# Large-scale Computing Frameworks: Experiments and Guidelines

Michele Baglioni[1], Fabrizio Montecchiani[1,*] and Mario Rosati[2]

[1]*Dipartimento di Ingegneria, Università degli Studi di Perugia, Italy*
[2]*E4 Analytics, 67039 Sulmona (AQ), Italy*

Abstract

Large-scale computing frameworks are key technologies to fulfill the computational requirements of massive data analysis. In particular, while Apache Spark has emerged as de facto standard for big data analytics after Hadoop's MapReduce, tools such as Dask and Ray can greatly boost the performance of Python applications in distributed environments. The goal of this paper is to study the performance of these three frameworks on a common playground. We focus on cloud-native architectures, which merge the benefits of big data and cloud computing. We refrain from considering high-level features such as ML models, we instead consider simple data processing operations, common ingredients of more complex pipelines. As a byproduct of our experiments, we offer a set of guidelines for the development of cloud-native data processing applications.

**Keywords**

Big data engines, Apache Spark, Dask, Ray, Kubernetes

## 1. Introduction

Massive data processing pipelines running on single-node machines or large clusters can benefit of computing frameworks that hide low-level operations needed to parallelize the workload. In this regard, so-called Big Data engines play a crucial role as they make it simple to write applications that transparently scale with the underlying infrastructure. For instance, Apache Spark[1] has emerged as the de facto standard for big data analytics after Hadoop's MapReduce [1]. It offers an open-source unified analytics engine for large-scale data processing, with transparent fault-tolerance. Spark embraces key principles such as in-memory computing, data locality and lazy evaluation, in order to achieve great speedups and strong scalability. While Spark has a powerful set of programming interfaces for multiple languages, other projects aim for a seamless and transparent scaling of Python applications. Among them, Dask[2] is a newer open-source library to scale Python code, providing a familiar user interface [2]. Indeed, it mirrors the APIs of other popular libraries such as Pandas, scikit-learn and NumPy. Similarly,

*ITADATA2023: The 2nd Italian Conference on Big Data and Data Science, September 11–13, 2023, Naples, Italy*
*Corresponding author.
michele.baglioni1@studenti.unipg.it (M. Baglioni); fabrizio.montecchiani@unipg.it (F. Montecchiani); mario.rosati@e4-analytics.com (M. Rosati)
0000-0002-0877-7063 (F. Montecchiani)

[1]https://spark.apache.org/
[2]https://www.dask.org/

Ray[3] is an open-source compute framework to scale Python applications, introduced only about 5 years ago, with a focus on reinforcement learning, deep learning, hyperparameters tuning, and model serving [3]. We point the reader to the official documentations of these three frameworks for additional information and details. Also, some design features will be discussed in Section 2.

The growth of these technologies has been accompanied by the growth of cloud-native applications, which are increasingly studied in research and adopted in industry. Cloud-native applications consists of several processes that run in isolated containers, which are typically spread over the nodes of a cluster. In this respect, Kubernetes[4] is becoming the standard technology to automate software deployment, scaling, and management through containers orchestration [4].

Given the popularity of Spark, Dask, and Ray, and the importance of making informed choices in terms of technology, the goal of this paper is to study the performance of these three frameworks on a common playground. We use a cloud-native environment based on Kubernetes, and we experiment standard data processing pipelines fed by a large dataset consisting of numerical values. In particular, while we are well aware of the differences and peculiarities in terms of high-level APIs offered by the considered frameworks, our focus is on standard preprocessing operations that are typically performed in any data-processing pipeline. We do not consider high-level features such as ML/DL models as they require diverse datasets and specialized hardware (such as suitable GPUs), to be conveniently evaluated. As a byproduct of our experiments, we offer a set of guidelines for the development of cloud-native data-processing applications. To the best of our knowledge, this is the first paper that experimentally compares these three frameworks. The only similar comparison we are aware of is restricted to Dask and Spark, and it focuses on neuroimaging pipelines [5].

**Paper structure.** A detailed description of the experimental setting can be found in Section 2. Results and guidelines are discussed in Section 3 and Section 4, respectively. Section 5 concludes the paper with future research directions.

## 2. Experimental Setting

In this section we describe the infrastructure and the benchmark used to run the experiments.

