Accelerating Data Analysis in Simulation Neuroscience with Big Data Technologies

Judit Planas(✉), Fabien Delalondre, and Felix Schürmann

Blue Brain Project, École Polytechnique Fédérale de Lausanne, Geneva, Switzerland
{judit.planas,fabien.delalondre,felix.schuermann}@epfl.ch

Abstract. Important progress in computational sciences has been made possible recently thanks to the increasing computing power of high performance systems. Following this trend, larger scientific studies, like brain tissue simulations, will continue to grow in the future. In addition to the challenges of conducting these experiments, we foresee an explosion of the amount of data generated and the consequent unfeasibility of analyzing and understanding the results with the current techniques.

This paper proposes Neurolytics, a new data analysis framework, together with a new data layout. The implementation of Neurolytics is mainly focused on simulation neuroscience, although we believe that our design can be applied to other science domains. The framework relies on big data technologies, like Apache Spark, to enable fast, reliable and distributed analyses of brain simulation data in this case. Our experimental evaluation on a cluster of 100 nodes shows that Neurolytics gets up to 374x speed-up compared to a thread-parallel Python implementation and is on par with a highly optimized Spark code. This demonstrates the suitability of our proposal to help scientists structure and understand the results of their experiments in a fast and efficient way.

Keywords: Scientific data analysis · Simulation neuroscience
Big data · Scientific frameworks · Spark · Python

1 Introduction

Since the 1950s, the use of computers has been adopted by many scientific fields to help researchers advance in their areas [1,3,16]. Since then, the evolution of computer technology has helped the progress in computational and simulation science domains. Like many other fields, neuroscience is taking advantage of this growing computational power, but the real progress is yet to come. Neuroscientists collect experimental data at the laboratory which is then used to create models of the brain structure and its behavior. Later, these models are translated into computer programs that can digitally reproduce the neural activity of brain regions. Finally, simulation results are interpreted and sometimes used as an input to refine the computational models. In order to attempt to simulate

© Springer International Publishing AG, part of Springer Nature 2018
Y. Shi et al. (Eds.): ICCS 2018, LNCS 10860, pp. 363–377, 2018.
https://doi.org/10.1007/978-3-319-93698-7_28

the largest possible brain region, the use of high performance computing (HPC) resources is needed. In the past, numerous studies could not have been possible without the aid of computers, and especially HPC, such as [6,13,15,17–19].

The human brain has more than 86 billion neurons and they communicate to each other through a type of connection called *synapse*. Each neuron can have up to 10,000 synapses (in some cases even more) and the brain is able to create, rewire or destroy these connections over time (synaptic plasticity). Many applications simulate the behavior of the brain in different ways. In general, most simulations are divided into *time steps* and the state of neurons is recorded at each step. The final result is that at the end of the simulation we obtain a set of files that describe how the state of neurons evolved over time. The degree of detail, precision and simulated parameters (*e.g.* electrical signals, neuron connectivity, etc.) varies depending on the simulator and the purpose of the study. For example, the degree of detail can vary from modeling neurons as points (point neuron models), to modeling their tree structure (multi-compartment models), to modeling their behavior at a molecular level (subcellular models).

While it is essential to continuously optimize and update simulation science applications to exploit new software and hardware technologies, it is also important to inspect the whole scientific workflow to detect potential bottlenecks. We believe that understanding simulation results is as important as running the simulation itself and, with the growing computational power, there is a growing trend of simulation output data from gigabytes to terabytes (or even more) in the near future. Therefore, we focus our motivation and the work presented in this paper on the post-processing of neural simulation output data. First, we propose a new data layout to structure the information of simulation results that works efficiently with most of the data post-processing tasks done by the scientists. And second, we present Neurolytics, a parallel and distributed framework for data analysis. The particularity of Neurolytics is that it has been designed with domain-specific knowledge of simulation neuroscience. Hence, it is able to better understand and efficiently handle simulation output data. However, we believe that the logical structure of the framework can be easily adapted to other computational science domains. Neurolytics provides two sets of functionalities: one is oriented to users with little programming knowledge, with which they can compute the most common analyses with only a few lines of code, whereas the other one targets users with advanced programming skills and allows them to construct their own analyses with the help of the framework.

The paper is structured as follows: Sect. 2 gives more background information and the related work. Sections 3 and 4 describe our proposal and Sect. 5 contains its experimental evaluation. Finally, Sect. 6 concludes the paper.

2 Background and Related Research

This section gives additional background information related to this paper and explains previous and related work.

