Evaluate the Dask distributed computing framework in respect to various scientific computing tasks

Jeyhun Abbasov
Institute of Computer Science
University of Tartu
Ulikooli 18, 50090 Tartu, Estonia
jeyhun.abbasov@ut.ee

***Abstract*— Data Science is becoming more prevalent every day. That is the reason, there are a lot software libraries for data manipulation and analysis. One of the most popular software libraries that using for scientific computing tasks is Dask. Dask scales Python code from multi-core local machines to large distributed clusters in the cloud. In this paper, we will evaluate the Dask distributed computing framework in respect to various scientific computing tasks. We will look at two different experiment. In the first experiment, we will try to find how the Dask reading file speed, computing metrics (mean, std), finding the unique value, cumulative sum and groupby aggregation differs with other popular Python Frameworks such as Modin(Ray), Vaex. In the second experiment, we will try to find runtime performance comparison of Apache Spark and Dask Big Data engines in processing neuroimaging pipelines.**

I. INTRODUCTION

Data Science is almost indispensable for various industry domains, including marketing, healthcare, finance, banking, policy work, and more. There are many software libraries that tries to make Big Data processing more efficient. One of the most popular software libraries is Dask. This Python package is versatile and supports parallel processing. Dask is made up of two components:

- Optimized for computation dynamic task scheduling. It's comparable to Airflow, Luigi, Celery, or Make, but it's been tailored for interactive computational workloads.
- Standard interfaces like NumPy, Pandas, or Python iterators may work in distributed or larger-than-memory settings thanks to parallel arrays, dataframes, and lists, commonly referred to as "Big Data" collections. Dynamic task schedulers are used in combination with these concurrent collections.

Dask emphasizes the following virtues:

- Familiar: Offers Pandas DataFrame and NumPy Array objects that are parallelized.
- Flexible: Offers a task scheduling interface for more customized workloads and project integration.
- Native: Distributed computing is made possible in pure Python with access to the PyData stack.
- Fast: Low overhead, low latency, and minimum serialization operations that are required for quick numerical algorithms
- Scales up: Operates robustly on clusters with thousands of cores
- Scales down: On a laptop, setting up and running a single process is simple.
- Responsive: It offers quick feedback and diagnostics to help humans and was created with interactive computing in mind.

Dask is convenient on a laptop. It installs trivially with conda or pip and extends the size of convenient datasets from "fits in memory" to "fits on disk". Dask can scale to a cluster of 100s of machines. It is resilient, elastic, data local, and low latency. For more information, see the documentation about the distributed scheduler. This ease of transition between single-machine to moderate cluster enables users to both start simple and grow when necessary.

Dask represents parallel computations with task graphs. These directed acyclic graphs may have arbitrary structure, which enables both developers and users the freedom to build sophisticated algorithms and to handle messy situations not easily managed by the map/filter/groupby paradigm common in most data engineering frameworks. We originally needed this complexity to build complex algorithms for n-dimensional arrays but have found it to be equally valuable when dealing with messy situations in everyday problems.

II. BACKGROUND

"Dask Parallel Computation with Blocked algorithms and Task Scheduling" that was published in 2015 by Matthew Rocklin is one of the main articles used in this paper. Their main focus was to investigate how the performance differs on dask parallel computation with blocked algorithms and task Scheduling. They explores dask graphs, dask arrays, dynamic task scheduling, dask for general computing.

In [1], In comparison to Dask's Bag, Delayed, and Futures, the authors assess PySpark's RDD API. This is the second of the main articles used in this paper in the experiment section.

Furthermore, the works [5, 12] analyze the many elements of Dask distributed computing framework, and how it could be applied to various scientific computing tasks. In [5], the authors explores under the hood, CARS makes use of Dask graph-based and distributed computing of the different steps with lazy evaluation and dynamic task allocation, allowing to run the computation on several tenth of nodes seamlessly.

Finally, work [16, 17] assesses the performance of the Apache Hadoop, Apache Spark, MPI software libraries for large scale data sets.

III. CHARACTERISTICS

In this section, we will look at mainly the characteristics of Dask distributed computing framework. Also, we will discuss briefly the characteristics Python Frameworks such as Modin(Ray), Vaex, and Apache Spark Big Data engine that we used for experiment.

a) Dask distributed computing framework

Python-based Big Data engine Dask is becoming more and more well-liked among scientists. To cut down on data transfers, pointless processing. Dask workflows may make extensive use of multithreading to greatly reduce the cost of data transfer when communication is not constrained by Python's GIL. Dask, in contrast to Spark, uses data lineage rather than coarse-grained operations to provide fault tolerance. Dask is also compact and modular, enabling users to just install the parts they need.

