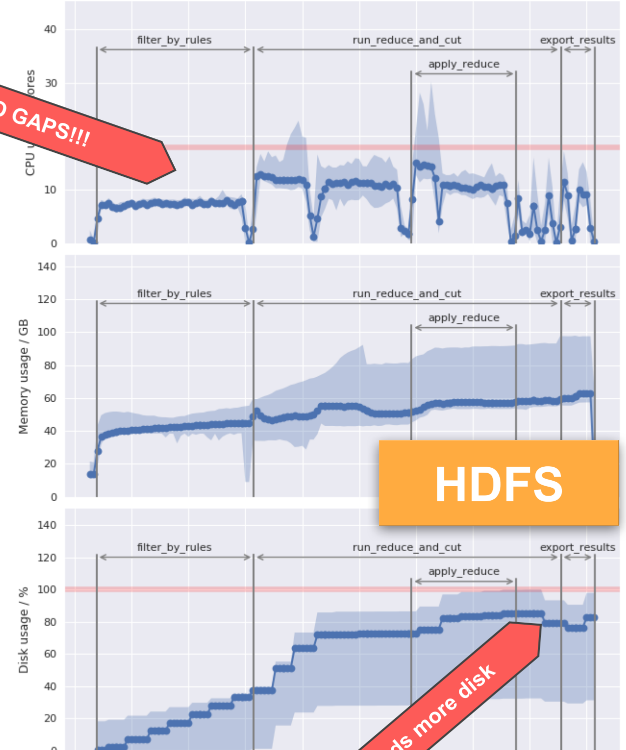
Custom script for spawning a **temporary Spark** cluster

- Sets up local directory structure and launch Spark master, workers
- Launches PySpark application and tears down cluster when done

Spark Performance

After debugging:

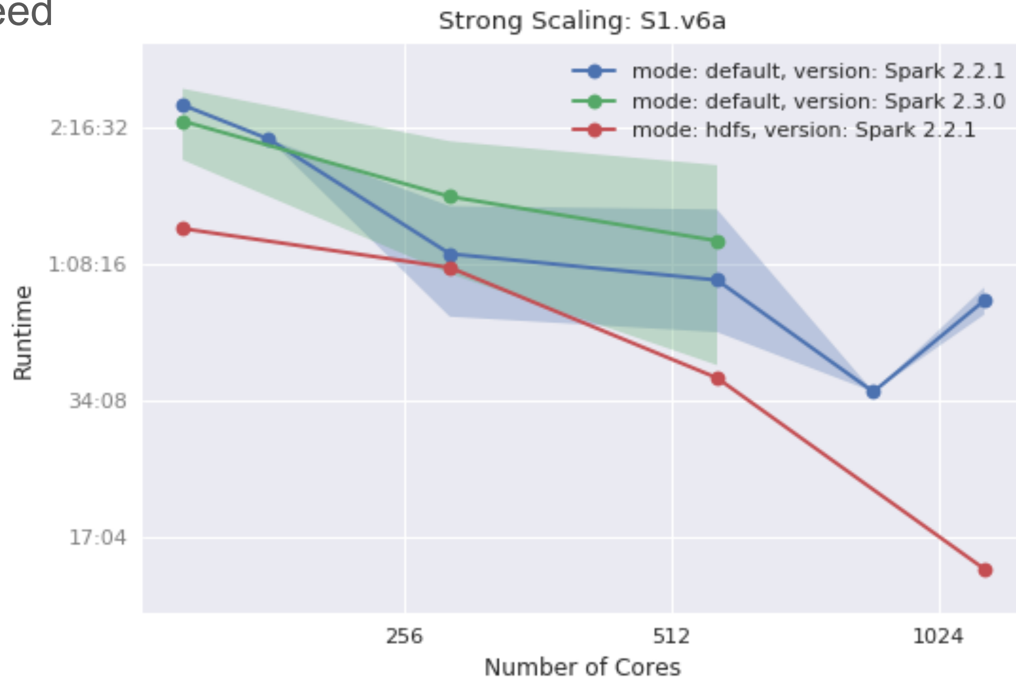
- File system interactions very slow
- Non-linear increase in file count
- GPFS performance worsens



Project
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Why HDFS: Strong Scaling @ 1.7 M Neurons

