Performance comparison of Dask and Apache Spark on HPC systems

Mathieu Dugré, Valérie Hayot-Sasson, Tristan Glatard
Department of Computer Science and Software Engineering
Concordia University, Montréal, Québec, Canada
{mathieu.dugre, valerie.hayot-sasson, tristan.glatard}@concordia.ca

***Abstract*—**

***Index Terms*—Performance, Big Data, Dask, Spark, Neuroimaging**

From Mathieu: Discuss which keyword to use for the paper.

I. Introduction

The raise in data sharing coupled with improved data collection technologies lead neuroimaging into the Big Data era [1], [2]. From Mathieu: cite CONP portal While common neuroimaging workflow engines, such as Nipype [3], are well rounded to process standard compute-intensive pipeline, they lack Big Data strategies (i.e. in-memory computing, data locality, and lazy-evaluation) to improve the performance of increasingly prevalent data-intensive pipelines. A previous study [4] notes the importantce of in-memory computing and data locality to improve the performance of data-intensive pipelines. In this work, we study how performance benchmarks generalize accross Big Data engines in an HPC environement. In particular, we study the performance difference between Dask [5] and Apache Spark [6], for their sutability to process neuroimaging pipelines. Our objective is to assess whether Spark or Dask has substantial performance benefits to process data-intensive pipeline in HPC clusters.

Spark and Dask both provide in-memory computing, data locality, and lazy-evlaution, which is common for Big Data engines. Both of their scheduler operate dynamically, which benefit applications with runtime unknown ahead of time [5]. They also provide rich high-level API and support a variety of scyeduler and deployment environment, such as Mesos [7], YARN [8], Kubernetes, and HPC clusters. Although sharing similarities, these engines have differences.

First and foremost, Spark is written in Scala while Dask is in Python. Given the popularity of Python in the scientific community, this arguably gives a edge to Dask due to the serialization cost between Python and Scala. However, if not careful, the Python Global Intepreter Lock (GIL) can significatly reduce the parallelism of an application, in some cases. The difference in programmming language also provides different benefits. On the one hand, as part of the Scipy ecosystem, Dask provides almost transparent integration with APIs such as Numpy arrays, Pandas dataframe, or RAPIDS GPU accelerated code framework. On the other hand, Spark's Java, Scala, R, and Python APIs allow to easily parallelize pipelines with minimal performance loss; thanks to the JVM. Our study is restrained to performance, although, we recognize that, in practice, other factors affect the choice of an engine.

While laptops or workstations can be sufficient for some applications, data-intensive neuroimaging pipelines often require large infrastructure to perform studies. This paper focuses on the performance of Big Data engine in HPCs environments.

The next section introduce the Lustre file system and both Big Data engines used: Dask and Apache Spark. Following, we present the design of our benchmarking experiments. We consider two type of data-intensive neuroimaging pipelines: high-resolution imaging and large functional MRI studies. The pipelines we chose involve different compute and I/O patterns; e.g. map-reduce or map-only and requiring partial data in memory or the whole dataset. Our experiments were executed on a dedicated cluster representative of an HPC environment used to process state-of-the-art data-intensive neuroimaging studies. Further sections present our results, dicsussion, and concusion.

II. Background

A. Lustre

B. Dask

Dask is a Python-based Big Data engine with growing popularity in the scientific Python ecosystem. Dask was designed with data locality and in-memory computing in mind, to mitigate the data transfer bottleneck in Big Data workflows. Data locality, popularized by Map-Reduce [9], schedules tasks where the data reside. In-memory computing minimizes the overhead of transferring data to disk by keeping data in memory when possible. Dask uses lazy evaluation to reduce unnecessary communication and computation. The engine builds a dynamic graph before execution, allowing it to determine which task to compute. Dask workflows can further reduce data transfer by leveraging multithreading whenever Python's GIL does not restrict it. Fault-tolerance is achieved by recording data lineage: the sequence of operations used to modify the initial data.

Dask offers five data structures: Array, Bag, DataFrame, Delayed, and Futures. Arrays offer a clone of NumPy API for

distributed processing of large arrays. Bags are a distributed collection of Python object that offers a programming abstraction similar to PyToolz. Dataframes are a parallel composition of Pandas Dataframes used to process a large amount of tabular data. Dask Delayed offers an API for distributing arbitrary functions that do not fit in the above frameworks. Lastly, Dask Futures can also execute arbitrary functions; however, it launches computation immediately rather than lazily. Dask modularity allows users to install only required components making it lightweight.

