External Scheduler: Ray

Ingmar Biemond*
2725600
i.biemond@student.vu.nl

Simon Bezemer*
2731853
s.bezemer2@student.vu.nl

Maurits Reijnaert*
2739377
m.b.reijnaert@student.vu.nl

Sacheendra Talluri
Support Cast
s.talluri@vu.nl

Alexandru Iosup
Support Cast
a.iosup@vu.nl

ABSTRACT

Context. Efficient scheduling in distributed systems is important for optimizing computational resources and reducing processing times.
Goal. This study will explore the possibility of the use of an external scheduler in distributed systems, and measure its performance compared to standard schedulers.
Method. We choose a popular distributed computing framework, Ray, and write a custom external scheduler as a service for it. We modify the Ray source code to support scheduling through this external scheduler. To evaluate its performance, we conduct three different experiments using different benchmarks to compare the external and standard schedulers using a distributed computing framework, Ray.
Results. Our results show that while the external scheduler outperforms in pure computation, it struggles to keep up when the scale increases and more communication is necessary.
Conclusions. Our findings suggest that there could be meaning in developing external schedulers, however, our scheduler was not implemented well enough to overcome difficulties and thus could not outperform the standard scheduler.

1 INTRODUCTION

Distributed computing has become crucial in modern computing. By leveraging the combined computing power of multiple machines and distributing a job's workload across them, it is possible to enhance performance, scalability, and fault tolerance. However, efficient management of the resources provided by this network of machines remains a significant challenge, especially with workloads continually growing increasingly heterogeneous and dynamic.

1.1 Context

Scheduling is therefore at the core of resource management in distributed computing. It involves the allocation of tasks to available resources, such that system performance is maximized while still meeting application requirements. For a scheduler to be effective, it must address multiple, often competing, objectives: optimizing resource utilization, minimizing task latency, ensuring fairness, and accommodating application-specific constraints. The inherent unpredictability of distributed environments, such as varying network conditions, node failures and fluctuating workloads further complicates this.

In modern distributed computing, frameworks such as Spark, Dask and Ray have emerged as powerful tools for developers to build and scale distributed applications, by abstracting away a large part of the intricate complexities involved in managing distributed resources, task coordination and fault tolerance. This enables developers to seamlessly execute tasks and manage resources across clusters of machines. They feature built-in schedulers, that are designed to optimize task placement and resource utilization, accommodating a wide range of workloads, from machine learning to data processing.

1.2 Problem

Despite the versatility of the schedulers provided by distributed computing frameworks such as Spark [26], Dask [17] and Ray [13], they are designed to be as ubiquitous as possible, meaning that, while their performance is usually acceptable, it is not optimized for specific use cases. However, because it is difficult to change the scheduling approach used by these frameworks, developers generally use the framework's one-size-fits-all scheduler. This may cause them to lose out on the performance, efficiency, and sustainability benefits of some new scheduling approaches. Also, because Spark, Dask and Ray each have different scheduling approaches, it is difficult for scheduler implementers to even get started on writing custom schedulers since they would have to write specific implementations for each of these frameworks.

In this paper, we aim to design an external scheduler as a service. We focus on Ray for our implementation, but it can be used with Spark, Dask, or other frameworks as well, provided they are sufficiently modified to communicate correctly with our external scheduler.

This external scheduler as a service could provide developers who have specific scheduler requirements that Ray does not natively support with a way to implement these features themselves.

We focus on modifying the Ray source code to provide a simple and easy-to-understand interface for the external scheduler to communicate with. We also write a simple external scheduler for this modified Ray code to interface with, to test its functionality and performance.

1.3 Challenges

There are some significant challenges in implementing this external scheduler functionality for Ray.

First, since Ray is a large project, it is fairly complicated to make significant changes to the source code without breaking the functionality present in Ray.

Second, because Ray has two schedulers, one handling scheduling on a global level, dividing the work between nodes, with the

*These authors contributed equally to this report.

other handling local scheduling of tasks running on the nodes, we cannot implement external schedulers for both schedulers in Ray within the limited time we have for this project. As such, we focus on replacing the global Ray scheduler, letting the nodes themselves schedule using the existing Ray code.

Lastly, because we are focused on our scheduler's functionality, we expect overall performance to be worse than the default Ray scheduler. However, further research could be conducted into improving our external scheduler to improve this performance, possibly resulting in even better performance than Ray's scheduler offers.

1.4 Approach

We decided to look at distributed system frameworks and how they use schedulers. We found that the scheduler is a core part of distributed systems, however, because of their importance, it is difficult to add or improve features to the scheduler. This results in the schedulers not having adopted the latest advances. While most distributed systems like Ray [13], Dask [17], and Spark [26] have their own scheduler, it would be more desirable to have an external scheduler that all distributed systems can decide to use or at least have the option to use.

In this paper, the focus will be on designing and implementing an external scheduler. This paper will describe the different design decisions available and the trade-offs of each decision, and modify Ray to use an external scheduler. Finally, we will experimentally evaluate the performance of our external scheduler.

1.5 Contributions

We summarize our contributions as follows:

- ⋆ Design and implementation of an external scheduler
- ⋆ Ray modified to use this scheduler
- ⋆ Experimental evaluation of Ray using our scheduler

2 SYSTEM MODEL

In Ray's native design, Ray workers execute tasks. These tasks can contain other tasks which the worker adds to its local work queue, unless the task requirements cannot be fulfilled or if the work queue is full. In that case, the task is added to the global work queue. The global scheduler then schedules the task on a different worker. The scheduling process inside the global scheduler works like this: First, the scheduler selects a set of suitable workers that meet the task requirements. Then, the scheduler estimates the shortest completion time for each of these suitable workers. The real-time statistics required for this estimation come from the Global Control Store (GCS), the scheduler does not store any state locally. There can be multiple global schedulers to balance the scheduling load. Finally, the scheduler sends the task to the worker with the shortest estimated completion time. Only a reference to the task is transmitted, all input arguments and program data are shared through the distributed object store.

In case a worker fails, its tasks are rescheduled through the global scheduler.

In our design, the Ray global schedulers do not make the scheduling decision. Instead they make a request to our external scheduler service, which tells the global scheduler which worker should execute the task. The local schedulers remain the same. Contrary to the global schedulers, the external scheduler stores its state locally. It also does not interact with the GCS.

