UNIVERSITY OF AMSTERDAM

Ibis Data Serialization in Apache Spark

February 9, 2020

Students:
Dadepo Aderemi
12160555

Mathijs Visser
12815551

Supervisor (eScience Center):
dhr. dr. Jason Maassen

Supervisor (UvA):
dhr. dr. Adam Belloum

Abstract

With the demand for real-time big data analytics, the efficiency and performance of big data tools have become increasingly more important. One of these tools is Apache Spark, and like most other distributed applications, serialization plays an important role in its performance. Ibis is an alternative serialization algorithm that has been developed with performance and efficiency in mind. This research shows the implementation and performance difference of Ibis serialization in Apache Spark. In our research 15 out of 17 Spark classes, where direct serialization calls are made, were replaced with Ibis serialization. This research measured the performance impact of using Ibis serialization on the RDD level APIs. The performance difference was measured using three different benchmarks, TeraSort, SparkPi, and memory persistence. TeraSort and memory persistence both heavily utilize serialization and SparkPi focuses on computational performance.

We concluded the performance difference between Ibis and the default Java and Kryo serialization differed per benchmark. The benchmarks that heavily utilize serialization have shown that Ibis was 15% to 20% faster than native Java serialization, and 5% to 10% faster when compared to Kryo serialization. We also measured slight differences in memory utilization, however, we were not able to conclusively say which serialization type has a clear advantage in terms of memory utilization. Ibis serialization did not lead to a notable performance difference in computationally-oriented benchmarks.

Contents

1 Introduction

Apache Spark[1, 2], is widely used in large-scale data processing with features for batch processing, as well as, real-time streaming. With the ever-increasing demand for real-time big data analytics, the performance of these applications has become ever important. Serialization plays an important role in the performance of distributed Spark applications[3]. In this paper, we show the design, implementation and performance comparison of Ibis serialization in Apache Spark. Ibis serialization takes advantage of compile-time code generation, zero-copy buffer management, and optimized object creation to improve performance.

1.1 Spark serialization

By default, Spark comes with two serialization implementations. Java object serialization[4] and Kryo serialization[5]. There are many places where serialization takes place within Spark. By default most serialization is done using Java object serialization. The reason for using Java object serialization is that Java serialization is more flexible and does not have any prerequisites. Kryo, on the other hand, requires the application programmer to register custom classes ahead of time to achieve the best performance.

1.2 Ibis serialization

Ibis[6] is an open-source, distributed computing software project developed at the Vrije Universiteit Amsterdam. The goal of the Ibis project is to create efficient Java-based grid computing software. One of the Ibis projects is an efficient serialization implementation that utilizes compile-time code generation, zero-copy buffer management and optimizes object creation to improve performance. Ibis serialization is described in further detail in section 3.2.

1.3 Research Questions

We are interested in the performance impact of using Ibis' serialization techniques in Apache Spark. Our main research question is defined as:

Can Apache Spark's performance be improved by taking advantage of Ibis' serialization techniques?

To answer this question we answer the following sub-questions:

1. What components of Apache Spark can benefit from Ibis' fast serialization?
2. How can Ibis' serialization techniques be integrated into Apache Spark?
3. How does the performance of Apache Spark differ when using Java, Kryo and Ibis serialization?

1.4 Structure

The remainder of this paper is structured as follows. In section 2 we look at related work. In section 3, we give the necessary background information underpinning the work done in this research. In section 4 we describe our approach to modifying Spark and measuring the performance difference between the different versions. In section 5 we show the findings of our measurements. In section 6 we discuss our findings. The conclusion that can be drawn from our findings is given in section 7. Finally, in section 8 we give suggestions for future research that we were not able to perform due to the limited amount of time set for this research.

2 Related work

In recent years, there have been improvements to Spark using various methods such as Remote Direct Memory Access (RDMA)[7], and zero-copy buffer management in the network stack as shown by Li et al.[8]. Apache Spark has also shown serialization performance can be improved by using Kryo [3, 5], a third-party serialization library. Apache Spark originally only supported Java serialization[2]. In version 2.0.0, Spark adopted Kryo serialization which has shown to improve serialization performance[3].