- HDFS increases turnaround speed

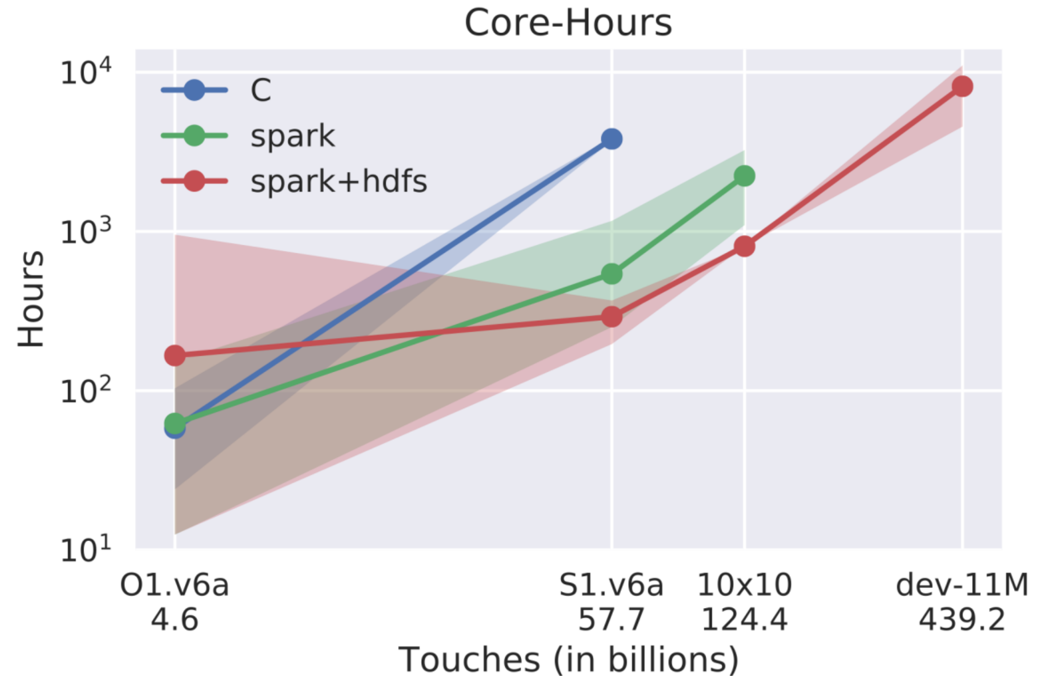


Blue Brain Project

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Why HDFS: Weak Scaling

- C++ implementation: OOM
- GPFS limits runtime
- HDFS limits scale

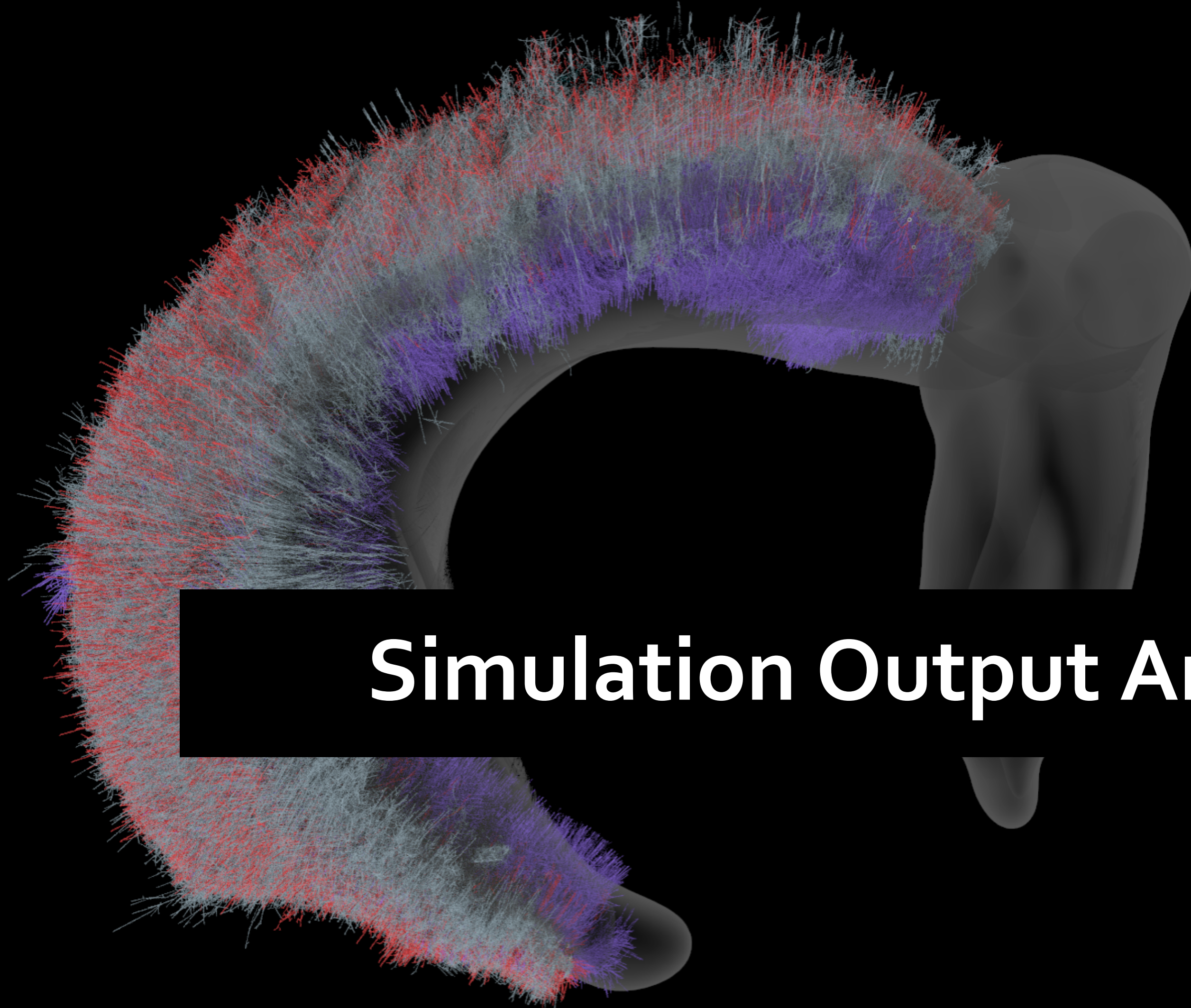


Blue Brain Project

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Lessons Learned

- Successful transition from C++ implementation to Spark
- Execution plans grow unreasonably and need to be broken up
- Partitions sizes should be tuned for optimal execution
- Built-in RNG support is lackluster
- Parallel file systems become unusable very fast