In Dask, a scheduler decides where and when to execute tasks using the Dask graph. API operations generate multiple fine-coarse tasks in the computation graph, allowing a more straightforward representation of complex algorithms.

The Dask engine is compatible with multiple distributed schedulers, including YARN and Mesos. Dask also provides its own *Dask Distributed scheduler*. We chose to use Dask Distributed scheduler to keep the environment balanced between the engines.

In the Dask Distributed scheduler, a *dask-scheduler* process administrates the resource provided by *dask-worker*s in the cluster. The scheduler receives jobs from clients and assigns tasks to available workers. Task scheduler uses a LIFO (Last-In-First-Out) job scheduling policy. That is an utter process branch of the Dask graph before proceeding with the next one.

Dask offers multiple ways to deploy a cluster, including, but not limited to, SSH configs, Kubernetes, SLURM, PBS. For our experiments, we used the Dask SLURM cluster API.

C. Apache Spark

Apache Spark is a widely-used general-purpose Big Data engine. Like Dask, it aims at reducing data transfer costs by incorporating data locality, in-memory computing, and lazy evaluation.

Spark offers three options to schedule jobs: Spark Standalone, Mesos, and YARN. Spark Standalone is a simple built-in scheduler. YARN is mainly used to schedule Hadoop-based workflows, while Mesos can be used for various workflows. We limit our focus to Spark Standalone scheduler, as researchers are likely to execute their workflows in an HPC environment where, usually, neither YARN nor Mesos is available.

In the Spark Standalone scheduler, a *leader* From Tristan: I am all in favor of inclusive terminology, but if Spark didn't update their wording we shouldn't do it for them coordinates the resource provisioned by *workers* in the cluster. A *driver* process receives jobs from clients and requests workers from the leader. Jobs are divided into stages to be executed onto workers. Each operation in a stage is represented by a high-level task in the computation graph. Like Dask, Spark Standalone scheduler uses a LIFO policy to schedule tasks. Spark Standalone has two execution modes: (1) the client mode, where the driver process runs in a dedicated process, and (2) the cluster mode, where the driver runs within a worker process. Our experiments use the client mode since cluster mode is not available in PySpark.

Spark's primary data structure is Resilient Distributed Dataset (RDD) [10], a fault-tolerant, parallel collection of data elements. RDDs are the basis of the other Spark data structure: Datasets and DataFrames. Datasets are similar to RDD but benefit additional performance by leveraging the Spark SQL's optimized execution engine. The DataFrames are Datasets organized into named-columns and are used to process tabular data. While the DataFrame API is available in all supported languages, Datasets are limited to Scala and Java.

Python is a standard programming language in the scientific community, offering numerous data processing libraries. While serialization from Python to Java, an operation required when using Spark's Python API, creates overhead, we found it minimal [11]. We focus on PySpark API to have a more balanced environment between the different engines and for its suitability to neuroimaging.

III. Methods

A. Infrastructure

For our experiments we used a dedicated cluster. Each compute node has 2 16-cores CPUs (64 threads total), 256 GB 2666 MHz memory in 8-channel, and 2.88 TB SSDs of mounted storage. The nodes are interconnected with a dedicated 10 Gbit/s Ethernet connection. Centos 8 with Linux kernel *4.18.0-240.1.1.el8_lustre.x86_64* installed on the compute nodes.

Both Spark and Dask are configured to have 8 worker processes each with 8 threads. Each worker is allocated 31.5 GB of memory. A new cluster is spinned up and teared down for each experiment.

Dask ?? and Spark ?? From Mathieu: Add version when after locking it for experiments. was used for our experiments.

B. Dataset

We used BigBrain [12], a 3-D image of the human brain with voxel intensity ranging between 0 and 65,535. We converted the blocks into the NIfTI format, a popular format in neuroimaging. We left the NIfTI blocks uncompressed, resulting in a total data size of 648 GB. To evaluate the effect of block size, we resplit these blocks into 1000, 2500, and 5000 blocks of 648 MB, 259.2 MB, and 129.6 MB, respectively. sam library was used to resplit the image.

We also used the dataset provided by the Consortium for Reliability and Reproducibility (CoRR) [13], freely available on datalad. The entire dataset is 408.4 GB, containing anatomical, diffusion and functional images of 1,397 subjects acquired in 29 sites. We used all 3,491 anatomical images, representing 39 GB overall (11.17 MB per image on average).