Previous work has also been done in improving the performance of other systems by introducing Ibis. As an example, in [9], Maassen et al. demonstrated that an Ibis based implementation of Java's RMI can lead to improved throughput of up to a factor of 9. The research leads to the creation of Ibis RMI which is a test case to validate the efficiency of the Ibis stack. The Ibis RMI implementation has identical API to the official Java RMI and is written in pure Java, removing the need for any custom Java compiler. It should be noted that the Ibis RMI implementation makes use of other parts of the Ibis stack as well and not just the serialization layer.

In [10], it has been shown that serialization performance in resource-constrained platforms, like the Java Micro Edition [11] can be improved by applying techniques used in high-performance computing environments. The research built upon the serialization module in Ibis. It was able to successfully demonstrate that object marshaling is possible, with decent performance characteristics, in a resource-constrained environments. This is another example of applying Ibis to achieve improved serialization in other systems.

To the best of our knowledge, our work in this research would be the first attempt at integrating portions of the Ibis project into Spark.

3 Background

In this section, we give the background information necessary to understand the rest of the paper. This section is further divided into two subsections, namely, Spark's inner workings and the inner workings of Ibis' serialization.

3.1 Apache Spark

Spark has three main data structure APIs, that abstract away the underlying data structures. The oldest and most important API is the Resilient Distributed Dataset (RDD) API. An RDD is an immutable and distributed collection of objects. The objects stored in RDDs can, for example, be Java Virtual Machine (JVM) objects. More recent versions of Spark contain the DataSet and DataFrame abstraction, which is an additional layer on top of the RDD API. In earlier versions of Spark, the DataFrame was a distinct API, however, in more recent versions of Spark, this is no longer the case, as the DataFrame API is an alias for a specialized type of DataSet [12]. DataSets and DataFrames provide a richer API and benefit from the Catalyst optimizer[13], which is a Just-in-time (JIT) compiler for Spark queries.

Spark utilizes two serialization implementations, by default Spark will use Java's object serialization[3, 4] when storing data to disk. The other serialization implementation is Kryo [5]. Kryo is only used by default when sending RDDs with primitive types, or arrays thereof, over the network[3].

The importance of serialization in Spark depends on the application that is being run by Spark. When an application reads or writes to disk, all objects have to be (de)serialized. Another use-case where serialization plays a big role is in network communication. Due to the distributed nature of Spark applications, network communication has to occur every time a block of data is not directly accessible by a worker node.

3.1.1 Application execution in Apache Spark

Spark provides a fast and general-purpose cluster computing environment. Applications that run in the Spark environment are a combination of jobs, where a job is the high-level representation of the work items to be performed.[14] A Spark job is further decomposed into one or more stages. [15] A stage is a collection of tasks that are separated at shuffling boundaries. A shuffle boundary implies computation that does not depend on data to be copied from other nodes. The Spark task is the final atomic unit of computation that is sent to each node where it operates on the RDDs.[15] This relationship between the different components of a Spark application is depicted in figure 2.

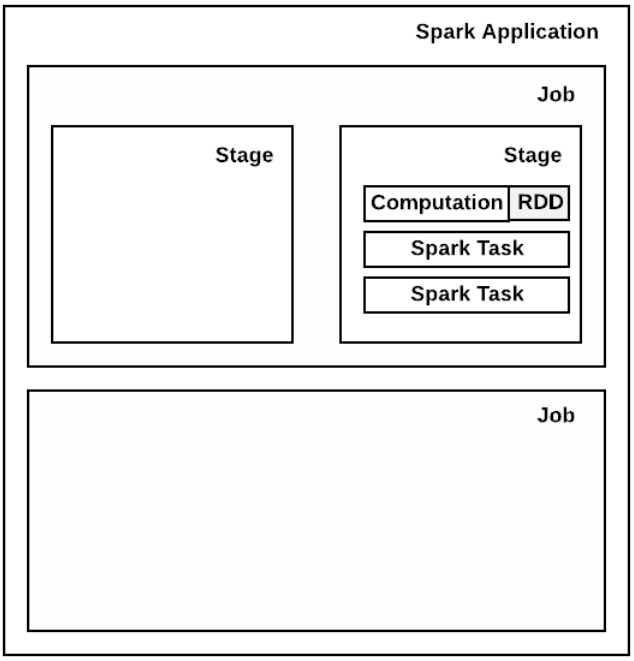


Figure 1: Composition of a Spark application.