### 2.1. Infrastructure

**Hardware.** In terms of hardware, the experiments were conducted on a Dell PowerEdge XE8545 server, designed to take advantage of the industry's most advanced technologies. The server is equipped with two sockets and has a 4U format, with two 128-core AMD EPYC processors, NVIDIA A100 Tensor Core GPUs and 1024 GB of RAM. As for storage, it has 4 Dell Ent NVMe CM6 disks of 2TB each. See also Table 1 for full references. Furthermore, it is equipped with NVLink, PCIe Gen 4.0 and NVMe SSD to improve I/O and data processing performance. These technologies speed up data transfer between the CPU and GPU, improving

[3]https://www.ray.io/
[4]https://kubernetes.io/

the overall efficiency of the system. The server is certified by NVIDIA for its performance, manageability, security and scalability.

| | |
|---|---|
| **Processor** | Dual AMD EPYC 7713, 64C, 2.8 GHz |
| **Memory** | 1024 GB (16x64GB @ 3200 MT/s) |
| **GPUs** | 4 x NVIDIA A100 SXM4 80 GB |
| **Operating System** | Linux Ubuntu Server 20.04 LTS |
| **Storage** | 4 x Dell Ent NVMe CM6 RI 1.92TB 2.1.8 |

**Table 1**
Dell PowerEdge XE8545 hardware specifications.

**Kubernetes.** We deployed a bare-metal single-node Kubernetes cluster on the previously described server. We remark that the scheduling and resource allocation process is a critical aspect for such experiments [6]. To face this issue, we employed Volcano[5], an add-on for running HPC workloads on Kubernetes. In particular, different from Kubernetes, Volcano features gang scheduling[6], which ensures that each job will run at “full speed”, that is, it will start to run only when all its tasks are ready to be deployed. The adopted scheduler configuration is reported in Listing 1.

Listing 1: Volcano scheduler configuration.

```
apiVersion: v1
date:
   volcano-scheduler.conf: |
     actions: "enqueue, allocate, backfill"
     tiers:
     - plugins:
       - name: priority
       - name: gang
       - name: compliance
kind: ConfigMap
metadata:
   creationTimestamp: "2023-02-15T09:16:47Z"
   name: volcano-scheduler-configmap
   namespace: volcano-system
```

For each framework a specific queue has been defined, as reported in Listing 2, in order to decouple the workloads of the different frameworks; see also Fig. 1 for a schematic illustration.

Listing 2: Volcano queue definition.

```
apiVersion: scheduling.volcano.sh/v1beta1
kind: Queue
metadata:
   name: test-spark
```

[5]https://volcano.sh/en
[6]https://volcano.sh/en/docs/plugins/

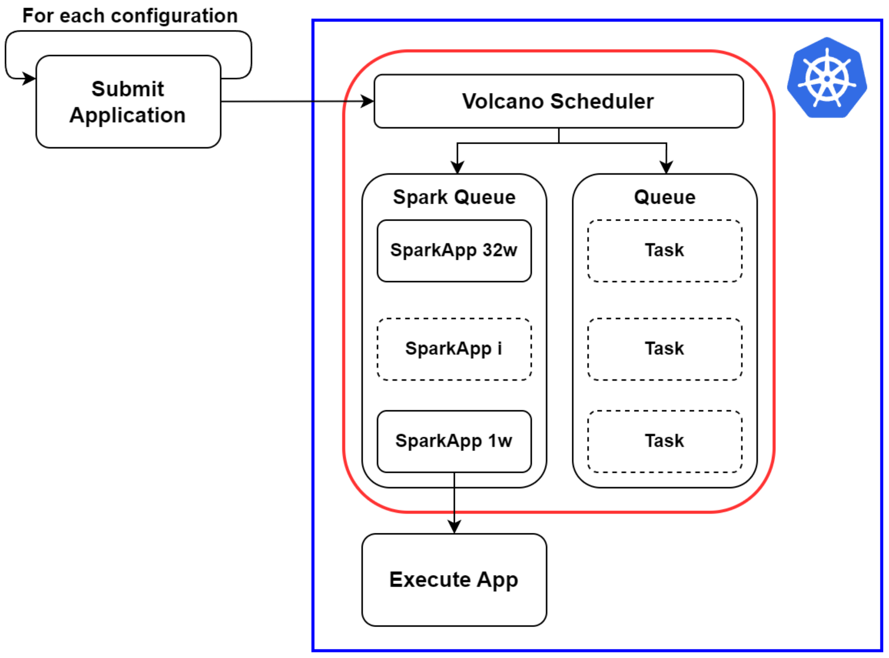

**Figure 1:** Volcano queue for Spark applications.

```
    namespaces: default
specs:
    weight: 1
    capabilities:
      cpu: 32
      memory: 512Gi
```