Simulation Output Analysis

Blue Brain Project

EPFL
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

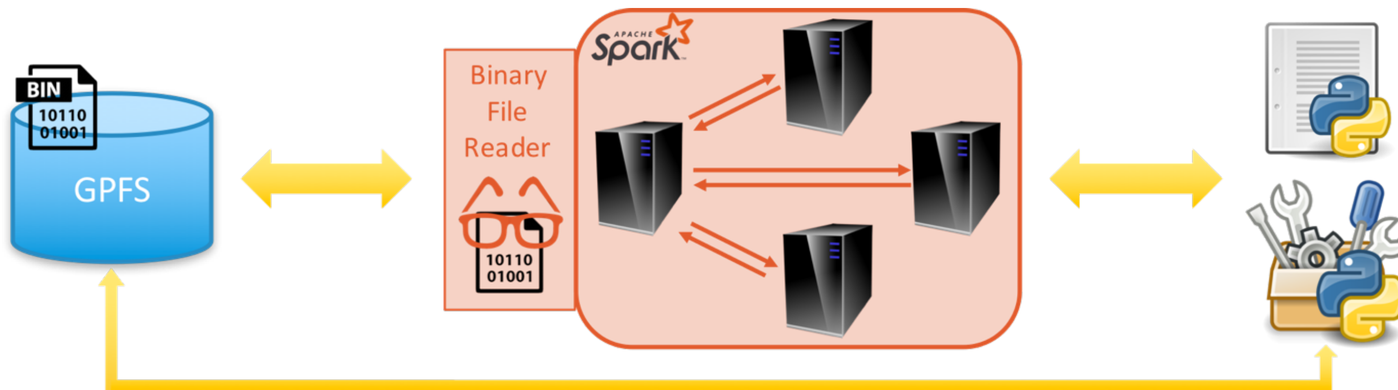
Data Analysis Workflow with Spark

Pros

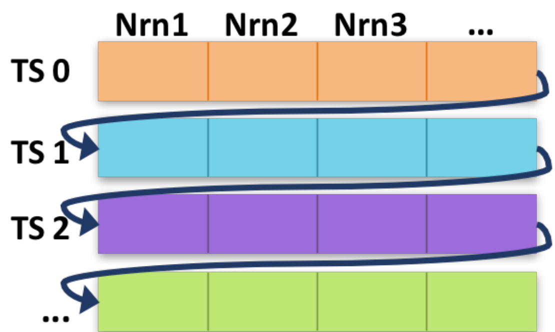
- Python support
- Scalable (cluster)
- Can be hidden from the final user
- Fits our type of analysis
- Compatible with on-site system

Cons

- NumPy support is critical:
  - RDD: OK
  - DataFrame: needs data conversion
- Missing native reader for custom formats
- UDFs in Python add overhead
- On-site GPFS can't be migrated to HDFS



Current Output Layout vs [Spark] Key/Value Layout



Nrn	TS	Src	Data
1	0		
1	1		
1	2		
1	...		
2	0		
2	1		
2	2		
2	...		
...	0		
...	1		
...	2		
...	...		

- Matrix-like organization
- Written by rows (time steps) → good performance
- Multiple read patterns: by rows or columns or randomly… → cannot optimize for each use case

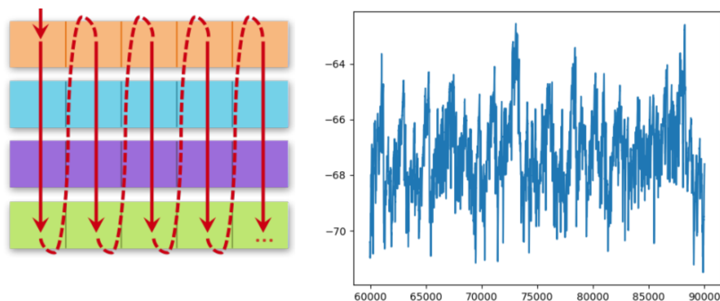
- Key/value layout breaks major ordering
  - Key: Nrn + TS pair
- Open design to
  - Add/remove columns at convenience
  - Hold arbitrary data
- Faster (non-sequential) access time

Spark Data Analysis Evaluation

- In order to evaluate the Spark data analysis framework, we carefully choose two real use cases, with opposite data access patterns:

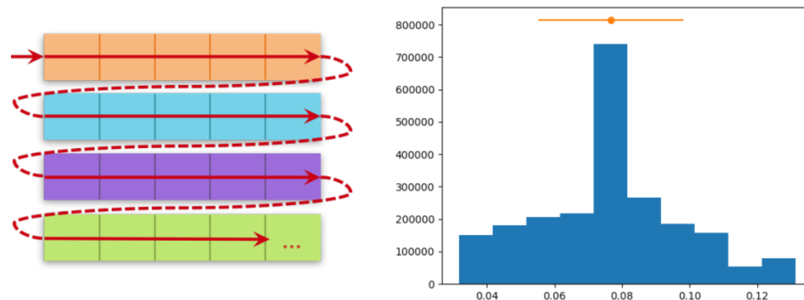
Computation by GID (neuron ID)

- Mean values per neuron over time
- Column-major access



Computation by TS (time step)

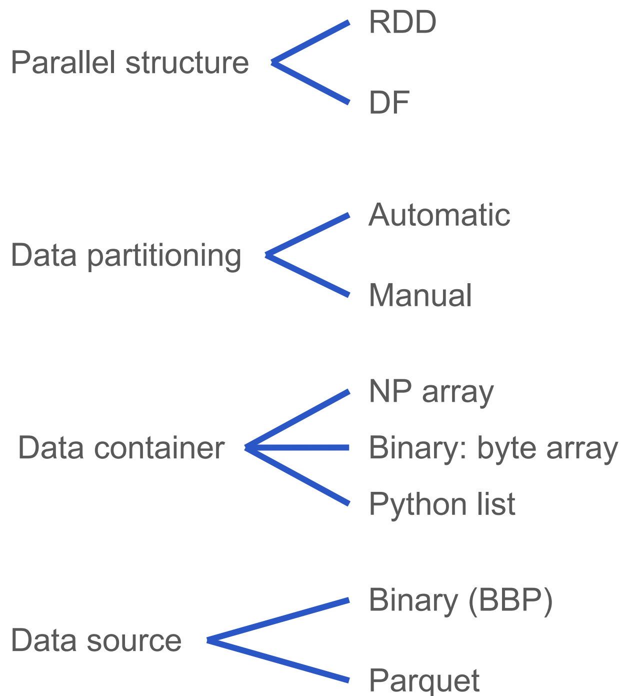
- Histogram of neuron values per time step
- Row-major access