2.1 Simulation Neuroscience

An important aspect of simulation neuroscience is to study brain activity by running simulations on a computing system. Scientists often focus only on certain brain regions, therefore it is more precise to refer to those simulations as *brain tissue simulations*. Currently, even the most powerful supercomputers of the world cannot fit certain detailed cellular models representing the full human brain, and it will still take some time until we can fit realistic models.

In order to conduct brain tissue simulations, the first step needed is to create a model from data collected during *in vivo* or *in vitro* experiments. Then, these data are used to drive the *in silico* experiments (brain tissue simulations). Typically, we can divide *in silico* experiments into three phases: first, the user (the scientist) sets up the desired parameters of the simulation, like the neural model (brain region, number of neurons, neuronal connectivity, ...), the amount of biological time they want to simulate, the simulation time step, etc. Second, the neural simulator starts the simulation and stores the results on persistent storage (commonly hard disk). Usually, simulators alternate between advancing the simulation phase and storing the results to disk in order to improve performance. Thus, the generation of results has a direct impact on simulator execution time. Finally, once the simulation finishes and all the results are generated, they are processed by visualization or analysis tools that neuroscientists use to interpret them. Sometimes, they are also used as an input to refine the computational model. One of the challenges that scientists are currently facing is the complexity of simulation data analysis: as simulation data grows in size, it becomes more difficult for them to manage all the data, and sometimes it is even impossible to analyze the whole dataset due to hardware restrictions (like memory capacity) or programming complexity. In this context, we foresee an emerging need for improving the analysis workflow as more powerful computing systems enable larger or more detailed neural simulations. Consequently, we have detected the need for scalable data analysis solutions applied to neural simulations and also a redesigned data layout that suits the different analysis types.

2.2 Simulation Output Data

Even though different simulators (like NEST [12], NEURON [14] or STEPS [4]) generate different output results, we can find common points between them: first, the identification of different entities, the neurons. Second, the sense of evolution over time: neurons start in a certain state and they evolve to a potentially different state as time advances. And third, simulators record some type of information about neurons. If we look at specific frameworks, we will find many combinations: some have subdivisions of neurons to better represent their state, some will not record data at each time step, but every N time steps, and there is a wide variety of the kind of information (the neuron state) they can collect. But still, we can find the three aforementioned common points in most simulators.

Figure 1 illustrates an example of a simulation data layout and the different access patterns of post-processing tools, that can be divided into two types:

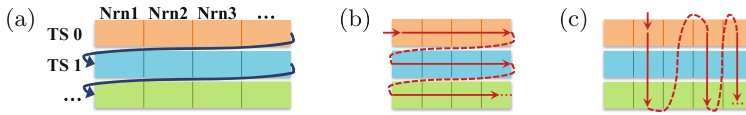

Fig. 1. (a) Example of simulation data layout: organized by time steps (rows), each time step (TS) contains information related to each neuron (Nrn). The contents of each neuron depend on the simulator and its configuration; Different access patterns: (b) sequential access: usually delivers good I/O performance and (c) random access: I/O performance becomes oftentimes the bottleneck

sequential accesses or random accesses. Although Fig. 1 refers to simulation neuroscience data, we can find many other science fields that use the same data structure to organize data in a 2-dimensional way.

2.3 Previous Research

There have been several efforts in the past years to improve the scientist workflow. Eilemann *et al.* [7] and Schürmann *et al.* [20] present a key/value store approach combined with IBM BlueGene Active Storage (BGAS) technology [10]. Later on, this work has been followed by other experiments on database frameworks, like Ceph, Cassandra or LevelDB, and flash-storage caching solutions [9]. What differentiates our proposal from these approaches is: (i) non-specialized hardware: Neurolytics sits on top of a widely adopted software framework that can be deployed on standard systems with no need for specialized hardware and (ii) Neurolytics combines domain-specific knowledge with an efficient data layout specially designed for neuroscience domains.

2.4 Related Work

Dask [5] is a flexible parallel computing library for analytic computing in Python. It has two main components: a dynamic task scheduler optimized for computation and a set of data collections for big data. Dask supports NumPy, Pandas and enables distributed computing by providing parallelized NP arrays and Pandas dataframes. Dask was initially designed for parallel numerical algorithms on a single computer, although nowadays it can run on a distributed cluster as well.

Thunder [11] is a collection of tools to analyze images and time series data in Python. It can run up to large scale clusters, using Spark underneath. Thunder provides the base classes and methods to conduct the analysis of data coming from a variety of fields, like neuroscience, video processing and climate analysis.