3.1.2 Serialization in Apache Spark

To be able to turn Spark applications into a series of tasks that operate on the RDDs, Spark provides a general engine based on DAGs (Directed acyclic graph) and data sharing [16]. This involves turning the application logic into a DAG, made up of stages by the DAG Scheduler. After this, the DAG scheduler determines the optimal location on the cluster to run the stages and then passes them to the low-level Task Scheduler[14]. The Task Scheduler is the component that is finally responsible for sending the tasks to the cluster to be run.[17].

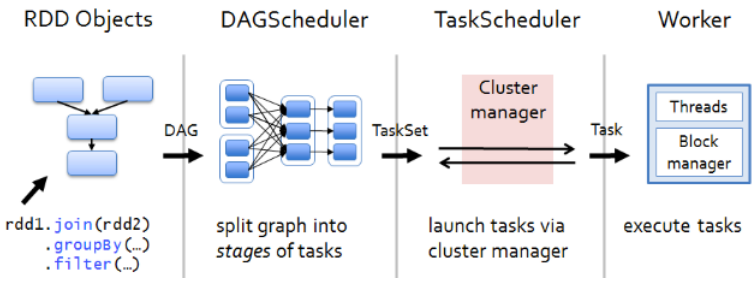


Figure 2: Executing a Spark application. [18]

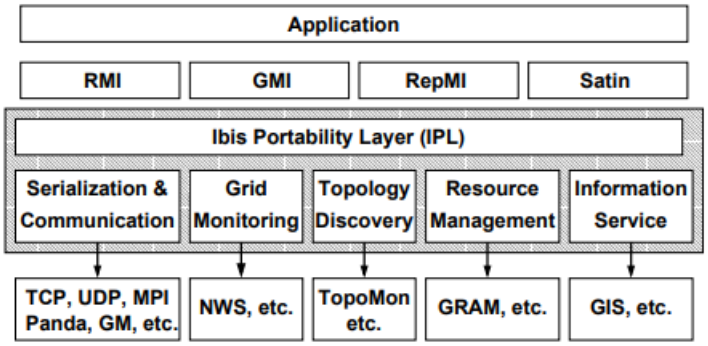
This process of analyzing and executing Spark applications leads to communication between various components within the Spark framework. These communications use both data structures internal to Spark and those defined by the Spark application being executed. We identified the various places where serialization occurs within the Spark code base and we were able to replace the serialization mechanism with the one provided by Ibis in all, except

two instances. See Appendix A for the list of locations where we found direct serialization calls within Spark. The incompatibilities that prevented replacement with Ibis serialization is discussed in section 4.2

3.2 Ibis

As explained in the introduction, Ibis is an open-source distributed computing software project. Figure 3 gives an overview of the different modules that are part of the Ibis project. This research however only focuses on the serialization module of Ibis.

Figure 3: TeraSort memory usage during execution. Adapted from [9]



The Ibis serialization module differs from normal Java serialization in several ways. With native Java serialization, the application programmer implements the empty special *java.io.Serializable* interface. When an object that implements this interface is being serialized, Java will not only serialize this object, but also the type, version and any references it uses. Java serialization will do this at run-time using run-time type inspection, also called *reflection* in Java. Ibis serialization optionally moves some of this work to compile-time by generating methods for each class that implements *ibis.io.Serializable*. This compilation step is performed by the ibis compiler *Ibisc*. Ibisc works on jar archives and will rewrite the necessary bytes in an already compiled archive. This compilation step is optional, when the generated methods are not present, Ibis will fall back to type inspection.