Finally, to ensure reproducibility, Table 2 provides all the versions and configurations used for the entire software stack.

**Storage.** Read and write operations are fundamental in the context of data processing. Therefore, it is essential to configure the system to operate in the best conditions. In particular, we implemented a scalabale object storage system via MinIO[7] within the Kubernetes cluster. This system consists of 4 physical disks for a total of 7 TiB of available space.

## 2.2. Benchmark

**Dataset.** The dataset was generated synthetically by sampling random values in the range $[1, 10^3]$ from a uniform probability distribution. The dataset has $1.6 \cdot 10^9$ rows and a schema, represented in Table 3, where there are three columns named a, b, and c. The first column contains integers, while the other two columns contain decimal numbers represented as doubles.

[7]https://min.io/

| | |
|---|---|
| **k8s client** | v1.24.10 |
| **k8s server** | v1.24.9+rke2r2 |
| **minio** | 12.2.0 |
| **volcano** | 1.6.0 |
| **spark** | 3.1.1 |
| **pyspark** | 3.1.1 |
| **spark-operator** | gcp:v1beta2-1.3.8-3.1.1 |
| **ray-operator** | 0.4.0 |
| **ray** | 2.2.0 |
| **dask-operator** | 2023.1.1 |
| **dask** | 2023.3.1 |
| **dask_kubernetes** | 2023.3.0 |
| **pandas** | 1.5.3 |
| **numpy** | 1.24.2 |
| **lz4** | 4.3.2 |
| **msgpack** | 1.0.5 |
| **pyarrow** | 11.0.0 |
| **s3fs** | 2023.3.0 |
| **openblas** | 0.3.17 |
| **num_threads (openblas)** | 1 |

**Table 2**
Versions of the components of the software stack.

| | a | b | c |
|---|---|---|---|
| 0 | int | double | double |
| ... | ... | ... | ... |
| $1.6 \cdot 10^9$ | int | double | double |

**Table 3**
Schema of the dataset.

**Format.** As for the format in which to store the dataset, we used Apache Parquet [7]. It is a self-describing data format that embeds the schema into the data itself. It supports efficient encoding and compression schemes that help lower data storage costs and maximize the effectiveness of data queries. Parquet has additional advantages, such as storing data in compressed form using Snappy [8], a method developed by Google and designed for space and query efficiency. The dataset was divided into 160 blocks. The archived dataset has a size of 13.8 GB, and, once loaded into memory, it occupies about 39 GB. Dividing the dataset into blocks has the advantage of facilitating the parallelization of data processing operations on a cluster, as each block can be processed independently of the others. The block count of 160 equals the least common multiple of the number of worker nodes in all configurations evaluated. This ensures that, in all cluster configurations we evaluated, the workers can perform their read operations concurrently.

**Operations.** At high-level, the benchmark consists of reading data from the distributed storage, manipulating the data with elementary functions, and finally writing the data back in the storage; see Fig. 2 for an illustration. The pipeline is therefore composed of the following