Array, Bag, DataFrame, Delayed, and Futures are the five primary data structures used by Dask. Large arrays are processed using a Dask Array. It gives users access to a distributed copy of the well-known NumPy library. A Dask Bag is a concurrent collection of Python objects, similar to Spark's RDD. It provides an abstraction for programming akin to the PyToolz toolkit. To process a lot of tabular data, a Dask Dataframe is a parallel composite of Pandas Dataframes. Random activities that don't fit in the Array, DataFrame, or Bag APIs are supported by Dask Delayed. Last but not least, Dask Futures enable arbitrary activities similarly to Delayed but execute in real-time rather than idly.

All APIs, with the exception of Dask Bag, use the local multithreaded scheduler by default. Instead, Dask Bag makes use of the regional multiprocessing scheduler. In our experiments, every Dask data structure was used, with the exception of Dask Array and Dask DataFrame.

The internal representation of a Dask application that the scheduler will run is called a Dask graph. The computation graph can more easily describe complicated algorithms because to the way API actions produce several tiny jobs. Similar to Spark, the Dask engine works with a variety of distributed schedulers, including YARN and Mesos. The Dask Distributed scheduler is another distributed scheduler offered by Dask. Despite the fact that Dask is compatible with the schedulers frequently used in HPC clusters, we opt to utilize Dask's Distributed scheduler in order to keep the environment between the two engines balanced. The resources given by cluster workers are managed by a process named daskscheduler in the Dask Distributed scheduler. The scheduler gets jobs from clients and distributes them to workers who are available. Dask Distributed finishes processing a graph branch before going on to the next one, just like Spark's scheduler does.

b) Apache Spark Big Data Engine

A popular general-purpose Big Data engine is Apache Spark. The Resilient Distributed Dataset (RDD) [12], its primary abstraction, is a parallel collection of fault-tolerant data items. Data lineage, or the order of actions performed to alter the original data, is recorded to achieve fault tolerance. The RDD serves as the foundation for Datasets and DataFrames, two further Spark data structures. Similar to RDDs, datasets also make advantage of Spark SQL's optimized execution engine to boost speed. Datasets with named columns are called DataFrames and are used to handle tabular data. Datasets are only supported by Scala and Java, whereas the DataFrame API is present in all available language APIs.

Three schedulers are supported by Spark: Spark Standalone, YARN, and Mesos. A straightforward default scheduler included with Spark is called Spark Standalone. Contrarily, Mesos may be used to plan

a range of different processes, whereas the YARN scheduler is primarily intended to schedule Hadoop-based workflows. We concentrate on Spark's Standalone scheduler since researchers are likely to run their operations in HPC settings without YARN or Mesos installed. The master, the workers (or slaves in Spark), and the driver are the three major processes that make up the Spark Standalone scheduler.The cluster's worker processes provide resources, which the master coordinates. After receiving the application, the driver contacts the master to request employees and assigns duties to them. A task is divided into segments that the staff must finish following a certain methodology. High-level jobs are used on the computation graph to represent each stage's operation.

c) Modin (Ray) Python Framework

You can easily speed up your pandas notebooks, scripts, and libraries using Modin using Ray or Dask. Modin allows easy integration and compatibility with existing pandas code, in contrast to other distributed DataFrame libraries. It is equivalent even when using the DataFrame constructor.

Modin supports inplace semantics, although unlike pandas, its implementation's underlying data structures are immutable. Due to the fact that they cannot be modified, this immutability enables Modin to internally chain operators and better manage memory layouts. Due to the ability to share common memory blocks among all dataframes, this reduces memory usage compared to pandas in many typical situations.

By having a changeable pointer to the immutable internal Modin dataframe, Modin offers the inplace semantics. When an inplace update is performed, Modin will act as though it were not inplace and will only update the pointer to the resultant Modin dataframe because this pointer can change but the underlying data cannot.

d) Vaex Python Framewok

Central to Vaex is the DataFrame (similar, but more efficient than a Pandas DataFrame), and we often use the variable df to represent it. A DataFrame is an efficient representation for a large tabular dataset, and has: a number of columns, say x, y and z, which are backed by a Numpy array; wrapped by an expression system e.g. df.x, df['x'] or df.col.x is an expression; columns/expression can perform lazy computations, e.g. df.x * np.sin(df.y) does nothing, until the result is needed. A set of virtual columns, columns that are backed by a (lazy) computation, e.g. df['r'] = df.x/df.y A set of selections, that can be used to explore the dataset, e.g. df.select(df.x ¡ 0). Filtered DataFrames, that does not copy the data, df-negative = df[df.x ¡ 0]

IV. EVALUATION

That section will look at two different experiments in order to evaluate the Dask distributed computing framework in respect to various scientific computing tasks.