Additionally, Ibis optimizes object creation. When using native Java deserialization, objects are created using the *private native Class.newInstance* method inside the standard class library. Van Nieuwpoort et al.[9] have identified that this way of creating objects is relatively expensive. Ibis serialization optimizes this by creating a *generator* for each class. This generator uses the expensive *newInstance* method only once when a class has first been encountered. For every subsequent class creation, Ibis will use a lookup table to get the type information and create the object using the class constructor.

4 Approach

To achieve to goal of integrating Ibis serialization in Spark, and to measure the performance differences on the RDD APIs, we first started with researching the Spark codebase. The goal of this research is to identify all Spark components that use serialization. We have used this information to identify the changes that are necessary to integrate Ibis serialization. The necessary changes and our approach is explained in section 4.1. After the implementation, we will use standardized benchmarks to measure the performance difference. The benchmarks we used, the test environment and used configurations are explained in detail in section 4.3.

4.1 Modifying Apache Spark to use Ibis serialization

The serialization layer in Spark follows the well known, object-oriented heuristic of "program to an interface and not an implementation"[19]. This recommendation stipulates that a software component should not be used directly but via an abstraction. The abstraction can be provided via an interface or abstract class. This makes it easy to swap out one component for another. This guideline is followed in the Spark code base and it ensures that the serialization components are not used directly but via an abstraction layer specified by abstract classes. The serialization interfaces specified via abstract classes by Spark are *Serializer*, *SerializerInstance*, *SerializationStream*, and *DeserializationStream*. See section 9.0.2 in the Appendix, for the listing of these abstract classes.

We created a version of Spark that uses Ibis for serialization [20] by providing an Ibis backed implementation of these specified abstract classes. We used the *io* and *util* modules from the Ibis code base for this, as these modules contain the serialization logic needed. To provide a working implementation, a modified version [21] of these modules had to be created. These modifications and unresolved incompatibilities are further discussed in section 4.2. We then made the modified ibis modules a dependency of Spark and used them to implement concrete serialization classes for the abstract ones provided by Spark.

4.1.1 Implementing Serializer abstract class

This abstract class provides the mechanism that ensures that the underlying serialization happens in a thread-safe manner. It defines the method that is used to create the serialization object, represented by the *SerializerInstance*, that performs the actual serialization. The abstract class and the concrete Ibis implementation can be seen in listing 4 of the Appendix.

4.1.2 Implementing SerializerInstance abstract class

This abstract class represents an instance of a serializer. It defines methods that allow serialization and deserialization to be performed on specific classes from Java's standard library: namely *ByteBuffer*, *OutputStream*, and *InputStream*. The abstract class, together with the concrete Ibis backed implementation can be seen in listing 5 of the Appendix.

The methods defined in the abstract class are designed work on *ByteBuffer* and they transform the *ByteBuffer* into either *InputStream* or *OutputStream*. Since Ibis serialization expects the type *byte[]*, a modification was made to translate *ByteBuffer* to *byte[]*. This modification is further discussed in section 4.2.

4.1.3 Implementing SerializationStream abstract class

This abstract class represents a stream for writing serialized objects. It is used by *SerializerInstance* when performing the actual serialization of data to the output stream. Listing 6 in the Appendix shows the definition of the abstract class and the Ibis implementation.

4.1.4 Implementing DeserializationStream abstract class

This abstract class is the dual of *SerializationStream*. It represents the stream for reading serialized objects. It is used by *SerializerInstance* when reading data from the input stream. Listing 7 in the Appendix shows the definition of the abstract class and the Ibis implementation.

4.2 Modifications and incompatibilities

Replacing the serialization in Spark with Ibis resulted in incompatibilities that had to be resolved to have a working Spark setup. Not all of these incompatibilities could be resolved as part of this research. This lead to the inability to use Ibis serialization in certain parts of Spark.