Communication between the global scheduler and the external scheduler happens through synchronous messages on a request-reply basis. These messages arrive reliably and in order. The global schedulers act like clients and make requests that the external scheduler replies to. There is always a single external scheduler so there is no significant scalability. Additionally, there is also no fault-tolerance mechanism to recover from a failed external scheduler.

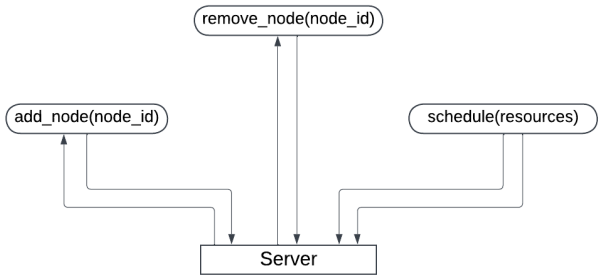

Figure 1: Overview of the client

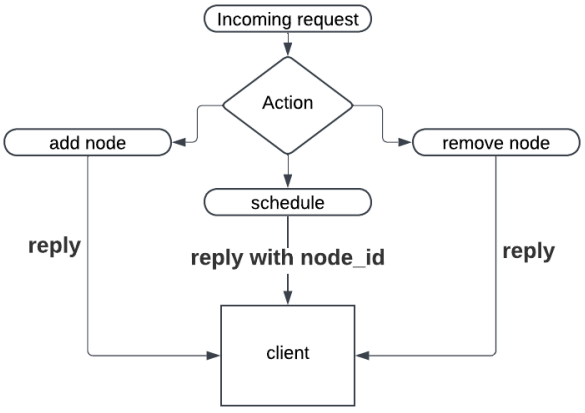

Figure 2: Overview of the server

There are three basic functions on the client side. First, add_node(node_id, available_resources) simply sends a message to the server to register a new node that can be used. The message includes the resource capabilities of that node. Second, remove_node(node_id) removes a node from the nodes that can be used. Third, the schedule(resources) function, is where the client sends the request to the server with the necessary resources of that task. The server will then respond with the node_id that the client can use to schedule the task on. These operations are shown in 1.

The external scheduler operates by continuously executing the following loop:

(1) It receives a message from a client.
(2) It checks if the message is an add/remove node or schedule request.
(3) A) If it is an add or remove node request then it adds or removes the node from a dictionary (that is stored on the server).
    B) If it is a schedule request it attempts to find a node that fulfills the resource requirements. An error is returned to the client when no such node is available.
(4) The scheduler replies to the client.
(5) The scheduler waits for the next incoming request.

A single iteration through this loop can be seen in figure 2.

3 SYSTEM DESIGN

This section gives a full overview of the system design. It also shows the system requirements and discusses the design choices and alternatives.

3.1 Users

Our external scheduler is designed for system developers who use a distributed computing framework like Ray. Whenever a new and improved scheduling mechanism is proposed, it can be implemented once in the external scheduler instead of implementing it for each framework. All systems that use the external scheduler benefit from improved scheduling mechanisms in the external scheduler, independent of which framework is used. Additionally, it is a lot easier to change the scheduling mechanism. The code in each framework can remain the same, the change is only applied to the external scheduler. This greatly reduces the effort it takes to switch between scheduling mechanisms. When these distributed computing frameworks perform better, it is beneficial to all of their users such as users of cloud services, researchers in high-performance computing and even organizations with big data workloads. The benefits are stronger with more frameworks that can use an external scheduler, however, this paper will focus on a single distributed system framework due to time constraints.

3.2 Requirements

The functional requirements for the scheduler are as follows:

(1) The scheduler must support connections from multiple nodes.
(2) The external scheduler must be able to accept tasks and schedule them for execution across available resources.
(3) The external scheduler must integrate with Ray without disrupting the system's functionality.
(4) The scheduler must dynamically assign tasks based on system resource requirements (CPU, memory, etc.)
(5) The scheduler must run on DAS-5 (Distributed ASCI Supercomputer 5) [3].

And the non-functional requirements are:

(1) The scheduler should attempt to minimize resources and maximize efficiency by making scheduling decisions based on a strategy.
(2) The scheduler must continue operation as long as Ray and the network function normally.
(3) The scheduler should be implemented in a way that allows future functionality additions.
(4) The scheduler should be easy to use on any system that it is run on.

3.3 Design process

The design process started by researching the field and reading existing papers on scheduling [12], as well as papers about architecture [4], resource allocation policies [15] and performance considerations [2].

After diving into the field, we decided on the users and use cases of our scheduler that we would focus on. Then, we set our functional and non-functional requirements. According to these requirements, we used the papers we read and listed all the options for each design option. Our main goal here was to understand the options and their alternatives.

Afterwards, we listed the options and compared them to see which solution suited our requirements best. We then started by thinking about the simplest and most necessary parts of the scheduler. This was a basic version of how the scheduler was supposed to function. After this, we used this basic version to build a high-level design. We focused on the major components and how they would interact. We also created diagrams that explain the system architecture we decided upon. This is also where we defined each component and explained what they did. Then in the low-level design, we spent a lot of time thinking about the interactions between the Ray system and the external scheduler.

When we had a clear picture of our design, we argued about its limitations. Then we implemented each component and described the policies we chose and why we chose them. This is explained in Section 3.5. Finally, after implementing, we checked our requirements and decided how well we implemented the scheduler, if we missed any requirements, or even if we could improve certain details. However, we did not have time to implement these improvements, so we will only briefly mention them in Section 6.1

3.4 High level design

First, when designing a scheduler, there are five architectural points: resource selection, complexity, load imbalance, scalability, and central points. The central points refer to the need to consider the use case when making decisions. For example, bank systems must be error-free to avoid financial losses, while social networks can handle some outdated data without much impact. In our case with Ray, we also have a multi-cluster environment, which means we had to choose an architecture for the multi-cluster environment. For simplicity, we chose a hierarchical scheduler. This is because of how Ray works internally. Each cluster has one or multiple global schedulers and each node has its own internal scheduler.