C. Applications

1) Increment: We adapted the increment application used in [4]. This synthetic application reads blocks of the BigBrain from Lustre and simulates computation by sleeping for a

specified period. To simulate intermediate results, we repeat the sleep process for a configurable amount of time. We prevent data caching of the blocks by incrementing their voxels value by one after each sleep operation. Finally, we write the resulting NIfTI image back to Lustre. Using this application we study the engines when their inputs are processed independently. The map-only scenario of this application mimics the processing of multiple independent subjects in parallel.

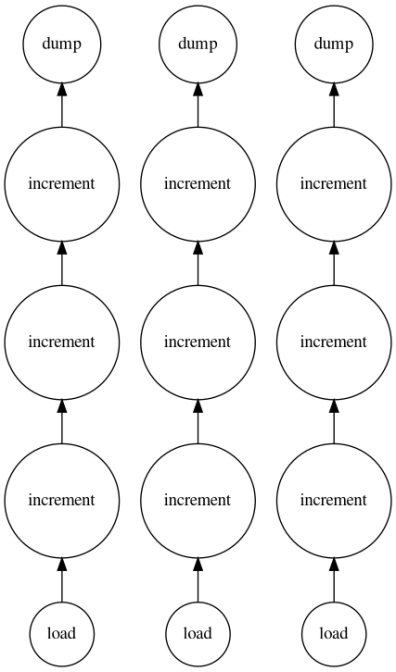

Fig. 1: Task graph for Incrementation with 3 iterations and 3 BigBrain blocks.

2) Multi-Increment: Our second application is an adaptation of the increment application. A significant difference is that, at each iteration, it uses a random BigBrain block as the increment value. This change allows the multi-increment application to have inter-worker communication while remaining simple.

3) Histogram: As our third application, we calculate the histogram of the BigBrain image. The application reads the BigBrain blocks from Lustre, calculates each intensity's frequency, and then writes the aggregated result back on Lustre. This map-reduce application has a very high read overwrite ratio. Moreover, this application requires shuffling, albeit of a limited amoutn of data. The amount of inter-worker communication is in-between the increment and multi-increment applications.

4) Kmeans: For our fourth application, we apply Kmeans clustering to the voxel intensities of the BigBrain image. We set the number of clusters to 3, to segment the white and grey matter and the noise. The application starts by reading the image blocks, combining all voxels in a 1-D array, and choosing initial centroids using the min, max, and intermediate values. It assigns each voxel to the centroids it is the closest and updates each centroid by computing the average of the voxels associated with it. It repeats the assignment and update steps for a configurable amount of time. Finally, the voxels of the image blocks are classified and written back to the file system. Updating the centroids involves substantial data communication between the workers.

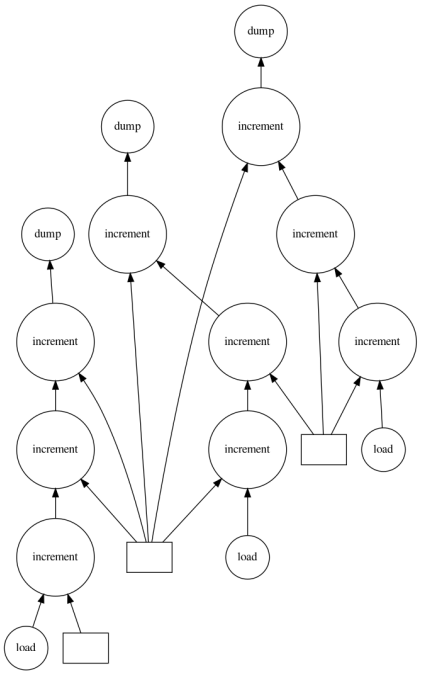

Fig. 2: Task graph for Multi-Incrementation with 3 iterations and 3 BigBrain blocks.

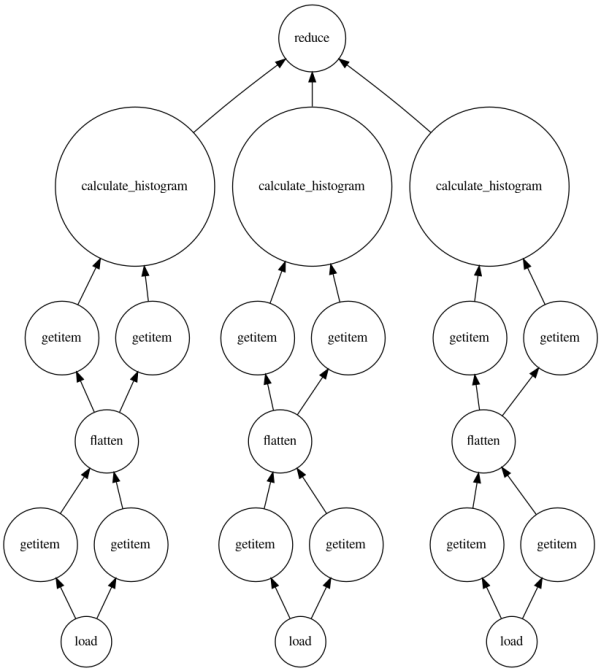

Fig. 3: Task graph for Histogram with 3 BigBrain blocks.