Experiment 1

There are some popular daily tasks in Data Science such as reading file, computing metrics (mean, std), finding the unique value, cumulative sum and groupby aggregation. In the first experiment, we will investigate the Dask againts the Modin(Ray) and Vaex using the mentioned test cases. Experiment 1 scripts are available at https://github.com/abbcyhn/ut-3-seminar.

4.1.1. Experimental Setup

The r/place Parquet dataset 12GB (22GB uncompressed) is used as a dataset for all tests cases. The Pandas has chosen as a baseline for comparison reason. The parameters of test environments are as follows:

- Processor - Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz 1.80 GHz
- Number of cores - 4
- Memory - 16.0 GB
- OS - Ubuntu 16.04.1 LTS
- Programming Language - Python 3.6

4.1.2. Test case 1 - Reading file comparison

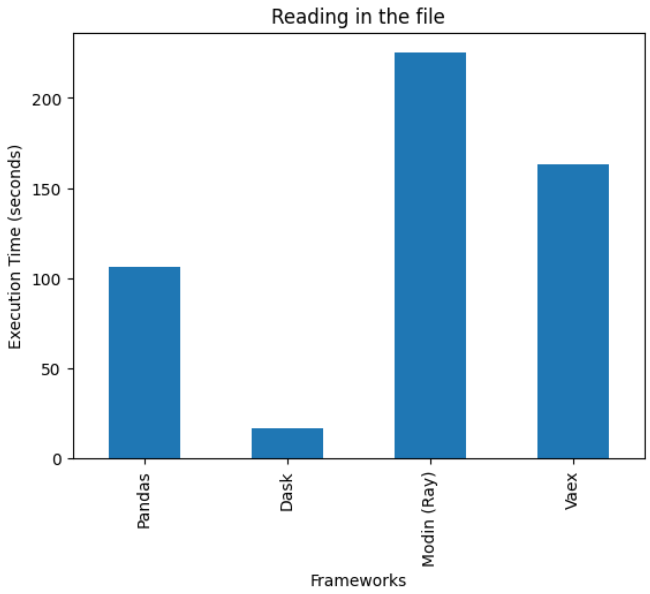


Figure 1: Reading file comparison

The Pandas framework loads the whole data in the memory when dealing with files. However the Dask

uses lazy loading, it means the Dask does not load the whole data in the memory. It is clear that, Dask outperforms in the first test case. The Modin seems the slowest framework for reading files. The Modin can use different engines (Ray, Dask), it tooks some time to setup the engine, and deal with the file. Last but not least, the Vaex python framework handles reading file in reasionable time.