There are several scheduling architectures that we can choose from. The chosen architecture must be able to meet all the requirements listed in Section 3.2. Additionally, we carefully consider the existing scheduler mechanism in Ray because that is what our external scheduler is replacing. The requirements and analysis of the Ray scheduler resulted in the following alternatives.

3.4.1 Alternatives.

Alternative 1: Schedule all tasks centrally. The first alternative is to create a central scheduler that would schedule all tasks, bypassing all Ray schedulers. The GCS remains unchanged to exchange task input and output. The central scheduler keeps track of a run queue and, for each task and worker, their status through interactions with the GCS.

The benefit of this design is that the scheduler has a lot of control and a lot of information about each task and worker. This allows us to implement advanced scheduling algorithms and analyze the scheduling performance in great detail. The downside is that this would make it very computationally heavy on our external scheduler, because of all the tasks and workers it would have to keep track of. Additionally, the scalability is limited by the scheduling latency of the central scheduler, because all tasks are scheduled through this single scheduler. When scheduling all tasks, the external scheduler is easily overloaded. The Ray developers realized this as well, and that resulted in the hierarchical scheduling setup in Ray. Building on top of this avoids this overloading problem.

2: Make scheduling decisions for the global schedulers. Due to the Global Control Structure (GCS) in Ray, the global schedulers are stateless. The scheduler uses the GCS to make scheduling decisions by estimating the completion time and data transfer times. In this design, we do not make these estimates to reduce implementation complexity. The external scheduler makes scheduling decisions on behalf of the global schedulers, while all other functionalities such as recovering from failed worker nodes, transferring input and output data, and data dependencies are handled by Ray. All tasks are considered equal, except for their resource requirements.

The benefits are that there is no additional latency for tasks that are scheduled locally. The interactions between Ray and the external scheduler are limited, making the implementation easier. Finally, the scheduling performance is better than the single central scheduler, because the local schedulers do some of the work. The downsides are that, firstly, it is not possible to make advanced scheduling decisions, because the external scheduler has limited information about each task and worker. Secondly, although improved compared to the central scheduler, the system is limited in scalability, especially in cases where the global schedulers do a lot of the scheduling work compared to the local schedulers.

3: Do all work of the global schedulers. The third alternative is to replace the global schedulers with an external variant. To schedule the tasks, the scheduler first checks if the task has dependencies on other tasks. If so, the scheduler adds this task to a list containing all tasks waiting for dependencies and marks the task as halted. When the dependencies of a task are complete, the scheduler executes the halted task. All tasks that are dependencies of other tasks have their result values stored to avoid executing the same task twice.

If the task is ready to be scheduled, the scheduler should check the specific resource requirements which are specified in terms of the number of CPUs, the number of GPUs, and others. If these are specified, the scheduler determines which workers can perform the task, and only execute this particular task if one such worker is available. If no such worker is available, this task is moved back in the queue, making a place for tasks which could be executed on the other available workers. If the task has no resource requirements, the scheduler simply executes them on the first available worker. When the task is completed and returns a value, this value is stored in a map and the halted tasks that are waiting for this value are updated.

The benefits of this approach are that, firstly, the system is highly scalable. Similar to how the Ray scheduling works, when the global schedulers are overloaded, more global schedulers are added. Secondly, similar to the central scheduler design, there is a lot of control over how the scheduling and data transmission between workers is done. The downside is that this design has a lot of components that all need to perform well to make the system function as a whole. Replacing the entire global scheduler is a lot of work and requires large changes in the Ray system. The additional interactions between Ray and the external scheduler could cause performance issues due to the overhead of (de)serialization and synchronization between the two if this is less efficient than how Ray currently does this. Additionally, it would be challenging to generalize the interactions so that the external scheduler could be used for other systems than Ray.

3.4.2 Chosen design.

The design we decided on is to make scheduling decisions for the global schedulers (see Section 3.4.1, paragraph 2). We did not choose the central scheduler due to its performance limitations. We also did not choose the third alternative because it is too ambitious and we would not be able to complete the project in time.

3.4.3 Qualitative evaluation.

A design where the external scheduler only makes the scheduling decisions for the global schedulers allows us to leave most scheduling-related functionalities outside the external scheduler. This improves the flexibility of the external scheduler because systems other than Ray handle these functionalities differently. Additionally, the computational load on the external scheduler is low, so running it as a central scheduler has less of a performance impact than when the scheduler also does other tasks.

One of the largest downsides of this approach is that it is impossible to make use of advanced scheduling strategies because the external scheduler has very little information about the system such as real-time worker utilization and data locality. We estimate that this causes a performance bottleneck, especially when the tasks vary a lot in resource requirements. Additionally, scalability could become an issue for workloads where the global schedulers have many tasks to schedule.

3.5 Low level design

This section explains our design choices for a more detailed view of the system. We compare software systems that we use as a basis for the external scheduler. A solid basis is required to transform the high-level design into something that can actually be implemented and that actually works. After considering the alternatives and explaining the chosen design, we evaluate the properties of the design and check that it meets all specified requirements 3.2.

3.5.1 Alternatives.

Programming language. We could have used any general-purpose programming language such as C, C++, or Java.

Communication Protocol. The three most feasible alternatives are 1) an MPI-based implementation [10], 2) making use of RPC mechanisms such as Google Protocol Buffers [9] and 3) a UDP-based implementation [1].

Chosen design. The external scheduler consists of two parts: 1) The external scheduler logic in Python [23]. We chose Python because of its benefits: it is easy to program because it is dynamically typed, it supports the required data structures natively, it uses simple syntax, and many details are hidden behind simple abstractions. The downside to Python is that there is little control over what happens 'under the hood' and performance is generally lower than for alternative languages. 2) Modifications to Ray, which are in C++ [19].

The two components communicate over TCP [11] through a custom protocol. We decided to use TCP for its reliable transmission and ease of implementation for both Python and C++. Google Protocol Buffers matches closely to our application, however, we are more familiar with TCP. Furthermore, the data in each message is structured, limited in complexity and the data has a small size, so we doubt that the Google Protocol Buffers are beneficial. We could also have used UDP for communication but that would require a custom retransmission mechanism because UDP packets could be lost in the network. TCP saves us from this effort but comes with congestion control, which increases communication latency in some cases.

When the Python part starts, it shows its listening IP and TCP port number, which are used to create a very simple configuration file. The Ray C++ code uses this configuration file to set up the connection. For detailed information on how this works, see the README in this project's GitHub repository.