This section goes over the modifications that were made in order to have Spark select Ibis as its sterilizer and also to resolve some of the incompatibilities encountered. We also highlight those incompatibilities that could not be resolved.

4.2.1 Modification to Spark to select Ibis for serialization

The main changes needed to make Spark pick Ibis for serialization was made in *SerializerManager.scala*: the component which configures serialization for various Spark components and in *SparkEnv.scala*: which is the configuration object that holds all the runtime environment objects for a running Spark instance. The other changes made can be seen in our Github repository [22], which contains the source code of the modified Spark.

4.2.2 Modification to Ibis to support Scala's Option type

One of the issues that arose when Ibis was introduced as the serialization component in Spark, was the occurrence of *scala.MatchError*. This is an error that is thrown when an object does not match any pattern of a pattern-matching expression [23]. Introducing Ibis lead to the *scala.MatchError* when pattern matching within the *BlockStoreShuffleReader.scala* class.

The reason Ibis serialization leads to this exception is due to how new instances are created in the processes of deserialization. Listing 1 shows the relevant section where the *newInstance* method is called when a new object is created during deserialization.

Listing 1: New instance creation during deserialization in Ibis

```
Object newInstance() {
    try {
        return newInstance.invoke(objectStreamClass, java.lang.Object[]) null);
    } catch (Throwable e) {
        return null;
    }
}
```

This creation process would always create a new instance of the object. This does not align with the requirement that the *None* value of Scala's Option type should be a singleton. The modification that resolved these incompatibilities is reproduced in listing 2.

Listing 2: New instance creation in Ibis

```
Object newInstance() {
    try {
        if (objectStreamClass.getName().equalsIgnoreCase("scala.None")) {
          return scala.Option.apply(null);
        } else {
          return newInstance.invoke(objectStreamClass, (java.lang.Object[]) null);
        }
    } catch (Throwable e) {
        return null;
    }
}
```

This ensures that the *None* value of the *Option* type is created using *Option*'s constructor instead of the *newInstance* method. This guarantees the required invariant are kept.

4.2.3 Modification to make Ibis work with ByteBuffer

As mentioned in 4.1.2, one of the abstract classes defined in Spark for serializing expects the *ByteBuffer* type, while component from Ibis serialization works with *byte arrays*. This resulted in the need to have code that converts *byte array* to *ByteBuffer* during serialization, and *ByteBuffer* to *byte array* during deserialization. The code for this conversion had to be introduced as part of being able to use Ibis serialization in Spark and can be seen in listing 3.

Listing 3: Converting ByteBuffer to byte array

```
private[this] def byteBufferToByteArray(bytes: ByteBuffer):Array[Byte] = {
  if (bytes.hasArray) {
    bytes.array()
  } else {
    val bytesArray= Array.fill[Byte](bytes.remaining())(0)
    bytes.get(bytesArray, 0, bytesArray.length)
    bytesArray
  }
}
```

Care was taken to ensure that a *ByteBuffer* that is not backed by an accessible *byte array* is also handled properly. The code for converting byte array to *ByteBuffer* was a simple call to the *ByteBuffer.wrap(...)* method, which wraps given byte array as *ByteBuffer*.

4.2.4 Unresolved Incompatibilities with Netty backed Block Transfer Service

Netty is an asynchronous event-driven network application framework for the development of high-performance protocol servers and clients [24]. Spark's network stack is implemented using Netty. After modifying Spark to use Ibis serialization, we observed that the Netty backed block transfer service throws an exception when a benchmark is run. A snippet of the stack trace can be seen in the Appendix, in listing 9. Due to time restriction, we were not able to further investigate this incompatibility and attempt a solution. We include a possible reason for the incompatibility in section 6.

4.2.5 Unresolved Incompatibilities with persisting to Hadoop filesystem

We observed that retrieving a saved file fails after modifying Spark to use Ibis for serialization. The stack trace of the exception can be seen in the Appendix in listing 8. Due to time restriction, we were unable to further investigate this incompatibility and attempt a

solution, although we mention possible explanations for this incompatibility in section 6.