4.1.3. Test case 2 - Compute metrics of a column (mean, std)

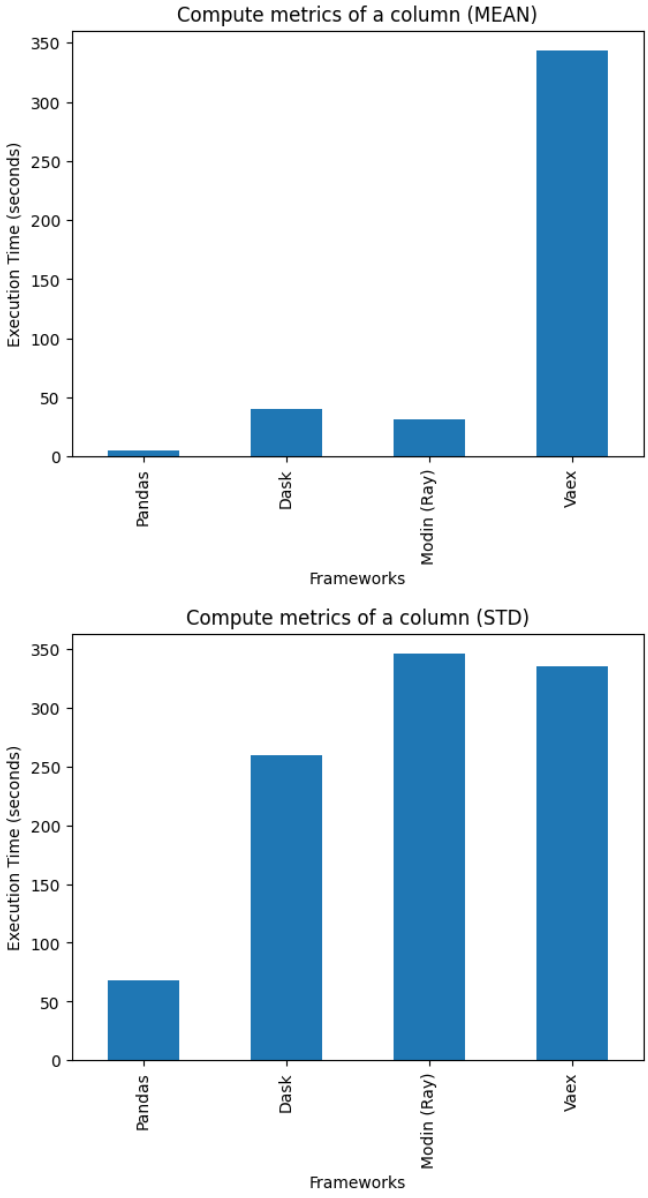


Figure 2: Comparison of computing metrics of a column (mean, std)

Both computing metrics of a column operations, mean and std, the Pandas outperforms other frameworks. The reason is that, the data is in memory, so the Pandas can compute given task more fast and efficiently than other frameworks. In mean computation the Dask and Modin (with Ray engine) are reasionable fast. Vaex seems slower than other frameworks in mean computation. However, Vaex is a little bit faster that Modin when computing standart deviation operation.

4.1.4. Test case 3 - Finding the unique value of a column

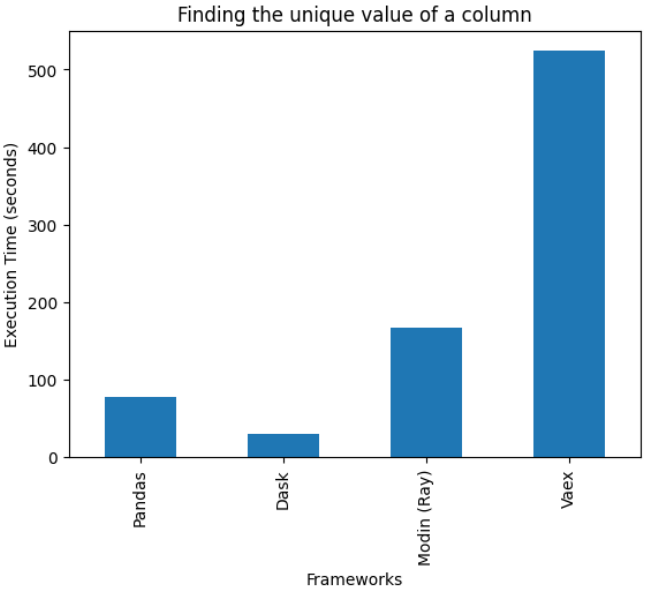


Figure 3: Finding the unique value of a column

In Figure 3, we clearly see that Dask outperforms other libraries, even if Python keeps data in memory and Dask is using lazy evaulation. Again, Modin takes reasionable time for finding the unique value of a column. Vaex seems slower that other frameworks for that particular test case.