Ray uses C++ classes for components in the scheduler system. The most relevant classes are:

- ClusterResourceManager. This class is used to manage node resources in a cluster. It is used to add and remove nodes, and when changing the resources of an existing node at run-time.
- ClusterTaskManager. This class is used to enqueue, dispatch, and cancel tasks.
- ClusterResourceScheduler. This class manages the available node resources and makes the scheduling decisions.

The modifications in these classes ensure that the Python part has up-to-date information about the nodes. The modifications in Ray also enable the Python part to make scheduling decisions. This happens through the GetBestSchedulableNode() function in the ClusterResourceScheduler. The Python code receives the required resources for that task via the *SCHEDULE* API message. When a node is added or removed through the ClusterResourceManager, the Python code similarly receives *ADD_NODE* or *REMOVE_NODE* API messages. The communication protocol between nodes and the external scheduler is shown in Table 1. Each message is prepended with the length of the message in bytes.

The [*resources*] referenced in Table 1 can be deciphered as follows:

```
[ASCII string resource key][0x0][8 bytes double
↪  resource value]...
```

A sample communication would look like this:

Function	Code	Message	Reply
ADD_NODE	$[0x0]$	[*nodeID*] [*resources*]	$[0x0]$
REMOVE_NODE	$[0x1]$	*nodeID*	$[0x0]$
SCHEDULE	$[0x2]$	[*resources*]	$[0x0]$[*nodeID*] if successful, else: $[0x1]$

Table 1: External scheduler communication protocol

```
<request>: [8 bytes length][0x0][8 bytes node
↪  ID][resource1][0x0][resource1
↪  value][resource2][0x0][resource2 value]
<reply>:   [8 bytes length][0x0]
```

This shows the request of type: 0 (ADD_NODE) with some node ID and two resources. The reply acknowledges that the request has been processed successfully.

Qualitative evaluation. The chosen design can meet all requirements shown in Section 3.2. Firstly, the scheduler can use multiple connections at the same time by separating the states for each connection and using a different TCP connection for each (FR 1). Secondly, The scheduler accepts all scheduling requests and responds with an error if no node has all the required resources. The Ray global scheduler then retries at a later moment or throws an error (FR 2). Thirdly, since the global scheduler remains the same and only the scheduling decision-making is altered, Ray can handle all kinds of actors and tasks that it could handle normally (FR 3).

The scheduler considers the task requirements and node resource capabilities. The global scheduler shares these values via messages with the external scheduler (FR 4). Since DAS-5 can compile and run C++ programs like Ray, run Python programs, and we do not use exotic or specialized hardware or software, our design also works on DAS-5 (FR 5).

The external scheduler considers the resources in making a scheduling decision (NFR 1). TCP ensures that the messages arrive intact and in order. Additionally, when a request cannot be fulfilled by the external scheduler, an error is returned and the system can continue to operate (NFR 2). While implementing we aimed to make our code flexible such as through classes in Python and by using standard, commonly used libraries. The API messages all contain a type identifier that has many values reserved for future use. These design decisions ensure that future functionalities could easily be added (NFR 3). The only required user action is making the configuration file. When the system has started, no further user input is required (NFR 4).

4 EXPERIMENT DESIGN

In this section, we evaluate the performance of our external scheduler when compared to the performance offered by the native Ray scheduler, with the help of three experiments. These are:

- Monte Carlo Estimation of π (Section 4.1)
- Ray Torch Training (Section 4.2)
- XGBoost Training (Section 4.3)

4.1 Experiment 1: Monte Carlo Estimation of π

4.1.1 Motivation. Distributed computing systems such as Ray are designed to efficiently handle large-scale and computationally intensive tasks. To evaluate the performance, we can use benchmarks that are both computationally intensive and suitable for distributed computer systems. The Monte Carlo method for estimating the value of Pi is an ideal candidate for this because, due to its simplicity, it is easily parallelizable, computationally expensive and relevant to real-world applications [18].

The Monte Carlo estimation works with a circle inscribed inside a square. The diameter of the circle is equal to the length of the sides of the square. Then, random points are distributed over the square with a uniform distribution. We take the ratio between the points inside the circle to those outside the circle. This ratio approximates the ratio of surface area outside and inside the circle.

This ratio multiplied by 4 is an estimation for π. With more points added, the estimation becomes more accurate. Since the points are independent, we can make use of parallel computation.

4.1.2 Setup. The source code to execute this benchmark is gathered from the Ray website [20]. Due to time constraints, we are only able to experiment with a limited set of configurations. These are the standard Ray scheduler compared to the external scheduler, each with 1, 2 or 4 DAS-5 nodes. The total execution time is measured using the "time" command line utility. The "real" time is considered. The experiments are repeated 5 times and averaged.

4.1.3 Variable and constant parameters. The variable parameter in this experiment is the number of nodes, in order to learn something about the scalability of the external scheduler, compared to the built-in scheduler inside Ray. The constant parameters are the number of tasks and the number of points per task. These are 10 tasks and 10 million points, following the recommendations from the Ray website.

4.2 Experiment 2: Ray Torch Training

4.2.1 Motivation. Efficient and scalable training of machine learning models has become increasingly necessary as datasets and model complexities grow. While frameworks such as PyTorch [14] offer robust support for training on single nodes or distributed systems, the overhead present in these frameworks due to manual configuration and resource management greatly limits scalability and ease of use. Distributed computing frameworks, such as Ray, greatly simplify parallelism and resource management while offering high performance. Ray Train, in particular, provides a powerful abstraction for distributed training.

To evaluate the performance of our external scheduler, we conducted a benchmark focused on evaluating the performance of Ray Train when using this scheduler, while comparing it to the default Ray scheduler's performance. This benchmark trains a deep neural network on the FashionMNIST dataset [8, 24]. By comparing the time required to perform training with both our scheduler and the default Ray scheduler, we can evaluate our scheduler's performance.

4.2.2 Setup. The source code used to perform this benchmark is gathered from the Ray website [21]. Due to time constraints, the vanilla PyTorch benchmarking functionality of the original benchmark was removed. The execution time is measured in two places, the first being the Global execution time, meaning the time it takes to complete a full training run, as well as the Local time, which is the time the head node used to perform its part of the training.