4.3 Performance evaluation

In this research, we distinguish between two versions of Ibis serialization, the compiled and the uncompiled version. As mentioned in the background, Ibis serialization allows to perform the type inspection at compile-time, instead of run-time. This does, however, require the user to perform an additional step in the building process of creating Spark applications. In this research, we measure the performance of both the compiled version, named Ibisc and the version without the additional build step, called Ibis.

When measuring the performance difference between the Spark versions it is important to choose relevant benchmarks. Our benchmarks aim to show the performance of different Spark components and features. To achieve this we have chosen the following three benchmarks:

- TeraSort
- SparkPi
- Persistence

These benchmarks and why we chose them are discussed in greater detail in sections 4.3.1 to 4.3.3.

During the benchmarks, we are interested in multiple metrics. We do not only measure the time it takes to complete the job, but also the amount of memory used in the process. We measured the memory utilization in 50 benchmark runs using the monitoring scripts as shown in our Github repository [22]. The memory utilization results show the mean of all 50 benchmark runs.

In order to gain insight into the amount of serialization calls, as well as the calling components, we implemented a tracing option in the modified version of Spark. The modified Spark version checks whether the *traceOn* environment variable is set, if it is, the application will provide traces for serialization calls. This information can be very valuable because we did not manage to replace all components where serialization takes place. This way we provide insight into how many times each serialization type was used per test using the *traceOn* environment variable. The *traceOn* environment variable was only set once and the result of this test is not included in the results because enabling tracing impacts performance. That is why we performed separate benchmarks to measure the number of calls per serialization type.

When running the benchmarks, there are a lot of different processes running. We aim to minimize this amount as much as possible, however, some processes, such as those of Hadoop and Yarn, are required to run Spark jobs. This causes the standard deviation of benchmarks to be relatively high when measured as a single entity. To reduce the standard deviation and to make the results more reliable we performed each benchmark 50 times. The presented results show the mean of those 50 benchmark runs.

4.3.1 TeraSort

TeraSort[25] is a standardized benchmark consisting of three parts, TeraGen, TeraSort, and TeraValidate. TeraGen is used to generate unsorted data, which can be sorted using the TeraSort script. TeraValidate can be used afterward, to verify the fully sorted data. In our

experiment, we measured the time it takes for the TeraSort task to run when sorting 1GB of data. The data was generated once, to ensure the input data for all benchmarks is the same.

In a TeraSort performance evaluation by Li. et al. [26] it has been shown that the shuffling phase of TeraSort takes 98% of the total execution time when using 4 datanodes interconnected with a 100MB/s link. Since data is serialized before being transferred over the network, this benchmark gives insight into the performance when serializing primitive types.

4.3.2 Persistence

Spark allows data to be persisted to memory or disk across operations. The Spark documentation calls this one of the most important capabilities of Spark[27]. There are four options when persisting data, namely, *useDisk*, *useMemory*, *serialized*, and *replication*. In our experiment, we are mostly interested in the third option. This option specifies whether data should be stored as serialized objects instead of JVM objects. Storing data as a serialized object is generally more space-efficient.
The benchmark we chose to run creates an array of custom objects which are serialized and persisted to memory using the *MEMORY_ONLY_SER* flag. We chose for memory only because we want to measure the serialization speed and not be blocked by disk write speeds. The code for this benchmark can be found in *benchmarks/persistence.java* in our Github repository[22].

4.3.3 SparkPi

SparkPi[28] is an example job that is included with the Spark source code. SparkPi estimates Pi up to 16 decimal places by applying a Monte Carlo method[29]. Monte Carlo methods are methods that rely on repeated random sampling to determine a numeric value. Although this benchmark does not use serialization extensively like the other benchmarks, we have chosen this test to give us insight into the performance difference of compute-intensive Spark jobs.

A single SparkPi run on our test setup takes around 15 seconds. Due to the short job duration, we have increased the number of benchmarks we run to 200 for SparkPi. We did this to make the results more reliable.