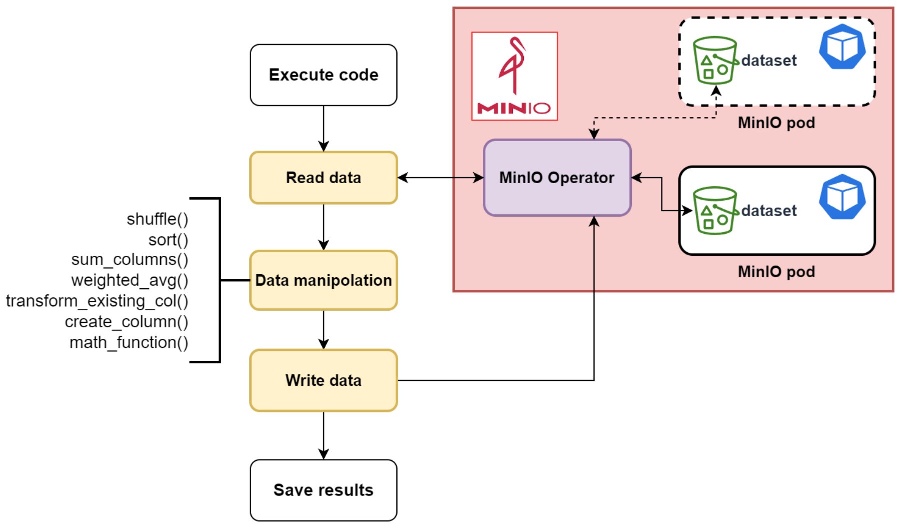

**Figure 2:** Schematic illustration of the performed operations.

operations:

- *Read:* the data are loaded into the main memory, performing a read operation from the distributed storage.
- *Sort:* the data items are sorted in ascending order based on the value of a specific column.
- *Shuffle:* the data items are randomly shuffled.
- *Create new column:* a new column of is created from other existing columns, for example to combine two columns or extract information from one column.
- *Transform of a column:* an existing column is transformed to make it more suitable for subsequent processing.
- *Sum between two columns:* the values of two columns are added together, generating a new column to store the result.
- *Weighted average:* the weighted average of the data items is calculated on the basis of certain weights assigned to each item.
- *Math function:* the following math function is applied to the values of multiple existing columns: $\cos(col1) * \arctan(col2) + \log(col3)$
- *Write:* Processed data is written as a new object into the distributed storage.

**Applications.** In order to support reproducibility of the experiments, we adopted the Kubernetes Operator Pattern [4, 9], which makes it possible to integrate domain knowledge into Kubernetes' orchestration process. In order to implement this pattern, we first define new types of resources that the Kubernetes API can manage (called CRDs); next, we add the operator, which is a software component running inside the cluster in order to manage the entire lifecycle of the CRDs. In particular, the operator interacts with the Kubernetes API and reacts to creation, modification, or removal of custom resources (CRs). Therefore, for each of the three frameworks,

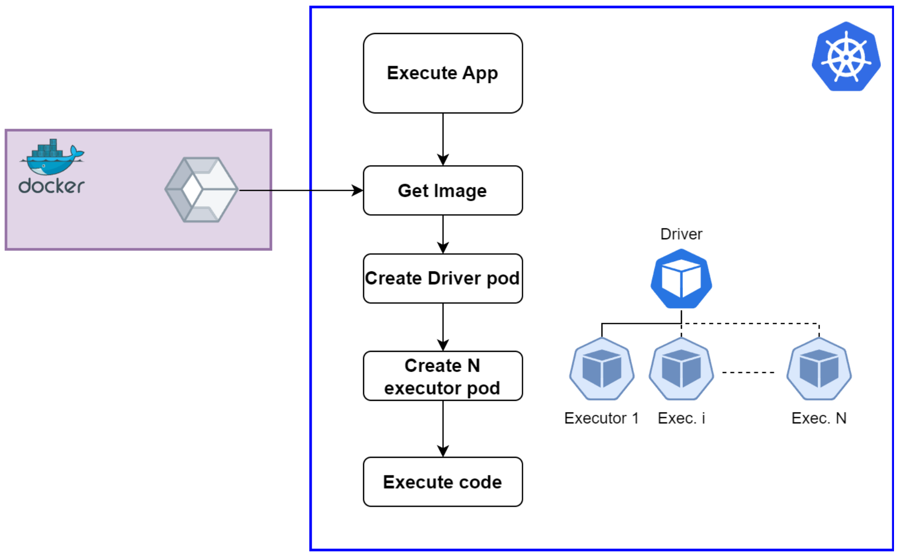

**Figure 3:** Deployment of a Spark application.

the respective operators were installed via Helm chart in the Kubernetes cluster. Fig. 3 shows how a Spark application is deployed and launched in the Kubernetes cluster. As it will be clarified later, each app is executed with a number $N$ of workers, with $N$ ranging between 1 and 32. Listing 3 shows one of the YAML files used to define Spark applications. The applications are deployed and launched similarly for the other frameworks. The only exception is that Dask does not provide a direct integration with the Volcano scheduler, hence we relied on a Volcano job to launch Dask applications.