The program is benchmarked on 1, 2 and 4 DAS-5 compute nodes. Each DAS-5 compute node has access to 32 CPU cores.

4.2.3 Variable and constant parameters. The variable parameters of this benchmark are the number of CPU nodes, the total number of workers spawned by the Ray instance, as well as how many CPU cores are allocated to each worker.

Nodes	Workers	CPUs per Worker
1	4	8
2	8	8
4	8	16

Table 2: Ray Torch Benchmark configuration

The configurations used for each benchmark are shown in Table 2. The reason the following configurations were chosen is, firstly, to populate all CPU cores available to the Ray instance. The number of workers, multiplied by the number of CPUs per worker, represents the total number of CPUs used by the benchmark. Secondly, when comparing the configuration for benchmarking 2 nodes compared to when benchmarking with 1 node, we have double the workers, but the same amount of CPUs per worker. This was done to evaluate if there is a communication bottleneck present in our external scheduler. As such, to have data to compare with, we chose not to increase the number of workers when benchmarking on 4 nodes, but instead doubled the number of CPUs per worker.

The constant parameters for this benchmark are the number of runs and the number of training epochs per run. Each benchmark was run 5 times, with 10 epochs. An epoch represents a single pass of the ML algorithm over the input dataset. On the Ray website [21], the benchmarks were run with 20 epochs, but to increase the number of runs and due to time constraints, these benchmarks are limited to 10 epochs.

4.3 Experiment 3: XGBoost Training

4.3.1 Motivation. We chose XGBoost training as a benchmark because XGBoost is a popular and widely used machine learning library [5, 7, 25]. It is a library designed for supervised learning tasks such as classification and regression [6]. The library uses decision trees as a base for gradient boosting. The iterative workload and resource sensitivity make it well-suited to be used as a benchmark. Another reason to use XGBoost is that Ray already features a ready-made benchmark on their website for evaluating Ray's performance with XGBoost [22].

Using XGBoost, we can test different training sizes and measure their performance using different amounts of parallelism. By running the XGBoost using our external scheduler and then the standard scheduler from Ray, we can evaluate our scheduler's performance. We used the dataset provided by Ray [16]. In this experiment, we compare three time metrics: the total runtime, the

runtime of the first iteration, and the average runtime of the remaining iterations. Every test was repeated 3 times and the mean and standard deviation was calculated for accuracy.

4.3.2 Setup. The benchmark gathered from the Ray website [22], was used to perform the following experiment. Due to time constraints, we decided to only compare the execution times against each other, because our external scheduler was relatively simple. The program was benchmarked on 1, 2 and 4 DAS-5 nodes. The setup only used one workflow.

4.3.3 Variable and constant parameters. The variable parameters are the number of nodes, number of workers and number of CPUs per worker. Compared with 1 node, only 1 worker with 12 CPUs is used. Then, when compared with 2 nodes, 2 workers with 12 CPUs per worker. Finally, when comparing with 4 nodes we use 4 workers with 12 CPUs per worker. These three experiments were both done for the external and the standard scheduler.

5 EVALUATION

This section shows and discusses the experiments' results. As we expected, the external scheduler's performance is lower than that of the standard scheduler. However, the results show that the external scheduler does not limit Ray's ability to handle diverse workloads and the number of nodes.

5.1 Experiment 1: Monte Carlo Estimation of π

5.1.1 Results. The results of this experiment are visualized in the figure 3. The red bars represent the external scheduler, and the blue bars are the standard scheduler. The x-axis shows the average execution time of the benchmark. The error bars show the standard deviation for each measurement. The y-axis is the various setups, described in the setup section 4.1.2.

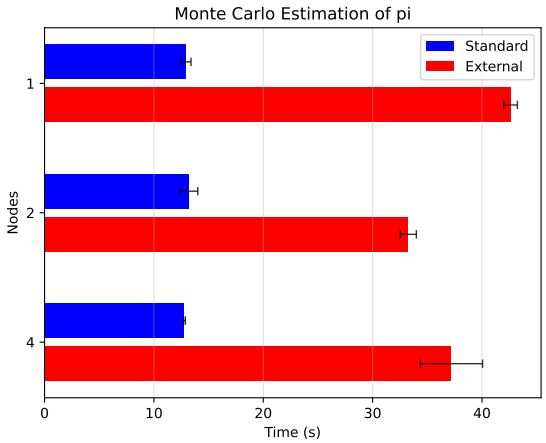

Figure 3: Standard vs External Ray scheduler mean runtime comparison

The results are also shown in the table 3.

Nodes	Mean (s)		SD (s)	
	Standard	External	Standard	External
1	12.96	42.62	0.432	0.6233
2	13.19	33.27	0.819	0.733
4	12.79	37.21	0.082	2.85

Table 3: Standard vs External Ray scheduler runtime comparison

5.1.2 Analysis. The results clearly show that the execution time is higher for the external scheduler than for the standard scheduler. Another interesting observation is that the execution time remains fairly constant when the number of nodes changes, even for the standard scheduler. It seems that Ray is unable to make use of the additional computational power. The difference between the external scheduler and standard scheduler for the same number of nodes might be explained by the additional overhead that is inherent in an external scheduler setup as we have here. Firstly, there is significant setup overhead for benchmarks like this that have a short execution time. The standard scheduler can make a simple function call inside the C++ code to make scheduling decisions. On the other hand, the external scheduler has to serialize and de-serialize the task requirements and transmit one request and one reply over the network. Additionally, the scheduling decision might not be as good as what the standard scheduler would decide because the external scheduler uses a simpler strategy. However the tasks in this workload are all very similar and there are no data dependencies between them.

5.1.3 Implications. The results demonstrate the importance of scheduling latency for workloads like the one in this experiment. The results also show that Ray does not benefit from more than one node for this particular setup. System builders should think of which scheduling approach to use for workloads like Monte Carlo simulations where many independent workloads have similar requirements.

5.2 Experiment 2: Ray Torch Training

5.2.1 Results. The results of this benchmark are shown in tables 4 and 5. All times in these tables are shown in seconds, rounded to two decimal places.

Table 4 shows the mean local runtimes measured by the benchmarking script, as well as the standard deviation (SD) of the measured runtimes. Table 5 shows the global runtimes measured by the benchmarking script, as well as their standard deviation.