Dask and Thunder provide generic tools for data analysis that can be very powerful for scientists with medium to advanced programming skills. The main difference with Neurolytics is that while we also look for generality, we believe that it is still important to provide domain-specific tools, specially designed for those scientists with basic programming skills, or even with limited time, that

look for straightforward solutions with low time investment. To the best of our knowledge, such domain-specific tools have not been proposed to date.

BluePy [2] is a scientific library with a pythonic interface. It is developed by the Blue Brain Project (BBP) [8] and provides support for accessing all the entities involved in the ecosystem of BBP simulations. This includes, on the one side, the circuit (neural network) data and, on the other, the simulation data. Scientists use these entities for both circuit and simulation analyses. The main difference between BluePy and our proposal is that BluePy has been designed to run on limited computing resources (single node), like a scientist desktop computer, while Neurolytics is designed to scale on big data computing facilities. Nevertheless, BluePy benefits from the use of several tools underneath, like NumPy, SQLAlchemy or other libraries developed by BBP. In the same way, we believe that libraries like BluePy could be plugged on top of Neurolytics to analyze large sets of data in a fast and easy way from the scientist point of view.

3 Data Organization

Data organization plays an important role at both the computing and post-processing steps. However, sometimes these steps have completely opposite needs. For example, usually, the file structure generated by the simulator is designed to optimize how data is generated and written, but this does not meet the requirements of efficient reading by some post-processing tools, like visualization or data analysis. The latter tend to access data randomly, in small chunks. For example, a typical use case is to access the data belonging to a subset of neurons (whose 3D position in space falls into a certain brain region and is completely unrelated to the order in which the simulator writes the data). This means that only a small, non-contiguous subset of cells in each time step are required, but the reality is that, due to hardware and software design, unneeded adjacent data will also be loaded. In the end, on the one hand, we can find some types of data analysis that are more efficient if data is structured by time step, but on the other, there are other types of analyses that would benefit from a neuron-major structured layout. Consequently, there is no consensus on how simulation output data should be organized. For this reason, we think that there is a need to redesign the data layout.

3.1 Data Layout Proposal

We believe that the best approach to organize data is a key/value layout where the basic structure is a table with the following information: neuron ID (Nrn), time step (TS), data source (Src) and simulation data (Data), that corresponds to the state of each neuron at each time step. This table can be directly instantiated by different types of frameworks, for example as a database table or a dataframe. Figure 2a illustrates this layout. The coloring code matches the one used in Fig. 1a to highlight how information can be redistributed. We leave this design open, as other columns could be added in the future. For example, at the

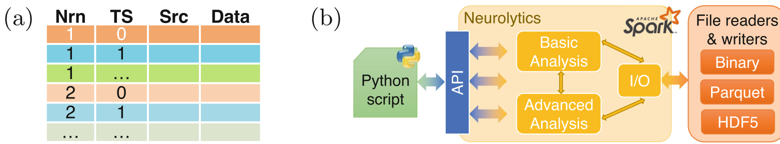

Fig. 2. (a) Proposed data layout, where the (Nrn, TS) key pair is used to access data; (b) Interaction of internal Neurolytics components and scientist's scripts

time of this table creation, certain precomputed analyses could also be added on-the-fly while report data is loaded. This could be interesting if certain operations are commonly computed by a large number of different analyses or users. Similarly, the data field is open to hold any type of information that the simulator generates, from a single value, to a struct, tuple or array of elements.

In addition to the minimum required data (Nrn, TS and Data), we propose to add the data source to have a reference to where data comes from. This may be useful for some types of analyses, but also for troubleshooting or data validation.

In the future, if proved necessary, other information related to simulation parameters or the neural network could be added. In this case, depending on the type of information, instead of creating a single, huge table, it may make more sense to add other tables and create links between them.

4 Neurolytics Framework

In addition to the strictly technical requirements, there are other factors that must be taken into account in our framework proposal.

Scalability. With the growing power of supercomputers, we expect simulations to be either more detailed (finer grain of time steps) or include a wider brain region (larger number of neurons). Therefore, reports will grow in terms of file size as well, most likely exponentially in the near future. In this scenario, it is only feasible to look for parallel, and preferably distributed, software solutions.

Python-Friendly. A large portion of the simulation neuroscience community use Python in their daily work, so proposing a Python-friendly solution would make the transition easier for them. In addition, there is the risk of low adoption, or even rejection by the community, if our framework requires mastering advanced programming skills or non-familiar programming languages.

SQL. Even if it is not a strong requirement, support of SQL-like queries would be preferred, as it would reduce the complexity of some types of analyses.