For this application, the Spark and Dask implementations differ slightly, to take advantage of the best-suited API from both engines. The Spark implementation uses the Map-Reduce paradigm, while the Dask one uses array programming.

5) BIDS App example: Our fifth application is BIDS App example: a neuroimaging pipeline to measure the brain volume from MRIs. For this application, we use the CoRR dataset. The application extracts the brain volume of each participant, then computes the average for each group of participants. Unlike the other applications, BIDS App example is a command-line executed in a Docker image (bids/example on DockerHub). We converted the Docker image to a Singularity image for use in HPC environments, From Mathieu: Cite paper on reason why this is done. using docker2singularity

D. Experiments

Table I the four parameters that are varied throughout the experiments. We varied (1) the number of workers to assess the scalibility of the scheduler for the engine, (2) the BigBrain block size in Increment, Multi-Increment, and Histogram to measure the effect of the different I/O pattern and parallelization degrees, (3) the number of iterations to evaluate the effect of number of task, and (4) the sleep delay to study the effect of task duration. It should note that increasing the number of iterations for a given sleep delay also increases the total compute time of an application.

To avoid potential external bias such as caching, background process and network load, we ran the applications in randomized order and cleared the page cache in between every experiments. Each benchmark was run ten times.

For each run, we measure the makespan of the application as well as the cumulative time spent in the different functions for read, processing, and writing data. The overhead calculation for each CPU thread is the end time of the last processed task minus the total runtime of the tasks ran for this thread. Summing those results gives the total overhead for the application.

IV. Results

A. Increment

Figure 4 depicts the total execution time spent in each code segment of the *Increment* application. The bars are broken down into Idle, Load, Dump, and other functions representing the overhead from the engine, the read and write time, and the aplication function, respectilvely. The mean of each bar was calulated from 10 repetitions, with the error bar representing the standard deviation. As expected, the compute time, from *Increment*, remains the same when we vary the number of nodes. However, the *Load* and *Dump* time increase with the number of nodes for both engines. This is explained by an I/O bottleneck on the Lustre file system.

From Figure 4, we also observe that Dask's overhead is higher than Spark's. While the overhead of both engines increases with the number of nodes, Dask's overhead is significantly worse when scaling. The difference in total execution time between the engines is explained from this overhead difference.