4.1.5. Test case 4 - Cumulative sum of a column

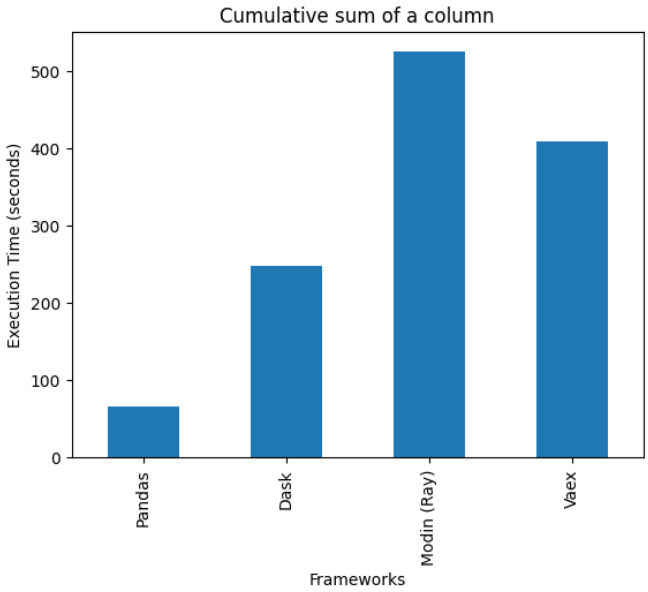


Figure 4: Cumulative sum of a column

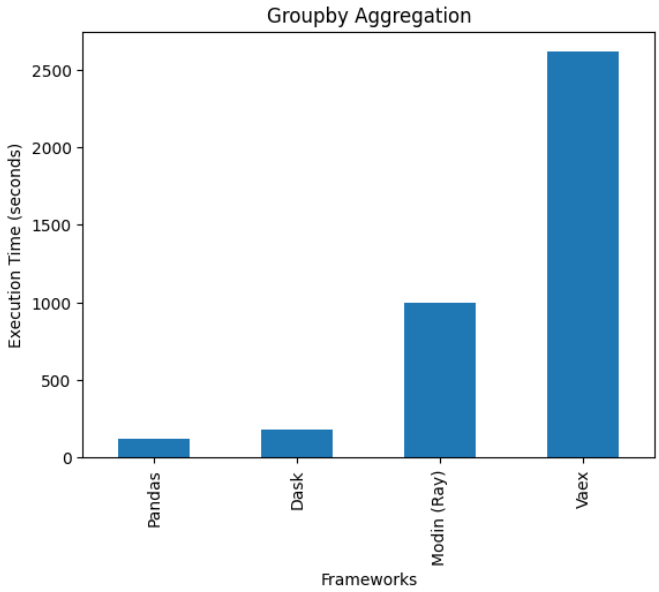


Figure 5: Groupby Aggregation

In Figure 4, we are testing the frameworks againts the cumulative sum operation. Again, Pandas is faster that other Python frameworks. Dask and Vaex take reasonable time for completing the given test case. However, Modin(Ray) seems the slower than others in the particular test case.

4.1.6. Test case 5 - Groupby Aggregation

This is the last test case for the Experiment 1. Groupby Aggregation is most used operation for scientific tasks. Again, we are comparing the Python Frameworks, againts the given test case. In Figure 5, we can see the execution time difference between frameworks when working with groupby aggregation operation. Pandas and Dask shows similar performance for given test case. Modin seems more efficient than Vaex when we are working with groupby aggregation. Last but not least, Vaex again is the slowets one for particular operation.

Experiment 2

This experiments is done by authors of [1]. To assess the engines in various circumstances, they employed three neuroimaging software applications. In this paper, we will discuss only one application. The incrementation application are simple synthetic applications representing map-only application, respectively [15]. Experiment 2 scripts are available at https://github.com/big-data-lab-team/paper-big-data-engines

-	Incrementation App
No of worker	1, 2, 4, 8
No of blocks	30, 125, 750
No of iterations	1, 10, 100
Sleep delay [s]	1, 4, 16, 64

Table 1: Parameters for the experiments [1]

In this experiment, four factors were changed, as shown in Table 1. In order to gauge the impact of various I/O patterns and levels of parallelization, the authors modified (1) the number of workers, (2) the number of BigBrain blocks in Incrementation, (3) the number of iterations and sleep delay in Incrementation, and (4) the job duration and task count.

4.2.1. Experimental Setup

BigBrain [10], a three-dimensional representation of the human brain with voxel intensities ranging from 0 to 65,535, was employed. The MINC has 125 blocks that house the original data. They transformed the blocks into the well-liked NIfTI format for neuroimaging. The NIfTI blocks were not compressed, resulting in an 81GB overall data size. They resplit these blocks

using the sam library into 30, 125, and 750 blocks of 2:7GB, 0:648GB, and 0:108GB to assess the impact of block size.

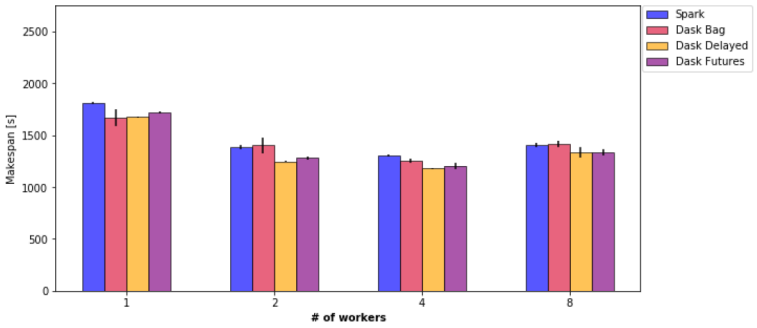
Additionally, they used the dataset made accessible on DataLad by the Consortium for Reliability and Reproducibility (CoRR [11]). The full dataset, which has anatomical, diffusion, and functional pictures of 1,397 patients taken at 29 locations, is 408:4GB in size. All 3,491 anatomical pictures, totaling 39GB, were utilised (11:17MB per image on average). The following are the test environment parameters:

- Processor - Intel Xeon Gold 6130, c8-30gb-186 cloud instances with 8 VCPUs
- Memory - 30 GB at 2666MHz
- OS - CentOS 7.5.1804
- Programming Language - Python