For all these reasons, and after considering different options, we believe that building Neurolytics on top of Spark is a promising approach for its maturity and the *ecosystem* it provides. Spark is part of the Hadoop BigData suite, the leading

open source project in big data. Also, Spark offers interesting features compared to other considered frameworks, i.e., the creation of optimized execution plans.

Neurolytics is a pythonic framework built on top of Spark that offers different types of functionalities classified into two categories: I/O and data analysis.

I/O. This category contains the components that interact with disk files and storage. The loading component includes all the functionalities related to reading simulation output files and loading its data into Spark distributed data structures. The storage component offers different options to store Spark data structures in Spark-supported formats. For example, once simulation data is loaded, we can use this functionality to store it as Parquet files. In addition, any data analysis computed after data loading can also be stored in any file format directly supported by Spark. Alternatively, other file formats could be supported as well, provided that the user implements the appropriate connector (plug-in).

Data Analysis. It consists of all the framework functionalities directly used for data analysis (basic and advanced functions), for example, grouping data with a certain criteria or doing some computation on report data.

Figure 2b shows the main components of Neurolytics and how they interact with each other. On the left side, there is the user (scientist) code, usually a Python script, that interacts with the interface of Neurolytics. Internally, our framework is structured as a set of components that communicate with each other (yellow arrows). Each component groups a different set of functionalities, as mentioned above. And on the right side, there are the *plug-in* agents. We call the plug-in agents all those components that are specific and implementation-dependent on external factors. For example, most of the entities involved in data loading and storage are considered plug-ins, because they need to understand the specific file format they are loading and they completely depend on both the format (binary, HDF5, Parquet, ...) and the internal layout (data structures, offsets to access certain data, ...). At this point, we have not identified plug-ins other than file readers and writers for Neurolytics.

We would like to highlight the convenience of Neurolytics design, as the core components are completely independent from file data sources. Therefore, by just adding the appropriate plug-in, the framework can immediately read data coming from any neuroscience simulator. In the same way, our data layout is flexible enough to adapt to many types of information as well. Our implementation targets neural data analysis, so we propose a specialized framework that understands the properties of neural simulation data and the relation between the different entities involved. However, our design can be easily adapted to other areas where data organization includes the sense of time evolution, and with minor changes, the framework could support most of 2-D data organizations.

4.1 User Interaction

As we mentioned before, our user profile are scientists from basic to advanced programming skills, therefore we have divided Neurolytics API into two groups.

Basic Functions. Little programming knowledge is required to interact with these functions. The main features offered by this group are: simulation data loading into Spark distributed structures (RDDs/DataFrames), write simulation data to a Spark-supported file format, aggregate data (by time step or by neuron), compute aggregated average values, generate the corresponding histograms and sample data. We expect advanced users to also use these operations.

Advanced Functions. We expect advanced users to use these functions in order to customize or fine-tune their data analysis. In addition to the basic set, we include here finer grain functions to get meta-information about simulation data, functions to retrieve the internal Spark distributed data structures (to enable custom distributed analysis and data manipulation) and generic functions to operate on the distributed data (for example: filtering, grouping by, element-to-element transformation). The latter accept user-defined functions and can be composed together to form complex analyses.

In order to maximize compatibility with other common Python libraries, we allow the user to choose the data structure returned by each function. In general, we always offer four options: *raw* (the original data structures, usually loaded from simulation output files), Python (basic Python data structures, like lists, tuples, etc.), NumPy N-dimensional arrays and Pandas DataFrames.

Since the amount of simulation output data can be significantly large, it is not uncommon to divide the information into several output files. This division can either be done by the simulator automatically, or manually by the scientist. In any case, grouping back together the information held in the resulting set of files may be a complex task, or even sometimes impossible due to the limited amount of dynamic memory (DRAM). Hence, Neurolytics adds an interesting functionality: the ability to aggregate all these subsequent simulation result files into a single data structure. Thanks to the Spark distributed data structures, we avoid memory problems, and our framework allows scientists to have a broader view of the simulation data at once, without any effort from their side.

4.2 Neurolytics Implementation Details

Spark offers two different distributed data structures: RDDs and DataFrames. Even though DataFrames are recommended over RDDs whenever possible, after a thorough evaluation, we realized that RDDs give several advantages in our context. Most of the data loaded from simulation data can be stored as NumPy arrays and scientists mainly use the NumPy library to do their analyses. Therefore, the main benefit of using RDDs is that they support the storage of NumPy arrays without conversions, as opposed to DataFrames, where NumPy arrays must be converted to a supported data format (like binary data or Python lists). However, the choice of NumPy arrays in the past was motivated for its performance and easiness of use, but this could change in the future, so we do not completely discard using DataFrames at some point.