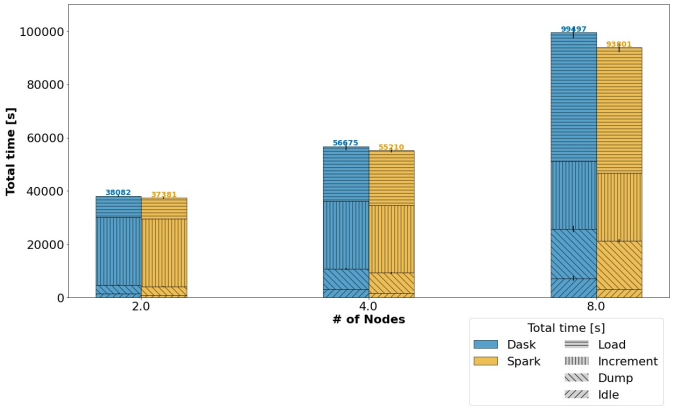

Fig. 4: Increment: varying nodes – 5000 block resplit, 5 iterations, 1 s delay

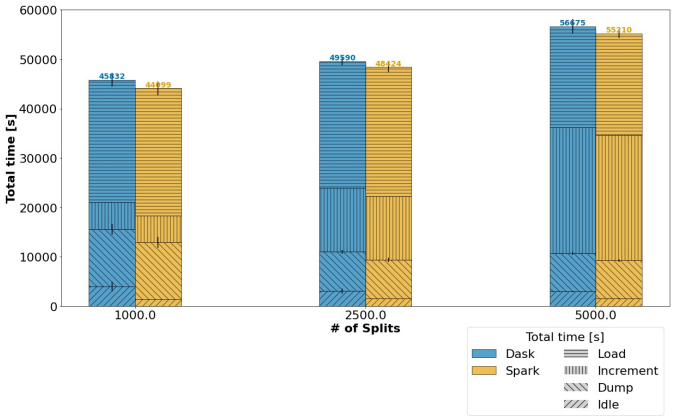

Fig. 5: Increment: varying resplit – 4 nodes, 5 iterations, 1 s delay

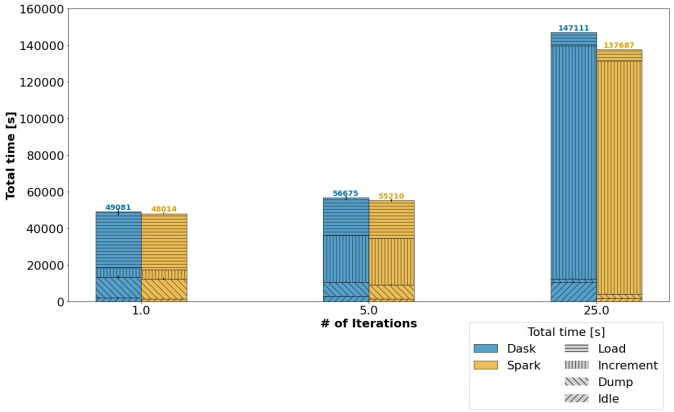

Fig. 6: Increment: varying iterations – 4 nodes, 5000 block resplit, 1 s delay

TABLE I: Parameters for the experiments

	Increment	Multi-Increment	Kmeans	Histogram	BIDS Example
# of Nodes	2, 4, 8				
Dataset size [GB]	648	486		648	39
Number of file	1000, 2500, 5000				3491
# of Iterations	1, 5, 25	1, 3, 9		-	

B. Multi-Increment

C. Kmeans

D. Histogram

Figure 7 depicts the execution time for the *Histogram* when varying the number of nodes.

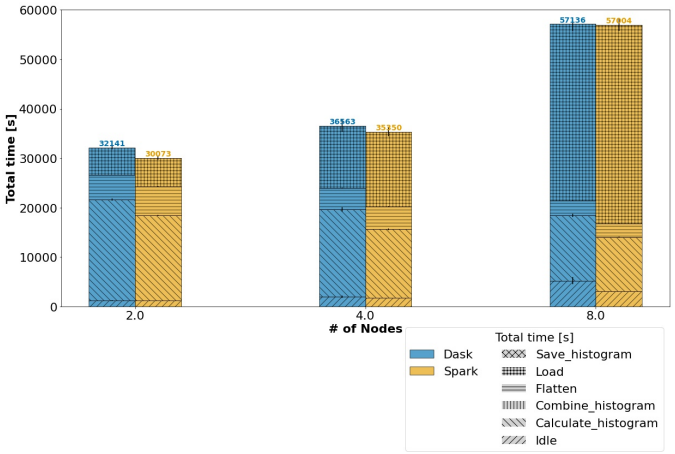

Fig. 7: Histogram: varying nodes – 5000 block resplit, 5 iterations, 1 s delay

Figure 7 shows the execution time for the *Histogram* when varying the number of resplit.

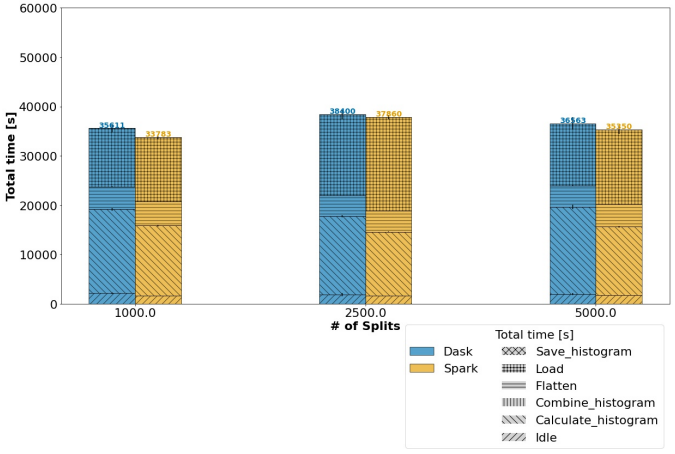

Fig. 8: Histogram: varying resplit – 4 nodes, 5 iterations, 1 s delay

E. BIDS Example

Figure 9 shows the total execution time for the *BIDS App example* application when varying the number of nodes. We observe that Dask has significantly lower execution time than Spark; between 10% to 15%. The main difference comes from a increased amount of stagger tasks for Spark, thus increasing the idle time of the application. Moreover, a shorther cluster deployement time for Dask reinforce the difference in idle time. As with other experiments, the I/O from Lustre affects the scaling of both engines as we increase the number of nodes.