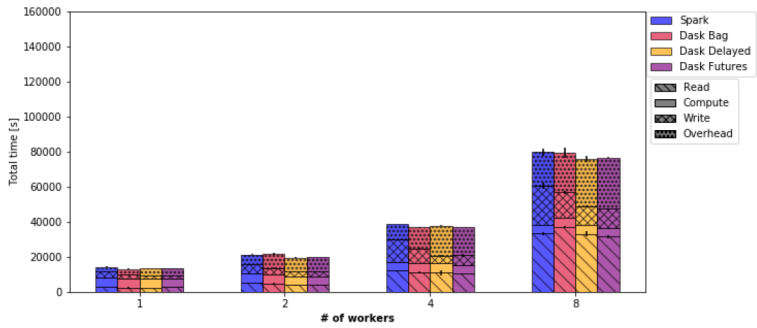
4.2.2. Test Case 1 - Incrementation: Number of workers

The lifespan of the Incrementation application is depicted in Figure 6a for various worker and engine counts. The error bars represent the standard deviations, while the bars represent the average makespan across three repeats. In general, the engines' makespan differences are negligible. Delayed and Futures perform marginally better than Bag, but Dask appears to have a tiny lead over Spark, 83:61 seconds on average. The makespan does not, by any means, decrease linearly with the number of employees for any engine. Even when the makespan grows by 4 to 8 employees. This is because the application is heavily impacted by data transfers and engine overhead.

The Incrementation application's entire execution time, split down into data transfer (read and write), calculate (sleep), and overhead time, as shown in Figure 6b. As anticipated, as the workforce multiplies, the computing time remains constant. However, the number of workers with regression slopes—Spark (2337 s=task), Bag (2650 s=task), Delayed (3251 s=task), and Futures (3570 s=task—increases the data transmission time and overhead correspondingly.



(a) Incrementation makespan



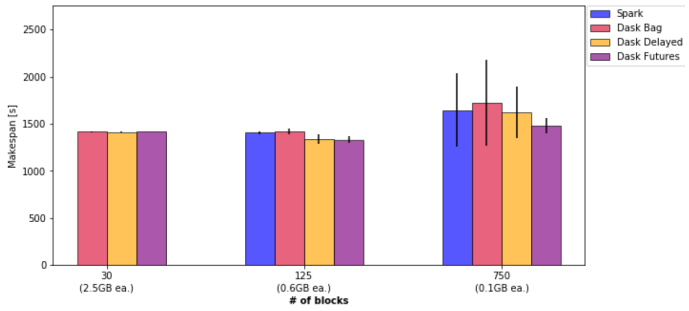
(b) Incrementation total time

Figure 6: 125 blocks, 10 iterations, 4 s delay, 8 CPUs/worker

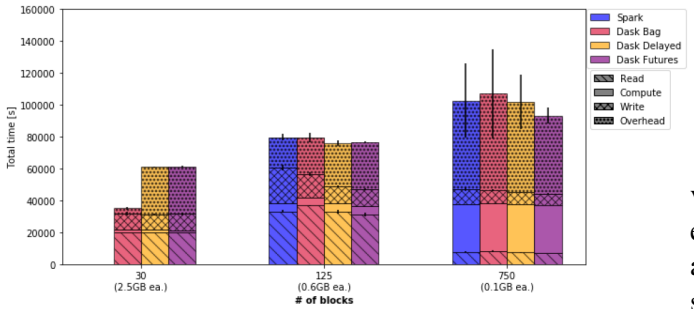
4.2.3. Test Case 2 - Incrementation: Number of blocks

The Incrementation makespan for a constant BigBrain picture size is shown in Figure 7a by altering the number of image blocks. Spark's 2GB job size restriction prevented the writers from running it for 30 blocks. Once more, they don't notice any significant variations amongst the engines. While makespan variability rises for all engines as the number of blocks grows, in general, engines scale fairly well.

The entire execution time for each function is displayed in Figure 7b. The fact that only 30 of the 64 available threads may be utilized concurrently causes the higher overhead of Delayed and Futures for 30 blocks. Again, when additional blocks are added, the data transmission time decreases yet the overhead time also increases. Due to the workers not being utilized to their maximum potential, or because some threads are idle, this is not detected for 30 blocks. The makespan variability described before is explained by the fact that the overhead variability rises with the number of blocks.