During the process of loading data, there is a first step where simulation output files are scanned to get their meta-data, like the list of neuron IDs and the

time series. Then, this information is used to parallelize the loading of simulation data with Spark. Nevertheless, special care is taken in order to read chunks of data that are large enough to get good file I/O performance. Therefore, several stages are performed until data loading is finished and simulation data is stored following the layout described in Fig. 2a.

In order to proof the suitability of Neurolytics, we have chosen to implement the support for NEURON simulator. NEURON simulates the electrical activity of multi-compartment neurons. It receives different input parameters, like the biological neural model and it generates the desired output data in a single, potentially large file. We think that this use case is interesting and versatile because even though NEURON generates the simulation output files in binary format, a large number of scientists convert the result file into an HDF5 file because it is easier for them to read this file format from their Python scripts. In addition, during the conversion process from binary to HDF5, they also transpose data to organize results by neuron instead of by time step. Consequently, we have developed two different plug-ins to read both file formats with their different internal organizations to demonstrate the flexibility of Neurolytics to adapt to different formats and completely opposed data organization. Moreover, Neurolytics design is open to accept additional plug-ins to support multiple file formats generated by any simulation neuroscience framework.

Finally, Neurolytics adds the ability to store simulation data or, in general, any DataFrame or RDD generated inside the framework into the Parquet file format. This can be particularly useful to convert data from one format to another, or even to accelerate data loading if the same report files must be analyzed in different occasions. As a proof of concept, we have implemented the Parquet support, but this feature can be easily extended by adding other storage plug-ins.

5 Experimental Evaluation

In this section we evaluate the behavior of Neurolytics and compare its performance to an equivalent thread-parallel Python implementation. In order to conduct the evaluation, we have identified three common data analyses that scientists use, along with the process of loading data. The rest of this section explains these types of data analysis and presents the evaluation results.

5.1 Analysis Description

We identified three common types of data analysis. For simplicity we tag them as *Query 1* (*Q1*), *Query 2* (*Q2*) and *Query 3* (*Q3*).

The first analysis (Q1) consists of calculating the mean values of each neuron over time. The following steps must be followed in order to get the result:

1. For each neuron, compute the mean value of its data at each time step.
2. Group the mean values by neuron to obtain its evolution over time.
3. Scientists usually create a Python dictionary with neuron IDs as dictionary's keys and values are arrays with the computed mean values per time step.

In the second analysis (Q2), scientists want to generate a histogram for each simulation time step. The procedure to compute Q2 is described as the following:

1. For each neuron, compute the mean value of its data at each time step.
2. Group the mean values by time step.
3. For each time step, generate a histogram of the mean values of each neuron.
4. Similarly to Q1, structure these data into a dictionary with time steps as keys and values contain the computed histogram for each time step.

The third analysis consists of extracting a sample from the whole dataset. In this case, the scientist wants to get all the data related to randomly chosen neurons. Usually, the final result will include the data from tens to few hundreds of neurons. In our evaluation, we have chosen to sample the data of 250 neurons.

We would like to emphasize the relevance of these three analyses to evaluate our proposal. On the one hand, the first analysis requires data to be grouped by neuron and the second analysis groups data by time step. As explained in Sect. 3, some analysis access data per neuron, and thus, would benefit from a neuron-major data layout; but some others access data per time frame, so there is not a clear consensus on how data should be better structured. On the other hand, Q3 requires to filter a small subset of data, so only small chunks of data need to be read at randomly accessed locations. Thanks to the heterogeneity of the three queries, they represent a large set of analyses. This also exposes a challenge to our proposal, as it has to show its versatility against completely opposite data organizations and access patterns.

5.2 Performance Results

The evaluation was performed on Cooley, an analysis and visualization cluster at Argonne National Laboratory (ANL). The system has a total of 126 compute nodes. Each node has 2x Intel Haswell E5-2620 v3 (12 cores) and 384 GB of RAM. Nodes are connected through FDR Infiniband interconnect. Apache Spark version 2.1.2 with Python 3.6.3 were used to run all benchmarks.

We present the scalability results of Neurolytics, running from 1 node up to full scale (96 nodes). For each execution, we placed the Spark master process on a different node than the ones running the Spark slave processes. In all cases we had exclusive access to the computing nodes. The results shown in this paper were computed as the arithmetic average of multiple runs (at least 10).