In these tables, 'Standard' refers to the measurements obtained when benchmarking with the default Ray scheduler, while 'External' refers to the measurements obtained when benchmarking with our external scheduler.

The different runtimes shown in Tables 4 and 5 are visualized in Figure 4 to allow for easier interpretation using a bar chart. Here, blue bars represent measurements obtained when benchmarking the default Ray scheduler, while red bars represent the measurements obtained when benchmarking our external scheduler. Lighter-coloured bars represent local runtimes, while darker-coloured bars represent global runtimes (see legend in the Figure

Nodes	Mean (s)		SD (s)	
	Standard	External	Standard	External
1	139.67	155.25	0.31	1.81
2	484.18	489.34	20.13	18.85
4	528.94	456.67	15.82	2.64

Table 4: Local Runtime comparison. Times rounded to 2 decimal places.

Nodes	Mean (s)		SD (s)	
	Standard	External	Standard	External
1	155.35	247.73	0.76	4.58
2	500.50	652.11	20.24	18.66
4	545.61	674.40	15.82	10.80

Table 5: Global Runtime comparison. Times rounded to 2 decimal places.

as reference). The black whiskers at the end of each bar are error bars, representing the standard deviation for each benchmark.

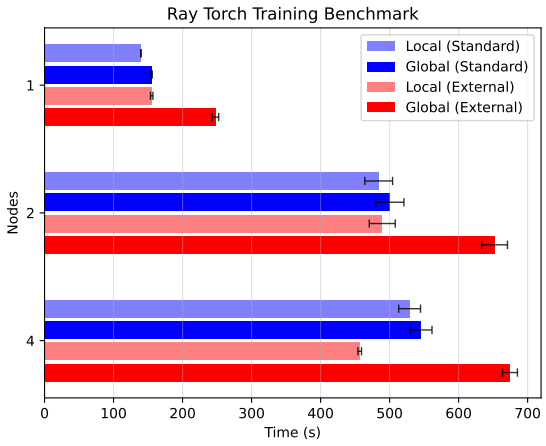

Figure 4: Standard vs External Ray scheduler mean runtimes comparison

5.2.2 Analysis. Based on the measurements shown in Tables 4 and 5, as well as Figure 4, it can be seen that our scheduler performs significantly worse in terms of global runtimes than the default Ray scheduler. On the other hand, the local measurements are sometimes better than those of the default Ray scheduler. This is likely because our scheduler only performs the scheduling functionality at the start of the benchmark, and then each node handles the local scheduling of its task using the local scheduler. As such, our scheduler adds a significant bottleneck during the initialization phase of the benchmark, as shown by the significantly greater global runtimes. The local runtimes, however, are quite close in the measurements obtained from benchmarking on 1 and 2 nodes, while when benchmarking with 4 nodes, our external scheduler even showed a lower local runtime than the default Ray scheduler.

Based on the standard deviation values shown in Tables 4 and 5, our external scheduler showed lower values than the default Ray scheduler, when executing on 2 and 4 nodes, while the standard deviation values when running on 1 node were lower for the standard scheduler than for our external scheduler. This means that the time required to complete the benchmark with our scheduler is generally more consistent across different runs than the time required to complete the same benchmark with Ray's default scheduler when running on multiple nodes, whereas it is the other way around when running on one node.

5.2.3 Implications. The results demonstrate the importance of low communication overheads when scheduling, which are what likely caused the global runtimes of our scheduler to be significantly greater than those measured with the default Ray scheduler. The results also show that, with this particular benchmark, there was no benefit to be obtained by utilizing multiple nodes. The single-node setup showed the lowest completion time when compared to the 2 and 4 node benchmarks.

There is a clear limit to how scalable our external scheduler is. When we tried to benchmark with 4 Ray nodes, in a 16-worker configuration, assigning 8 CPUs to each worker, Ray timed out and thus crashed. This was observed to be due to the long time it took for the external scheduler to process the requests at the end of a benchmarking run, which made Ray believe that a worker node had crashed, and thus made the program time out.

5.3 Experiment 3: XGBoost Training

5.3.1 Results. The results of the third experiment are shown in Table 6, Table 7 and finally Table 8. Table 6 summarizes the total average run time observed during the experiment. Table 7 summarizes the average duration of the initial iteration, and Table 8 summarizes the average duration of the remaining iterations.

Nodes	Mean (s)		SD (s)	
	Standard	External	Standard	External
1	698.02	869.99	37.38	29.60
2	542.23	922.81	77.21	19.73
4	377.28	1282.31	31.26	31.35

Table 6: Average Run Time comparison, rounded to two decimal places

Nodes	Mean (s)		SD (s)	
	Standard	External	Standard	External
1	643.96	820.83	37.55	29.62
2	516.53	895.35	76.57	17.84
4	360.61	1267.33	31.56	31.17

Table 7: Average First Iteration Time comparison, rounded to two decimal places

These results are then better visualized in bar charts. Figure 5, Figure 6 and Figure 7 respectively.

In these charts, the blue bars represent Ray with a local scheduler (Standard), while the red bars correspond to Ray with the

Nodes	Mean (s)		SD (s)	
	Standard	External	Standard	External
1	6.01	5.46	0.83	0.90
2	2.85	3.05	0.70	0.80
4	1.85	1.66	0.14	0.24

Table 8: Average Iteration Time comparison, rounded to two decimal places

proposed external scheduler (External). The x-axis indicates the time in seconds, whereas the y-axis indicates the configuration of the experimental setup, specifically the number of nodes, which reflects the number of workers utilized. The error bars denote the standard deviation, providing information on the variability of the observed measurements.

Figure 5: Standard vs External Ray scheduler average run times comparison

5.3.2 Analysis. As illustrated in Figure 5, the standard scheduler demonstrates higher performance compared to the external scheduler across all configurations in terms of total run time. A similar trend is observed in Figure 6, where the standard scheduler continues to outperform the external scheduler. However, it should be noted that the external scheduler exhibits a lower standard deviation, indicating more consistent results. In both figures, the performance of the standard scheduler improves as the number of nodes increases. In contrast, the performance of the external scheduler declines with additional nodes, suggesting a significant communication overhead in its implementation.