Blue Brain Project

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Configuration Comparison



Plot labels of selected combinations

	RDD		RDDKey		DFbin		DFpylist
Parallel structure	RDD		RDD		DF		DF
Data partitioning	Auto		Manual (GID)		Auto		Auto
Data container	NP array		NP array		Byte array		Python list
Data source	BBP	Parquet	BBP	Parquet	BBP	Parquet	BBP

Evaluation Platform

Hardware

On-site supercomputer

- 80 compute nodes (Skylake)
  - 2 x Intel Xeon 6140
  - 2 x 18 cores (72 threads with HT)
  - 384 GB DRAM
  - 2 x SSD P4500, 1 TB each
- 120 compute nodes (KNL)
  - Intel KNL 7230-tPRQ
  - 64 cores (256 threads with HT)
  - 96 GB DRAM + 16 GB MCDRAM
- Infiniband EDR 100 GB
- GPFS file system

Software

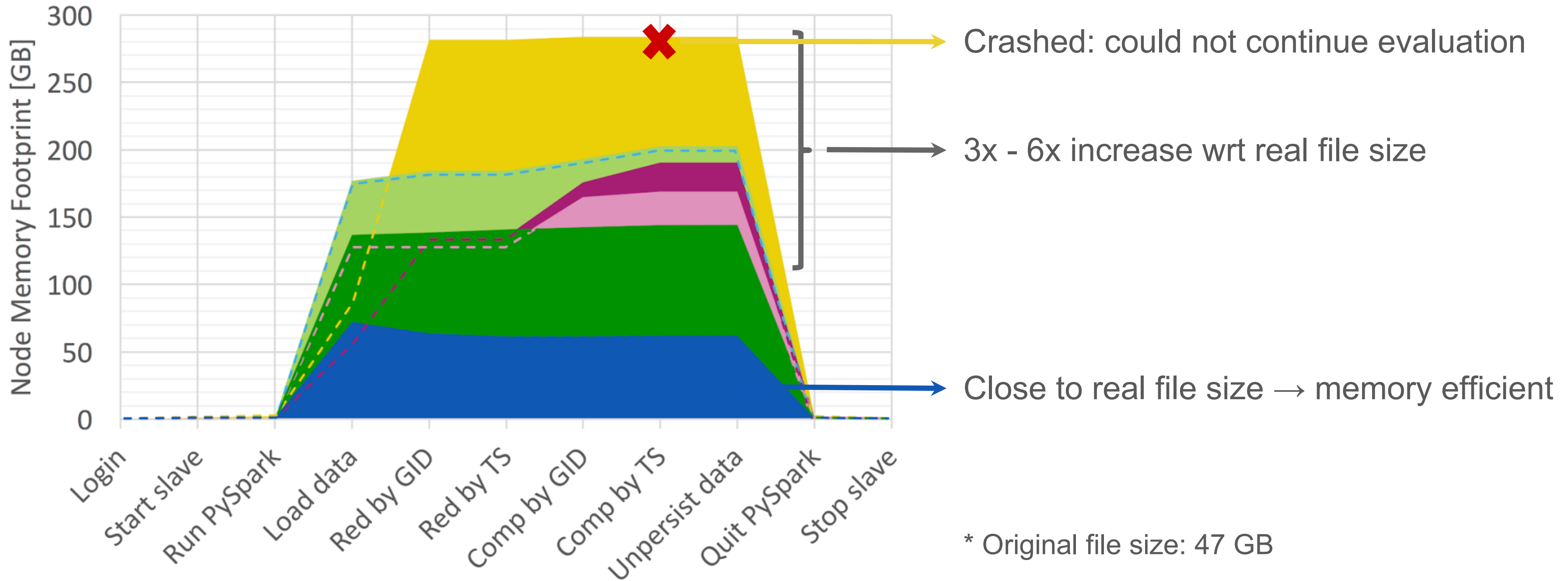
- Red Hat Enterprise Linux 7.3
- Java OpenJDK RE 1.8
- Apache Spark 2.2.1