We ran the three types of data analyses explained in this section with Neurolytics and we include as well the process of loading data from disk (GPFS). In addition, we provide the results of the same actions performed in: (i) a thread-parallel Python script, run on a single node (using 12 cores) and (ii) a Spark optimized code, executed directly on the same Spark cluster. We refer to these three runtime scenarios as Neurolytics, Python and Spark respectively. The Python code tries to mimic the scientist environment and to show what would be the benefits of using Neurolytics. It uses the NumPy library with NumPy structures for computations and data is loaded in chunks as large as possible to favor file I/O

reading performance. The optimized Spark code reproduces how an advanced programmer would write these data analysis with Spark (using Python).

In order to make this evaluation more realistic, we have used real simulation data generated by NEURON simulator. The activity of a somatosensory cortex region with 31000 neurons was simulated for 7 min of biological time. The activity was recorded (written to the simulation output files) every 100 ms (biological), generating a total of 4200 time frames. In order to control file size, the simulation was split into 14 runs, so 14 output files were created of 47 GB each (658 GB in total). Furthermore, we artificially duplicated these data to obtain a larger dataset of 1.3 TB. Both datasets were converted to the Parquet file format using Neurolytics, so we then ran all the experiments with both datasets (650 GB and 1.3 TB) in both file formats (original binary and converted Parquet).

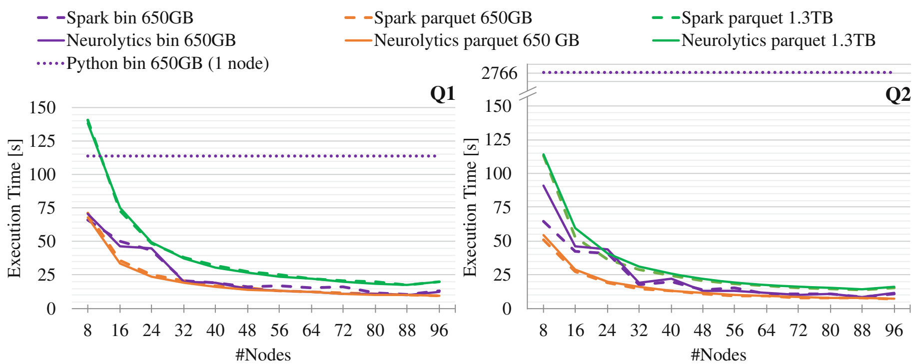

Fig. 3. Wall clock time of Q1 (left) and Q2 (right); note the axis is cut in Q2 (Color figure online)

First, we present the results of executing the three data analysis with the different combinations of runtimes (Python, Spark and Neurolytics), dataset sizes (650 GB and 1.3 TB) and file formats (binary and Parquet), followed by the results of data loading. For simplicity, we do not show all possible combinations, but only the relevant ones. All executions follow a strong scaling evaluation.

All charts presented in this section have the following conventions: (i) colors: **purple** experiments were run with the binary dataset of 650 GB; **orange** refers to the same dataset in Parquet format, and **green** refers to the duplicated dataset of 1.3 TB in Parquet format; (ii) line shape: **small dotted lines** refer to the thread-parallel Python executions; **large dashed lines** refer to the Spark experiments and **solid lines** refer to the Neurolytics executions. Finally, we would like to highlight that the Python data points (purple dotted lines) show the Python execution on a single node, but, for readability, this line has been extended along the charts.

Figure 3 (left) shows the time needed to perform the data analysis presented as *Query 1*. Python's dotted purple line, run on single node, has been extended

for readability. We can see that the lines corresponding to Spark and Neurolytics executions overlap almost perfectly, meaning that Neurolytics reaches an optimal performance in this type of analysis. The executions scale properly according to the number of nodes, and Neurolytics is always between 1.6–12x faster than the Python version. We cannot expect perfect scaling because this type of query requires data exchange between Spark slaves in both Spark and Neurolytics versions. There is also a perfect scaling between the 650 GB and the 1.3 TB datasets, as the latter takes at most twice the time (or sometimes even a bit less). As we increase the number of nodes, there is less work to do per Spark slave, reaching up to a point where the gain is minimal. However, these results are encouraging, specially when, in the near future, simulation output data grows even more in size. Finally, we would like to remark that thanks to the big DRAM capacity of the system, the whole dataset can be loaded in memory to perform the Python analysis. However, this is not the usual case, and it would lead to longer execution times on clusters with less DRAM. In such situation, Spark and Neurolytics would not be impacted, as data are stored in a distributed fashion.