Listing 3: Definition of a Spark Application.

```
apiVersion: "sparkoperator.k8s.io/v1beta2"
kind: SparkApplication
metadata:
  name: benchmark
  namespace: default
spec:
  type: Python
  mode: cluster
  image: "docker.io/docker-repo-name/benchmark-spark:latest"
  imagePullPolicy: Always
  pythonVersion: "3"
  mainApplicationFile:
    "local:///opt/spark/benchmark/data-processing.py"
  sparkVersion: "3.1.1"
  batchScheduler: "volcano"
  batchSchedulerOptions:
    queue: "test-spark"
    resources:
      cpu: 32
      memory: ”512G”
  restartPolicy:
    type: OnFailure
  driver:
    cores: 1
    memory: ”16g”
    labels:
      version: 3.1.1
    serviceAccount: spark
  executor:
    cores: 1
    instances: 32
    memory: ”16g”
    labels:
      version: 3.1.1
  deps:
    jars:
      - https://repo1.maven.org/maven2/org/apache/hadoop/
        hadoop-aws/3.2.0/hadoop-aws-3.2.0.jar
      - https://repo1.maven.org/maven2/com/amazonaws/aws-java
        sdk-bundle/1.11.375/aws-java-sdk-bundle-1.11.375.jar
    pyFiles:
      - local:///opt/spark/benchmark/transformations.py
```

## 3. Experimental Results

**Metrics and procedure.** In order to assess the performance of the experimented frameworks, besides the runtime, we considered both speedup and efficiency, briefly recalled below. Let $N$ be the number of available processors, let $t_i$ be the time taken to process the workload with $i \in [1, N]$ processors, then the metrics are defined as follows (a linear speedup and constant efficiency are ideal targets, see, e.g., [10]).

$$\text{speedup } S(i) = \frac{t_1}{t_i} \tag{1}$$

$$\text{efficiency } E(i) = \frac{S(i)}{i} \tag{2}$$

Each framework has been evaluated with a number of workers equal to $2^w$, with $w$ ranging in $[0, 5]$. The results of the various executions are extracted from the logs of the respective master node's pod. For the sake of reliability, the results are averaged over 10 executions of the same application with the same workers configuration.

**Overall performance.** We being by remarking that, given a pipeline of operations, each framework adopts a specific strategy to break the workload into smaller tasks and to assign the

| | N | Time (s) | Speedup | Efficiency |
|---|---|---|---|---|
| Spark | **1** | 7350 | 1.0 | 1.0 |
| | **2** | 3719 | 1.98 | 0.99 |
| | **4** | 1908 | 3.85 | 0.96 |
| | **8** | 1053 | 6.98 | 0.87 |
| | **16** | 627 | 11.72 | 0.73 |
| | **32** | 450 | 16.33 | 0.51 |
| Ray | **1** | 5850 | 1.0 | 1.0 |
| | **2** | 2874 | 2.04 | 1.02 |
| | **4** | 1660 | 3.52 | 0.88 |
| | **8** | 1095 | 5.34 | 0.67 |
| | **16** | 547 | 10.69 | 0.67 |
| | **32** | 290 | 20.17 | 0.63 |
| Dask | **1** | 5841 | 1.0 | 1.0 |
| | **2** | 3347 | 1.75 | 0.88 |
| | **4** | 2010 | 2.91 | 0.73 |
| | **8** | 1274 | 4.58 | 0.57 |
| | **16** | 843 | 6.93 | 0.43 |
| | **32** | 581 | 10.05 | 0.31 |

**Table 4**
Overall performance. The symbol N denotes the number of workers.

tasks to the workers. In particular, triggering the execution of single operations to measure each single run time would hinder such optimization strategies and invalidate the results. Therefore, we measure the run time of the whole pipeline, such that each framework can fully exploit its own optimization strategy. The run time includes the time needed to read the data, which will also be evaluated separately in the next paragraph.