Runtime Configuration *

- Exclusive access to allocated nodes
- Spark slaves use all cores
- Spark master runs on separate node
- Dataset size: 2 TB

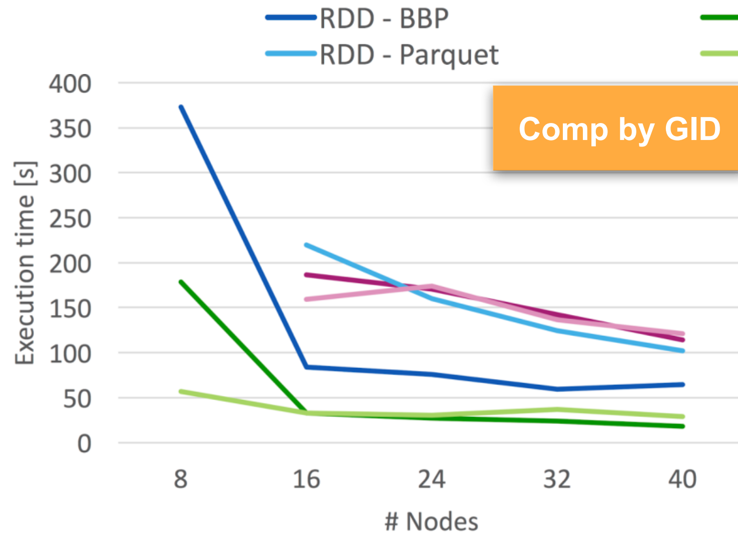
* *Unless specified differently*

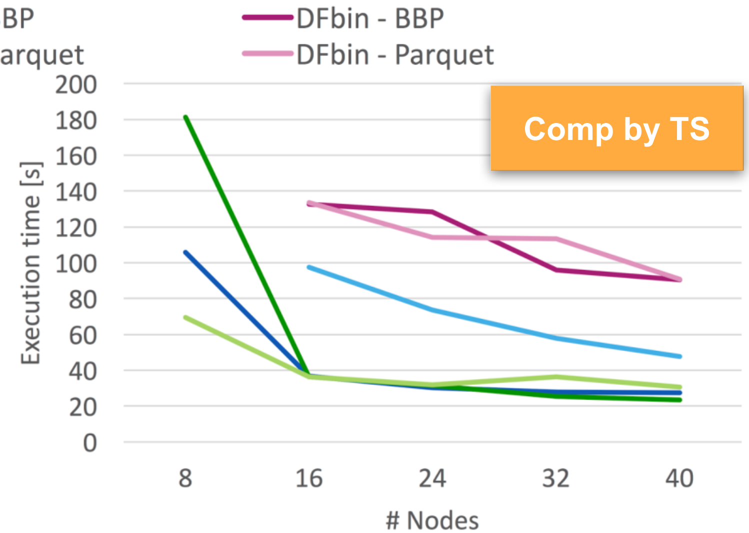
Memory Footprint



* Original file size: 47 GB

Computation by GID / TS Performance





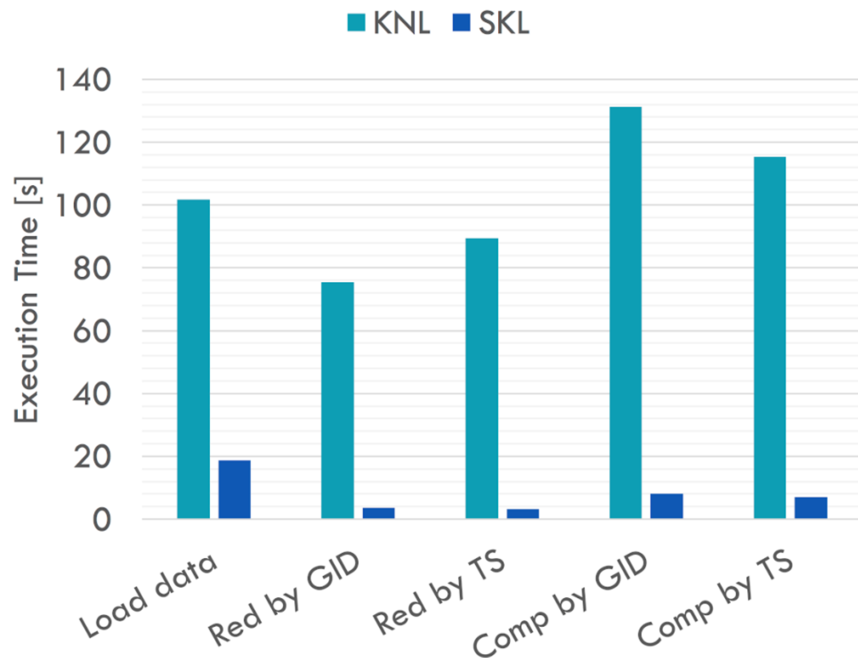
- RDDKey fastest thanks to data partitioning