Figure 3 (right) shows the time needed to perform the data analysis presented as *Q2*. Note that the vertical axis has been split to better show the results. Spark and Neurolytics present similar scalability compared to *Q1*. However, in this case, the benefit of computing this analysis with Neurolytics is evident. The reason is that the original data organization and the intrinsic operations of this query work in opposite directions, making the Python analysis very inefficient. On the contrary, thanks to our proposed layout, Neurolytics is not affected by these issues and offers good performance, reaching a maximum speed-up of 374x with respect the Python code. Spark executions use the same data layout as Neurolytics, so they also adapt well to different data organizations and operations. Similarly to *Q1*, we would like to remark that the Python analysis was done with all data loaded in memory, but this is only possible thanks to the hardware properties. Otherwise, this analysis would have been more complex to program and slower to execute. In addition, we observe different behavior depending on the file format: when data comes from Parquet files, which are part of the Spark ecosystem, the behavior of both Spark and Neurolytics is more regular, as opposed to the case when data comes from the original binary format, where we can see that in some cases performance scales irregularly.

These two types of analyses clearly show that Neurolytics can perform analyses requiring data in completely opposite formats. Moreover, performance is not impacted by the data layout, as the execution time is very similar in both cases.

Figure 4 (left) shows the time needed to perform the data sampling presented as *Q3*. Like in previous charts, the Python dotted line, run on single node, has been extended for readability. This type of analysis done in Python is quite slow due to the inefficiency of accessing simulation output files in random patterns that hinder I/O performance. In the case of Spark and Neurolytics, our proposed finer grain data layout favors a fastest execution. However, since the amount of data sampled is relatively small, the scaling performance saturates when we use

a large amount of nodes. In this query, Neurolytics reaches up to 18x speed-up with respect to the Python execution.

Figure 4 (right) shows the time needed to load the simulation data with the Python script and with Neurolytics (Spark RDD underneath). As in the other cases, the Python dotted line has been extended for readability. Neurolytics shows extremely good performance when loading Parquet data, as this file format has been designed to work with Spark. However, if we load from binary data, good performance is only achieved when using a few compute nodes; as we scale to a larger number of nodes, performance degrades. This is because the original simulation data files are not prepared nor structured to be read from massively parallel processes and, therefore, the data reading collapses when all Spark slaves are simultaneously accessing the same files through GPFS file system.

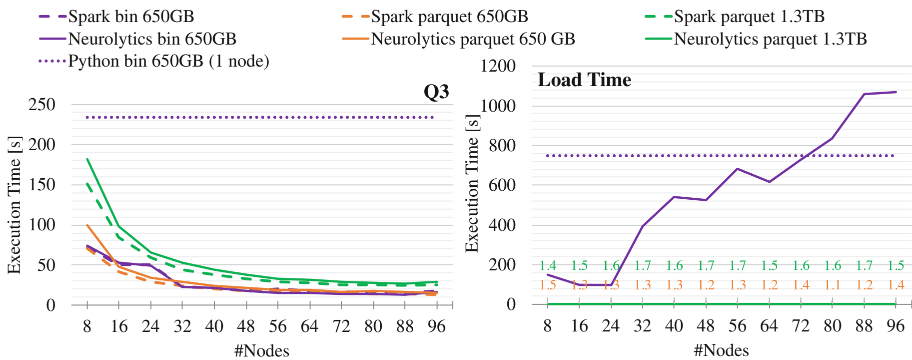

Fig. 4. Wall clock time of Q3 data analysis (left) and data loading (right)

6 Conclusions

With the increasing power of supercomputers, scientists can run larger simulations of their studies, but they also produce larger amounts of data that bring new challenges at the time of analyzing the results. From our point of view, such data analysis is as important as the simulation itself, and in some cases it is even used to refine the computational model to make it more accurate. We present in this paper Neurolytics, a Spark-based data analysis framework, together with a data layout with domain-specific knowledge. The implementation is focused on the field of simulation neuroscience and proves its suitability to process the results of such simulations. However, the design is general enough to be applied to other computational science areas.

In our evaluation, we used a real set of simulation output files (up to a few terabytes) to compute several data analyses and compared the performance of Neurolytics to a Spark optimized implementation (reference of maximum performance) and to a thread-parallel Python implementation (similar to real neuroscientist scripts). We can then extract the following conclusions: first, the

suitability of Neurolytics for large data analysis in computational science, specially in simulation neuroscience. Depending on the analysis, Neurolytics can get up to 374x speed-up at full scale compared to the Python parallel code. Our framework scales well up to full-scale executions (100 nodes) and is always on par with the performance of the reference Spark code. Second, the suitability of a new data layout with domain-specific knowledge for neural simulation data analysis. Our proposal adapts well to the most common types of analysis patterns (i.e. by neuron and by time step), one being completely opposite to the other. Finally, the current simulation output file is not the optimal approach for big data frameworks, like Neurolytics. However, low-impact changes, like a file converter, would address this problem and significantly improve the overall performance.