When analysing the remaining iterations' average time (i.e., all iterations excluding the first) with a single node, the external scheduler outperforms the standard scheduler in terms of iteration speed. The average time for two nodes for the external scheduler stands out as an exception to the overall trend. But the higher standard deviation in this case could suggest that there is an outlier in the data. The longer duration of the first iteration can be attributed to the significant amount of communication occurring within the

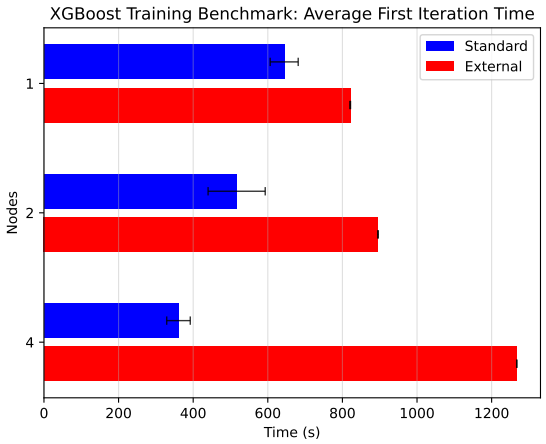

Figure 6: Standard vs External Ray scheduler average first iteration time comparison

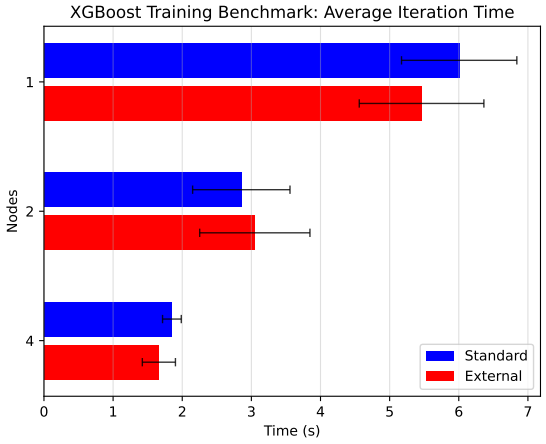

Figure 7: Standard vs External Ray scheduler average iteration time comparison

external scheduler during this phase. In contrast, Ray's standard scheduler demonstrates greater efficiency as it distributes the scheduling workload more efficiently.

5.3.3 Implications. The results indicate that while the external scheduler performs more efficiently in scenarios where no communication is required, its performance becomes bottlenecked when communication is necessary. This limitation is reflected by the data, which shows that as the number of nodes or CPUs increases, the communication overhead grows, leading to longer run times. The average run time per iteration increases for the external scheduler when more nodes/CPUs are utilized, suggesting that the communication overhead is sufficiently high to make data distribution disadvantageous. This trend is further explained in the average iteration times for both the external and standard schedulers where

in this scenario, there is minimal or no communication between nodes/CPUs. In these cases, increasing the number of nodes consistently leads to improved iteration speeds. Another thing to note is that the external scheduler shows greater consistency in its communication patterns, as indicated by the lower standard deviation. This consistency is likely a result of the simplified implementation of the external scheduler that introduces less performance variability.

6 DISCUSSION

In this project, we designed, implemented and evaluated an external scheduler that works with a modified version of Ray. We have shown our design process, along with the alternatives and design decisions for both the high-level and low-level design. We have implemented our design and evaluated its performance. Then we analyzed the results to give valuable insights into the performance. Our experiments were very time-constrained so that limits what we could learn about the performance of our implementation.

6.1 Future Work

When reflecting back on our work and also our results, we find that there are several points of note that we could improve our external scheduler on.

Our external scheduler has to process a lot of messages and do a lot of node assignments. This is a big bottleneck in our program and we could fix this by adding a new message type that changes an existing node. The performance could also increase when additional messages inform the scheduler of the current state of the system, such as which nodes are busy, where the input and output data are stored and so on. The scheduler can use this information to make better scheduling decisions.

Another great point of future work is modifying a different framework than Ray such that both frameworks can benefit from improved scheduling mechanisms.

The use of TCP and Python has shown to be successful in completing the project in time, however, there are alternatives which have the potential to decrease scheduling latency, thereby improving performance.

Currently, we are greatly limited by the scheduler's speed of processing messages through a single TCP connection. To improve this, using multithreading through, for example, a thread pool that processes all received messages, could be useful. This way, the main thread that supports the TCP connection could receive all incoming messages, but leave the processing of these messages and replying to them to other threads. This should significantly improve performance.

The performance characteristics of such an external scheduler remain unknown for large-scale configurations. For example, when performing Experiment 2, our scheduler was too slow when finishing a single benchmark iteration, causing Ray to time out and crash. Many distributed computing frameworks are often used on a larger scale than what we used in the evaluation, so it is important to conduct further research on methods to improve this. However, if the speed of message handling of our external scheduler was improved, it should also help with improving our external scheduler's support for large distributed systems.

A TIME SHEETS

Table 9 shows how much time each of us spent on the major parts of this assignment. The different time types are defined as follows:

- the `total-time` = total amount of time spent in completing the assignment.
- the `think-time` = total amount of time spent in thinking about how to solve the assignment.
- the `dev-time` = total amount of time spent in developing the code needed to solve the assignment.
- the `xp-time` = total amount of time spent in experiments for the assignment.
- the `analysis-time` = total amount of time spent in analyzing the results of the experiments for the assignment.
- the `write-time` = total amount of time spent in writing this report
- the `wasted-time` = total amount of time spent in activities related to the assignment, but which cannot be charged as `think-time`, `dev-time`, `xp-time`, `analysis-time`, or `write-time`.

Time (hours)	Ingmar	Simon	Maurits
`total-time`	85	95	80
`think-time`	5	5	5
`dev-time`	25	35	10
`xp-time`	15	10	15
`analysis-time`	5	5	5
`write-time`	15	15	35
`wasted-time`	20	25	10

Table 9: Time sheet of approximate time in hours spent by project contributors on different parts of the assignment.

Ingmar. I mostly spent time on developing the external scheduler code in Python. However, I have a large amount of wasted time due to the Ray source code being unable to compile on my local machine. I spent multiple days trying to get it to work, still failing in the end. Eventually, I used a working compiled version on DAS-5 for the experiments, so it was unnecessary to continue trying to compile on my local machine. I also spent a relatively long time on performing the benchmarking experiments. Afterwards, I also spent time on writing and polishing the report.