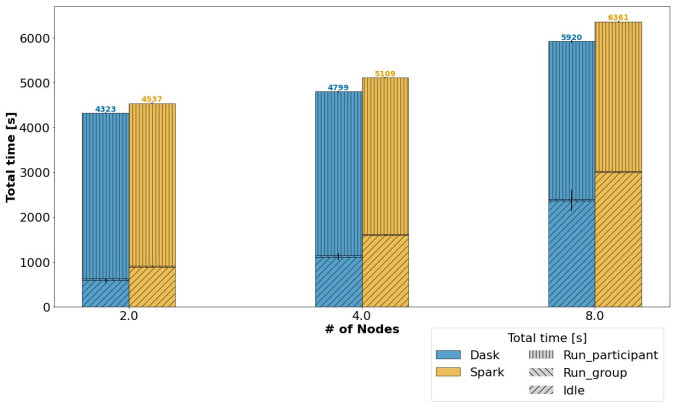

Fig. 9: BIDS App Example: varying nodes

V. Discussion

VI. Conclusion

Acknowledgment

References

[1] F. Alfaro-Almagro, M. Jenkinson, N. K. Bangerter, J. L. Andersson, L. Griffanti, G. Douaud, S. N. Sotiropoulos, S. Jbabdi, M. Hernandez-Fernandez, E. Vallee, D. Vidaurre, M. Webster, P. McCarthy, C. Rorden, A. Daducci, D. C. Alexander, H. Zhang, I. Dragonu, P. M. Matthews, K. L. Miller, and S. M. Smith, "Image processing and Quality Control for the first 10,000 brain imaging datasets from UK Biobank," *NeuroImage*, vol. 166, pp. 400 – 424, 2018.

[2] J. D. Van Horn and A. W. Toga, "Human neuroimaging as a Big Data science," *Brain imaging and behavior*, vol. 8, no. 2, pp. 323–331, 2014.

[3] K. Gorgolewski, C. Burns, C. Madison, D. Clark, Y. Halchenko, M. Waskom, and S. Ghosh, "Nipype: A Flexible, Lightweight and Extensible Neuroimaging Data Processing Framework in Python," *Frontiers in Neuroinformatics*, vol. 5, p. 13, 2011.

[4] V. Hayot-Sasson, S. T. Brown, and T. Glatard, "Performance Evaluation of Big Data Processing Strategies for Neuroimaging," in *19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), Larnaca, Cyprus*, 2019.

[5] Matthew Rocklin, "Dask: Parallel Computation with Blocked algorithms and Task Scheduling," in *Proceedings of the 14th Python in Science Conference*, Kathryn Huff and James Bergstra, Eds., 2015, pp. 126 – 132.

[6] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache Spark: A Unified Engine for Big Data Processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, Oct. 2016.

[7] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center." in *NSDI*, vol. 11, no. 2011, 2011, pp. 22–22.

[8] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache Hadoop YARN: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 5.

[9] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[10] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 2–2.

[11] M. Dugr, V. Hayot-Sasson, and T. Glatard, "A performance comparison of dask and apache spark for data-intensive neuroimaging pipelines," in *2019 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*, 2019, pp. 40–49.

[12] K. Amunts, C. Lepage, L. Borgeat, H. Mohlberg, T. Dickscheid, M.-É. Rousseau, S. Bludau, P.-L. Bazin, L. B. Lewis, A.-M. Oros-Peusquens, N. J. Shah, T. Lippert, K. Zilles, and A. C. Evans, "BigBrain: An Ultrahigh-Resolution 3D Human Brain Model," *Science*, vol. 340, no. 6139, pp. 1472–1475, 2013.

[13] X.-N. Zuo, J. S. Anderson, P. Bellec, R. M. Birn, B. B. Biswal, J. Blautzik, J. C. Breitner, R. L. Buckner, V. D. Calhoun, F. X. Castellanos *et al.*, "An open science resource for establishing reliability and reproducibility in functional connectomics," *Scientific data*, vol. 1, p. 140049, 2014.