As future work, we would like to evaluate the viability of integrating our proposal into the workflow of scientists, so that the most common data analyses are automatically computed in-place (where the simulator runs) and the results are immediately available to the scientist.

Acknowledgment. We would like to thank the BBP HPC and In Silico teams for their support. This work is funded by EPFL Blue Brain Project (funded by Swiss ETH board). An award of computer time was provided by the INCITE program and the ALCF Data Science Program. This research used resources of ALCF, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

References

1. Baillargeon, B., et al.: The living heart project: a robust and integrative simulator for human heart function. Eur. J. Mech. A/Solids **48**(Suppl. C), 38–47 (2014)
2. BluePy. https://developer.humanbrainproject.eu/docs/projects/bluepy/0.5.9
3. Charney, J.G., Fjörtoft, R., Von Neumann, J.: Numerical integration of the barotropic vorticity equation. Tellus **2**(4), 237–254 (1950)
4. Chen, W., et al.: Parallel STEPS: large scale stochastic spatial reaction-diffusion simulation with high performance computers. Front. Neuroinform. **11**, 13 (2017)
5. Dask: Library for dynamic task scheduling (2016). http://dask.pydata.org
6. Diesmann, M.: Brain-scale neuronal network simulations on K. In: Proceedings of 4th Biosupercomputing Symposium: Next-Generation ISLiM Program of MEXT (2012)
7. Eilemann, S., et al.: Key/value-enabled flash memory for complex scientific workflows with on-line analysis and visualization. In: 2016 IEEE IPDPS, pp. 608–617 (2016)
8. EPFL: Blue Brain Project. http://bluebrain.epfl.ch
9. Ewart, T., Planas, J., Cremonesi, F., Langen, K., Schürmann, F., Delalondre, F.: Neuromapp: a mini-application framework to improve neural simulators. In: Kunkel, J.M., Yokota, R., Balaji, P., Keyes, D. (eds.) ISC 2017. LNCS, vol. 10266, pp. 181–198. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-58667-0_10
10. Fitch, B.: Exploring the capabilities of a massively scalable, compute-in-storage architecture. www.hpdc.org/2013/site/files/HPDC13_Fitch_BlueGene_ActiveStorage.pdf

11. Freeman, J., et al.: Mapping brain activity at scale with cluster computing. Nat. Meth. **11**(9), 941–950 (2014)
12. Gewaltig, M.O., et al.: NEST (NEural Simulation Tool). Scholarpedia **2**(4), 1430 (2007)
13. Hereld, M., et al.: Large neural simulations on large parallel computers. Int. J. Bioelectromagn. **7**(1), 44–46 (2005)
14. Hines, M.: NEURON — a program for simulation of nerve equations. In: Eeckman, F.H. (ed.) Neural Systems: Analysis and Modeling, pp. 127–136. Springer, Boston (1993). https://doi.org/10.1007/978-1-4615-3560-7_11
15. Hines, M., et al.: Comparison of neuronal spike exchange methods on a Blue Gene/P supercomputer. Front. Comput. Neurosci. **5**, 49 (2011)
16. Lapostolle, P., Bail, R.L.: Two-dimensional computer simulation of high intensity proton beams. Comput. Phys. Commun. **4**(3), 333–338 (1972)
17. Markram, H., et al.: Reconstruction and simulation of neocortical microcircuitry. Cell **163**(2), 456–492 (2015)
18. Plesser, H.E., Eppler, J.M., Morrison, A., Diesmann, M., Gewaltig, M.-O.: Efficient parallel simulation of large-scale neuronal networks on clusters of multiprocessor computers. In: Kermarrec, A.-M., Bougé, L., Priol, T. (eds.) Euro-Par 2007. LNCS, vol. 4641, pp. 672–681. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74466-5_71
19. Reimann, M.W., et al.: Cliques of neurons bound into cavities provide a missing link between structure and function. Front. Comput. Neurosci. **11**, 48 (2017)
20. Schürmann, F., et al.: Rebasing I/O for scientific computing: leveraging storage class memory in an IBM BlueGene/Q supercomputer. In: Kunkel, J.M., Ludwig, T., Meuer, H.W. (eds.) ISC 2014. LNCS, vol. 8488, pp. 331–347. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07518-1_21