(a) Incrementation makespan



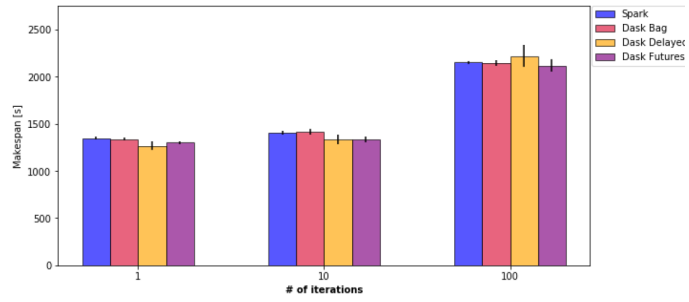
(b) Incrementation total time

Figure 7: 10 iterations, 4 s delay, 8 workers, 8 CPUs/worker

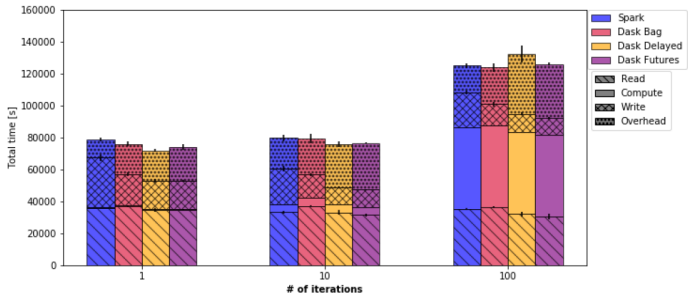
4.2.4. Test Case 3 - Incrementation: Number of iterations

The lifespan of the application is depicted in Figure 8a when the number of iterations is changed. Overall, the Dask and Spark APIs are once again equal, while Delayed and Futures are a little quicker than Bag and Spark for 1 and 10 iterations and Futures are faster than Delayed, Bag, and Spark for 100 iterations. The distinctions are still slight.

The overall execution time breakdown is displayed in Figure 8b. The authors find that all of the engines scale well as iterations increase.



(a) Incrementation makespan



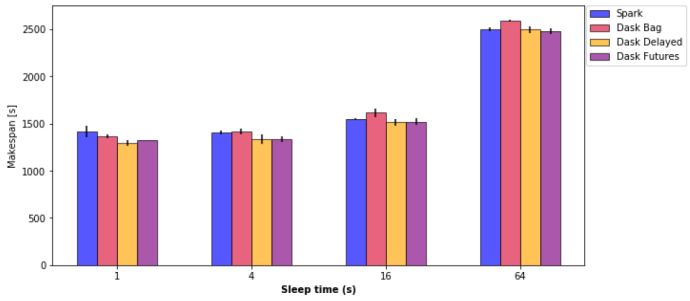
(b) Incrementation total time

Figure 8: 125 blocks, 4 s delay, 8 workers, 8 CPUs/worker

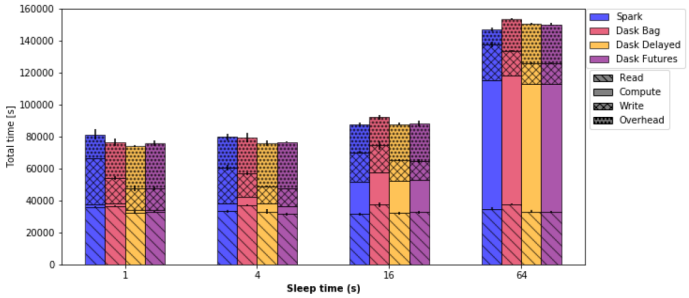
4.2.5. Test Case 4 - Incrementation: Sleep delay

The makespan of the Incrementation application for various sleep delays is shown in Figure 9a. Overall, all engines once more exhibit the same performance and adapt well to work length. Although Spark starts out slower than the Dask APIs, it quickly becomes quicker. Additionally, Dask Bag is somewhat slower than the other two Dask APIs, albeit not noticeably so.

The breakdown of total execution time is shown in Figure 9b. The overhead for Spark is the least. As was previously noted, changes in data transmission time virtually perfectly offset changes in overhead time.



(a) Incrementation makespan



(b) Incrementation total time

Figure 9: 125 blocks, 10 iterations, 8 workers, 8 CPUs/worker

V. CONCLUSIONS

In this paper, we evaulated the Dask in respect to various scientific computing tasks. We investigated two different experiment. In the first experiment, we find how the Dask reading file speed, computing metrics (mean, std), finding the unique value, cumulative sum and groupby aggregation differs with Modin(Ray), Vaex. The results of first experiment show Dask performs better that other frameworks considering execution times. In the second experiment, we spoke about contrasting Apache Spark and Dask, two big data engines. We looked at the engines for data-intensive neuroimaging applications that increment data. In general, the outcomes of the second trial indicate that there is little to no performance difference between the engines. It's interesting to note that variations in engine overheads have little effect on performance since they have no effect on data transfer time. When data transfers saturate the bandwidth, larger overheads are almost exactly offset by a shorter transfer time.