4.4 Test environment

Our test environment is managed using Apache Yarn[30]. Yarn is a resource manager and job scheduler for Spark. Our test setup consists of two servers acting as worker nodes in the Yarn cluster. One of the servers is designated as Spark application driver and acts as a Yarn client. The servers are directly connected with a gigabit link to prevent external variables from influencing the test results.

The file system of the servers is shared using the Hadoop File System (HDFS)[31]. Both servers are datanodes containing all data used in the tests. All tests were performed using the software versions as shown in table 1. All installations, except Hadoop and Spark, used the default settings. The changes made to the configuration were the minimal changes necessary to run Spark applications on two servers. Both servers used the same configuration, as can be found in the configuration repository on our Github page [22].

Software	**Version**
Linux Kernel	4.15.0-72-generic
Ubuntu	18.04.3 LTS
Apache Hadoop	3.2.1
Apache Yarn	3.2.1
Apache Spark	2.4.4
OpenJDK Java Runtime Environment	1.8.0_232

Table 1: Software versions

Java version 1.8.0 was used even though it is a version that has passed through the *End of Public Updates process* [32]. This is because Spark version 2.4.4 does not work with Java versions higher than 1.8.0. Support for versions of Java higher than 1.8.0 is only planned for version 3 of Spark [33].

To prevent previous runs from influencing the test results, our test environment is cleared using the cleanup scripts shown in our Github repository[22]. The cleanup scripts remove all Spark and Hadoop staging files and restarts the Hadoop services to prevent the reuse of objects from influencing the test results.

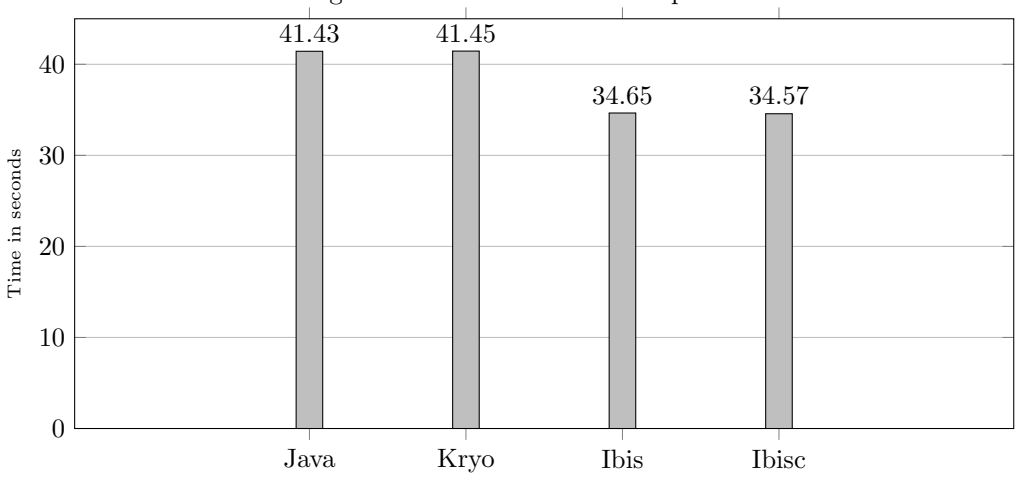
5 Results

This section describes the results of the experiments, as described in the approach section. For each test case, we give a performance comparison, as well as a comparison in memory utilization. We will also provide the number of calls that are being made to different serialization types during a single run of a benchmark. The values shown are the mean of all 50 benchmark runs. More detailed tables will also show the standard deviation (SD) and the relative standard deviation (RSD). The SD and RSD show the variance and reliability of the benchmarks.

5.1 TeraSort

Figure 4 shows the time it takes to sort 1GB of data using the TeraSort algorithm with different serialization techniques. We can see that the time to completion of Java and Kryo is nearly the same. This is explained by the fact that TeraSort does not use any custom classes. As explained in the background, Spark will use Kryo serialization during shuffling whenever there are no custom classes used in the Spark application. We can also see that the additional compilation step performed in Ibisc, only has a minimal impact on the application performance.