- DFbin + RDD-Parquet slower due to data conversions (binary → NP array)

- Comp by TS run after Comp by GID: some partial results cached → faster
- Significant speed-up from 8 to 16 nodes

Blue Brain Project

EPFL ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Spark Performance: KNL vs Skylake



Execution configuration:

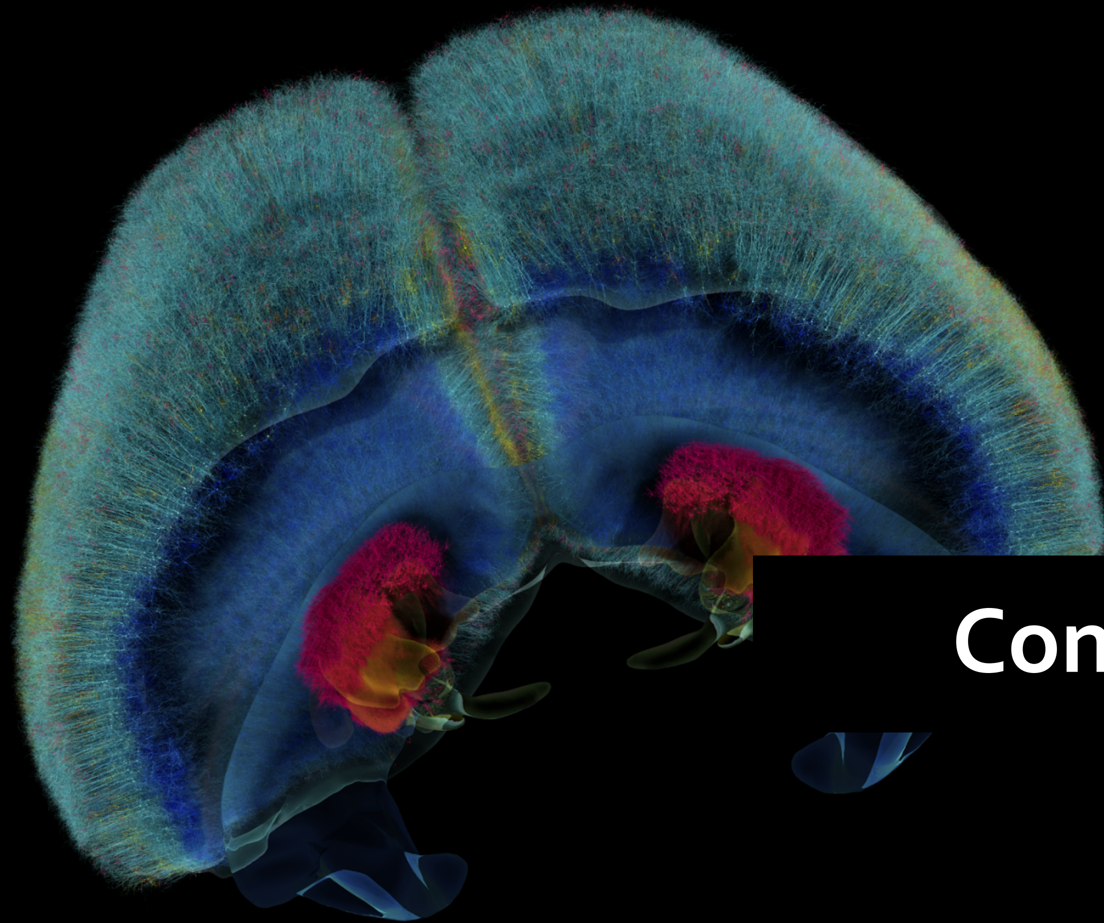
- RDD-BBP only
- 1 simulation report (47 GB)
- 1 + 8 Spark worker nodes
- Spark worker cores:
  - KNL: 16
  - SKL: 36