Table 4 reports the results of the experiments. In terms of overall execution time, Ray appears to be the fastest framework, with the best performance in all worker configurations except in the single worker setting, in which Dask is only a few seconds faster. Between Dask and Spark, we see that Dask outperforms Spark up to 2 workers, while Spark performs better between 4 and 32 workers. In terms of speedup and efficiency, Spark exhibits very good performance up to 16 workers, showing an efficiency drop with 32 workers. The Ray framework shows better scalability, with a smaller drop in terms of efficiency. On the other hand, Dask exhibits low scalability compared to the other two frameworks, its efficiency rapidly decreases as the number of workers increases.

**Reading performance.** In this paragraph we analyze the performance of the frameworks in terms of reading performance, that is, the time taken to read the data from the object storage and to load the data into the main memory. Table 5 reports the results of our experiments. Again, for the single worker scenario, Dask is the fastest framework, and it remains the fastest also with 2 workers. With 4 workers Spark becomes the fastest framework, and it remains the fastest also with 8 workers, together with Ray. With 16 and 32 workers, Ray is the fastest framework, with Spark slightly slower. In terms of speedup, again Ray and Spark outperform Dask, which also exhibits the worse behavior in terms of efficiency.

| | N | Time (s) | Speedup | Efficiency |
|---|---|---|---|---|
| Spark | **1** | 338 | 1.0 | 1.0 |
| | **2** | 165 | 2.05 | 1.02 |
| | **4** | 87 | 3.89 | 0.97 |
| | **8** | 50 | 6.76 | 0.84 |
| | **16** | 31 | 10.9 | 0.68 |
| | **32** | 25 | 13.52 | 0.42 |
| Ray | **1** | 381 | 1.0 | 1.0 |
| | **2** | 200 | 1.91 | 0.95 |
| | **4** | 100 | 3.81 | 0.95 |
| | **8** | 49 | 7.78 | 0.97 |
| | **16** | 25 | 15.24 | 0.95 |
| | **32** | 12 | 31.75 | 0.99 |
| Dask | **1** | 184 | 1.0 | 1.0 |
| | **2** | 141 | 1.3 | 0.65 |
| | **4** | 112 | 1.64 | 0.41 |
| | **8** | 92 | 2.0 | 0.25 |
| | **16** | 85 | 2.16 | 0.14 |
| | **32** | 77 | 2.39 | 0.07 |

**Table 5**
Read performance. The symbol N denotes the number of workers.

## 4. Discussion, Limitations and Guidelines

**Discussion.** We begin by briefly discussing possible relations between the design principles behind the three frameworks and the observed performance. Spark has been designed to handle large amounts of data and complex operations, using distributed data structures to keep data in memory and minimize disk access operations. In particular, its ability to optimize operations through its distributed execution engine, dividing the operations to be performed into tasks and distributing them efficiently on the nodes of the cluster, makes it able to scale very well with configurations with high availability of resources, while it appears less efficient with few workers. Ray is a system designed to handle high-speed, low-latency distributed processing operations. To achieve this goal, efficient communication between nodes and an actor-based programming model are used, which allows operations to be distributed asynchronously, minimizing workers' downtime. This architecture makes it very efficient in all configurations. Dask uses a data partitioning strategy based on blocks of variable size, which allows to better adapt to the size of the dataset and to minimize data movements between nodes. With few workers, Dask's data partitioning strategy proves to be particularly effective, as it allows you to make the most of the available resources, minimizing execution times. However, as the number of worker nodes increases, coordinating distributed operations becomes more complex, and Dask may show limits in terms of scalability compared to Spark and Ray. In particular, Dask can suffer from increased overhead for coordinating distributed operations and managing distributed memory, which can slow performance on large clusters.

**Limitations.** Our experiments where performed on a highly optimized infrastructure, however, the performance of the frameworks are affected by the use of containers. For instance, it was shown that Kubernetes pods may deteriorate data locality, and make a worse usage of memory and CPU [11]. Also, our experiments did not measure the resiliency of the three frameworks, nor their performance when some tasks are aborted or delayed. Our results should not be generalized to larger clusters, in particular with multiple racks in which network bottlenecks may arise. Similarly, much larger datasets, in the order of Terabytes, may lead to different behaviours of the frameworks. Also, we only considered simple data processing operations, while all the frameworks have advanced libraries to train and use ML/DL models. Since the performance of these advanced features strongly depend on their implementation and on the availability of specialized hardware, we cannot draw any conclusion about model training from our experiments.