Simon. I spent a large portion of my time on implementing and debugging the modifications of Ray. The ray source code is very sensitive to the configuration, virtual environment, versions of software, operating system, path variables etc. This caused many issues that took a lot of time to solve. I also spent a lot of time designing the system, for instance the interactions between the Python and C++ code and which software and protocols to use. I also did one of the experiments and wrote some parts in the report.

Maurits. I did not much on the developing of the code and thinking how to solve the assignment, however I spend most my time to write the report. I also took quite a long time for my experiment time. This was because I had a lot of issues trying to get the benchmark to work, and since my benchmark took 15 minutes to run on average it took a lot of time. It was my first time using DAS5 so it was a lot of trial and error.

B PROJECT SOURCE

The project's source is publicly accessible as a fork of the original Ray project on GitHub at: https://github.com/generalnobody/ray.

For instructions on how to build the project and use it with our external scheduler, see the README.

This report is also available within the same GitHub repository, at: https://github.com/generalnobody/ray/blob/ray-2.39.0-dev/report.pdf.

REFERENCES

[1] 1980. User Datagram Protocol. RFC 768. https://doi.org/10.17487/RFC0768

[2] Njoud AlMansour and Nasro Min Allah. 2019. A Survey of Scheduling Algorithms in Cloud Computing. In *2019 International Conference on Computer and Information Sciences (ICCIS)*. 1–6. https://doi.org/10.1109/ICCISci.2019.8716448

[3] Vrije Universiteit Amsterdam. 2012. DAS-5: Distributed ASCI Supercomputer 5. https://www.cs.vu.nl/das5/home.shtml

[4] Georgios Andreadis, Laurens Versluis, Fabian Mastenbroek, and Alexandru Iosup. 2018. A reference architecture for datacenter scheduling: design, validation, and experiments. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 478–492.

[5] S. Bhattacharya, S. R. K. S, P. K. R. Maddikunta, R. Kaluri, S. Singh, T. R. Gadekallu, M. Alazab, and U. Tariq. 2020. A novel pca-firefly based xgboost classification model for intrusion detection in networks using gpu. *Electronics* 9 (2020), 219. Issue 2. https://doi.org/10.3390/electronics9020219

[6] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. Association for Computing Machinery, New York, NY, USA, 785–794. https://doi.org/10.1145/2939672.2939785

[7] Y. Chen, C. Lin, H. Yen, P. Su, Y. Zeng, S. Huang, and I. Liu. 2022. Machine-learning algorithm for predicting fatty liver disease in a taiwanese population. *Journal of Personalized Medicine* 12 (2022), 1026. Issue 7. https://doi.org/10.3390/jpm12071026

[8] Torch Contributors. 2017. FashionMNIST - Torchvision 0.20 documentation. https://pytorch.org/vision/stable/generated/torchvision.datasets.FashionMNIST.html

[9] Chris Currier. 2022. *Protocol Buffers*. Springer International Publishing, Cham, 223–260. https://doi.org/10.1007/978-3-030-98467-0_9

[10] Jack J Dongarra, Steve W Otto, Marc Snir, David Walker, et al. 1995. An introduction to the MPI standard. *Commun. ACM* 18 (1995), 11.

[11] Wesley Eddy. 2022. Transmission Control Protocol (TCP). RFC 9293. https://doi.org/10.17487/RFC9293

[12] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. 2016. Altruistic scheduling in {Multi-Resource} clusters. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 65–80.

[13] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. arXiv:1712.05889 [cs.DC] https://arxiv.org/abs/1712.05889

[14] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

[15] Jeff Rasley, Konstantinos Karanasos, Srikanth Kandula, Rodrigo Fonseca, Milan Vojnovic, and Sriram Rao. 2016. Efficient queue management for cluster scheduling. In *Proceedings of the Eleventh European Conference on Computer Systems* (London, United Kingdom) *(EuroSys '16)*. Association for Computing Machinery, New York, NY, USA, Article 36, 15 pages. https://doi.org/10.1145/2901318.2901354

[16] Ray. [n.d.]. 10G XGBoost Data (Parquet Format). s3://air-example-data-2/10G-xgboost-data.parquet. First accessed: January 6th, 2025.

[17] Matthew Rocklin. 2015. Dask: Parallel Computation with Blocked algorithms and Task Scheduling. In *SciPy*. https://api.semanticscholar.org/CorpusID:63554230

[18] Rajkumar Sharma, Piyush Singhal, and Manoj Kumar Agrawal. 2021. Application of Monte-Carlo Simulations in Estimation of Pi. *IOP Conference Series: Materials Science and Engineering* 1116, 1 (apr 2021), 012130. https://doi.org/10.1088/1757-899X/1116/1/012130

[19] Bjarne Stroustrup. 2013. The C++ Programming Language.

[20] The Ray Team. 2024. Monte Carlo Estimation of pi. https://docs.ray.io/en/latest/ray-core/examples/monte_carlo_pi.html

[21] The Ray Team. 2024. Pytorch Training Parity. https://docs.ray.io/en/latest/train/benchmarks.html#pytorch-training-parity

[22] The Ray Team. 2024. XGBoost training. https://docs.ray.io/en/latest/train/benchmarks.html#xgboost-training

[23] Guido Van Rossum et al. 2007. Python programming language.. In *USENIX annual technical conference*, Vol. 41. Santa Clara, CA, 1–36.

[24] Han Xiao, Kashif Rasul, and Roland Vollgraf. 2017. Fashion-MNIST: A Novel Image Dataset for Benchmarking Machine Learning Algorithms. *arXiv preprint arXiv:1708.07747* (2017). https://arxiv.org/abs/1708.07747

[25] S. Yang, L. Cao, Y. Zhou, and C. Hu. 2023. A retrospective cohort study: predicting 90-day mortality for icu trauma patients with a machine learning algorithm using xgboost using mimic-iii database. *Journal of Multidisciplinary Healthcare* Volume 16 (2023), 2625–2640. https://doi.org/10.2147/jmdh.s416943

[26] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (Oct. 2016), 56–65. https://doi.org/10.1145/2934664