* Tried > 1 report @ KNL → crash
* Tried > 16 worker cores @ KNL → crash
Main problem: out of memory

Blue Brain Project
EPFL ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Lessons Learned

- Type of data containers impacts the memory footprint
- Appropriate source data format leads to better I/O
- Manual partitioning can increase analysis performance, at the cost of longer loading time
- Data conversions [obviously] add overhead
- Spark benefits from fast storage (shuffling, temporary files) and node memory



Conclusions

Blue Brain Project

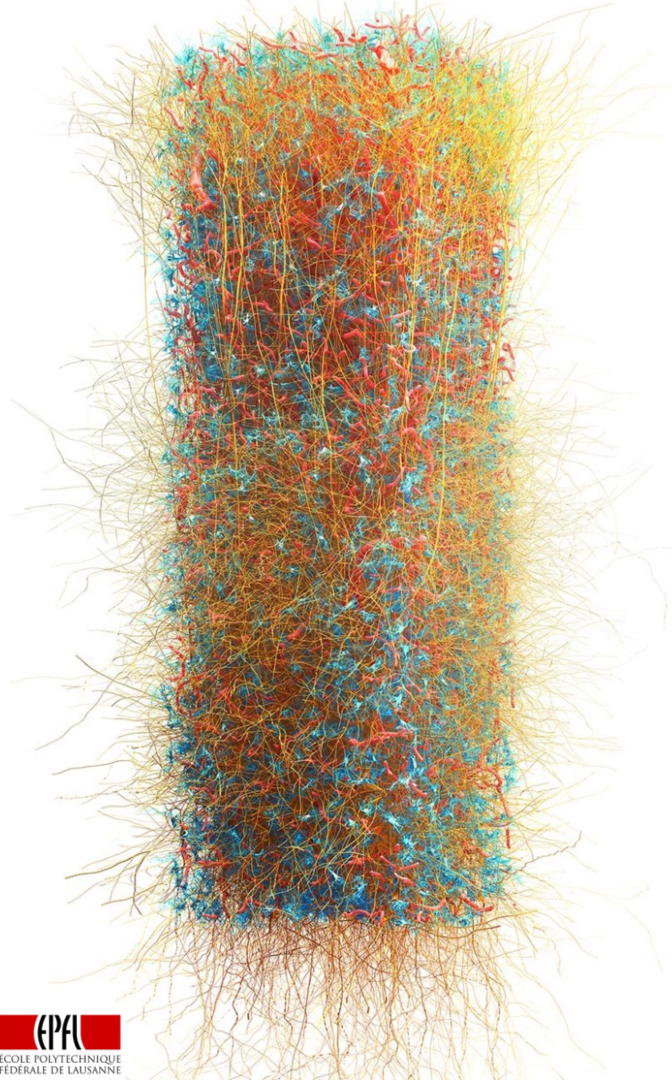
EPFL
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Conclusions

- The amount of data involved in brain tissue simulations is increasing → need for scalable solutions
- Spark improves the performance of our scientific pipeline at different stages
- Design and configuration decisions are a key aspect that impacts performance
- In our context, we are missing a few features:
  - NumPy support in DataFrames
  - Better integration with non-HDFS parallel file systems (GPFS)
  - Improved Pandas UDF support

Acknowledgements

- BBP *Cells & Circuits*, *Molecular Systems*, *Scientific Visualization* and *HPC* teams for the support, feedback and images provided
- An award of computing time was provided by the ALCF Data Science Program (ADSP)



Thank you! ☺

Questions…?