REFERENCES

[1] M. Dugré, V. Hayot-Sasson and T. Glatard, "A Performance Comparison of Dask and Apache Spark for Data-Intensive Neuroimaging Pipelines," 2019 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS), 2019, pp. 40-49, doi: 10.1109/WORKS49585.2019.00010.

[2] S. Böhm and J. Beránek, "Runtime vs Scheduler: Analyzing Dask's Overheads," 2020 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS), 2020, pp. 1-8, doi: 10.1109/WORKS51914.2020.00006.

[3] Mehta, Parmita and Dorkenwald, Sven and Zhao, Dongfang and Kaftan, Tomer and Cheung, Alvin and Balazinska, Magdalena and Rokem, Ariel and Connolly, Andrew and Vanderplas, Jacob and AlSayyad, Yusra. (2016). Comparative Evaluation of Big-Data Systems on Scientific Image Analytics Workloads. Proceedings of the VLDB Endowment. 10. 10.14778/3137628.3137634.

[4] Hernández, B. et al. (2020). Performance Evaluation of Python Based Data Analytics Frameworks in Summit: Early Experiences. In: Nichols, J., Verastegui, B., Maccabe, A.'., Hernandez, O., Parete-Koon, S., Ahearn, T. (eds) Driving Scientific and Engineering Discoveries Through the Convergence of HPC, Big Data and AI. SMC 2020. Communications in Computer and Information Science, vol 1315. Springer, Cham. https://doi.org/10.1007/978-3-030-63393-6-24

[5] D. Youssefi et al., "CARS: A Photogrammetry Pipeline Using Dask Graphs to Construct A Global 3D Model," IGARSS 2020 - 2020 IEEE International Geoscience and Remote Sensing Symposium, 2020, pp. 453-456, doi: 10.1109/IGARSS39084.2020.9324020.

[6] Ioannis Paraskevakos, Andre Luckow, Mahzad Khoshlessan, George Chantzialexiou, Thomas E. Cheatham, Oliver Beckstein, Geoffrey C. Fox, and Shantenu Jha. 2018. Task-parallel Analysis of Molecular Dynamics Trajectories. In Proceedings of the 47th International Conference on Parallel Processing (ICPP 2018). Association for Computing Machinery, New York, NY, USA, Article 49, 1–10. https://doi.org/10.1145/3225058.3225128

[7] A. Shafi, J. M. Hashmi, H. Subramoni and D. K. D. Panda, "Efficient MPI-based Communication for GPU-Accelerated Dask Applications," 2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid), 2021, pp. 277-286, doi: 10.1109/CCGrid51090.2021.00037.

[8] Elibol, Melih and Benara, Vinamra and Yagati, Samyu and Zheng, Lianmin and Cheung, Alvin and Jordan, Michael and Stoica, Ion. (2022). NumS: Scalable Array Programming for the Cloud. 10.48550/arXiv.2206.14276.

[9] J. Crist, "Dask and Numba: Simple libraries for optimizing scientific python code," 2016 IEEE International Conference on Big Data (Big Data), 2016, pp. 2342-2343, doi: 10.1109/BigData.2016.7840867.

[10] K. Amunts, C. Lepage, L. Borgeat, H. Mohlberg, T. Dickscheid, M.E.Rousseau, S. Bludau, P.L. Bazin, L. B. Lewis, A.M.OrosPeusquens, 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.

[11] 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

[12] T. E. Oliphant, "Python for Scientific Computing," in Computing in Science and Engineering, vol. 9, no. 3, pp. 10-20, May-June 2007, doi: 10.1109/MCSE.2007.58.

[13] Shafi, Aamir and Hashmi, Jahanzeb and Subramoni, Hari and K., Dhabaleswar and Panda,. (2021). Efficient MPI-based Communication for GPU-Accelerated Dask Applications.

[14] Kumar, Deepa and Rahman, M. (2017). Performance Evaluation of Apache Spark Vs MPI: A Practical Case Study on Twitter Sentiment Analysis. Journal of Computer Science. 13. 781-794. 10.3844/jcssp.2017.781.794.

[15] Ahmed, Nasim and Barczak, Andre and Susnjak, Teo and Rashid, Mohammad. (2020). A Comprehensive Performance Analysis of Apache Hadoop and Apache Spark for Large Scale Data Sets Using HiBench. 10.21203/rs.3.rs-43526/v1.