**Guidelines.** Based on our findings and with the above limitations in mind, Ray appears to be the most efficient framework over all configurations. On the other hand, Dask is the fastest framework with only one worker, but it is less scalable than Spark and Ray. Spark performs somewhere in between, with performance improving as nodes increase. In the case of read-intensive applications, Dask seems a good choice if only few nodes are available, while Spark or Ray would be recommended with a large number of workers. Besides performance, the ecosystem is an important aspect to consider when choosing a technology over another. In this respect, Spark stands out for its reliability, as it is certainly a well-tested framework, with an established community of developers offering a wide range of resources, including documentation and reusable code. On the other hand, Ray allows you to run Spark and Dask code on the infrastructure managed by the Ray Core, thus making it possible to use particular features of other frameworks or existing legacy code. Another key aspect when staring out a project with a new technology is the required learning curve. In this regard, Dask's learning curve is rather shallow for Python developers, with the possibility of using the same syntax of the most common libraries, such as Pandas and Numpy.

## 5. Future Work

We plan to extend our experiments by considering larger datasets and more advanced features offered by the three frameworks. In particular, we would like to compare their performance in terms of model training. To this aim, we plan to equip our cluster with suitable GPUs, which represent the current standard technology in this field. This type of comparison should involve different ML/DL models and hence multiple datasets, for instance, including images and text.

## References

[1] S. Salloum, R. Dautov, X. Chen, P. X. Peng, J. Z. Huang, Big data analytics on apache spark, Int. J. Data Sci. Anal. 1 (2016) 145–164. URL: https://doi.org/10.1007/s41060-016-0027-9. doi:10.1007/s41060-016-0027-9.

[2] M. Rocklin, Dask: Parallel computation with blocked algorithms and task scheduling, in: SciPy, scipy.org, 2015, pp. 126–132.

[3] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, I. Stoica, Ray: A distributed framework for emerging AI applications, in: OSDI, USENIX Association, 2018, pp. 561–577.

[4] B. Ibryam, R. Huß, Kubernetes Patterns: Reusable Elements for Designing Cloud-native Applications, O'Reilly Media, 2019. URL: https://books.google.it/books?id=Ax53wgEACAAJ.

[5] M. Dugré, V. Hayot-Sasson, T. Glatard, A performance comparison of dask and apache spark for data-intensive neuroimaging pipelines, in: WORKS@SC, IEEE, 2019, pp. 40–49.

[6] P. Liu, J. Guitart, Fine-grained scheduling for containerized HPC workloads in kubernetes clusters, in: HPCC/DSS/SmartCity/DependSys 2022, IEEE, 2022, pp. 275–284. doi:10.1109/HPCC-DSS-SmartCity-DependSys57074.2022.00068.

[7] K. Sharma, U. Marjit, U. Biswas, Efficiently processing and storing library linked data using apache spark and parquet, Information Technology and Libraries 37 (2018) 29–49.

[8] J. Janet, S. Balakrishnan, E. R. Prasad, Optimizing data movement within cloud environment using efficient compression techniques, in: 2016 International Conference on Information Communication and Embedded Systems (ICICES), 2016, pp. 1–5. doi:10.1109/ICICES.2016.7518896.

[9] S. Henning, B. Wetzel, W. Hasselbring, Reproducible benchmarking of cloud-native applications with the kubernetes operator pattern, in: SSP, volume 3043 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2021.

[10] D. Eager, J. Zahorjan, E. Lazowska, Speedup versus efficiency in parallel systems, IEEE Transactions on Computers 38 (1989) 408–423. doi:10.1109/12.21127.

[11] C. Zhu, B. Han, Y. Zhao, A comparative performance study of spark on kubernetes, J. Supercomput. 78 (2022) 13298–13322. URL: https://doi.org/10.1007/s11227-022-04381-y. doi:10.1007/s11227-022-04381-y.