Figure 4: TeraSort time to completion



In table 2, we can see the mean number of milliseconds it takes to complete one test. We can also see that the results did not vary a lot between individual tests, with an RSD between 2.00% and 2.41%.

Table 2: TeraSort time to completion in milliseconds.

	Java	**Kryo**	**Ibis**	**Ibisc**
Mean milliseconds	41436.64 ms	41459.20 ms	34650.12 ms	34574.12 ms
SD	971.35 ms	830.19 ms	800.62 ms	831.93 ms
RSD	2.34%	2.00%	2.31%	2.41%

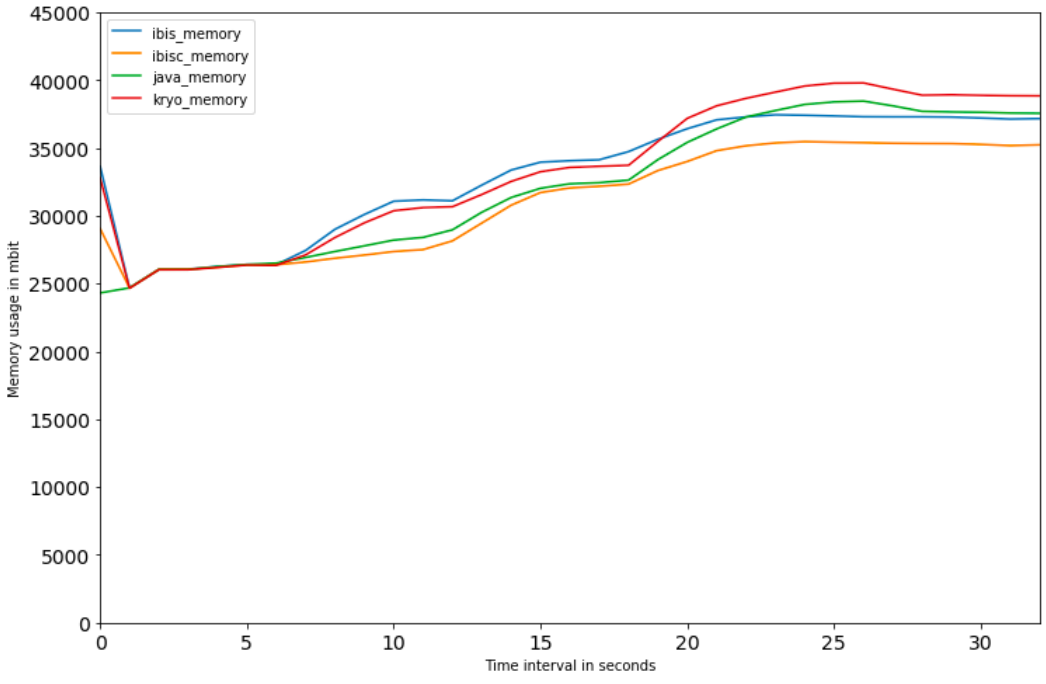
In table 3 the amount of calls per Spark component are shown. We can see that even though most components exclusively use Ibis serialization, there is still a significant amount of calls to native Java serialization. The calls to Java serialization all come from the NettyRpc component which was not modified. This benchmark did not make use of Kryo serialization in any component.

Table 3: Amount of serialization calls per component in one TeraSort benchmark run

Component	**Ibis serialization calls**	**Java serialization calls**
ClosureCleaner.scala	4	0
DAGScheduler.scala	6	0
TorrentBroadcast.scala	3	0
TaskSetManager.scala	6	0
TaskResultGetter.scala	6	0
TaskResult.scala	23	0
NettyRpcEnv.scala	0	104
Total	**48**	**104**

The memory usage during the execution of the job can be seen in figure 5. We can see that the memory usage of Java and Kryo is nearly identical, with the peak memory usage slightly above 40.000 Mbit. Both Ibis and Ibisc peak at 37.500 and 36.000 respectively.

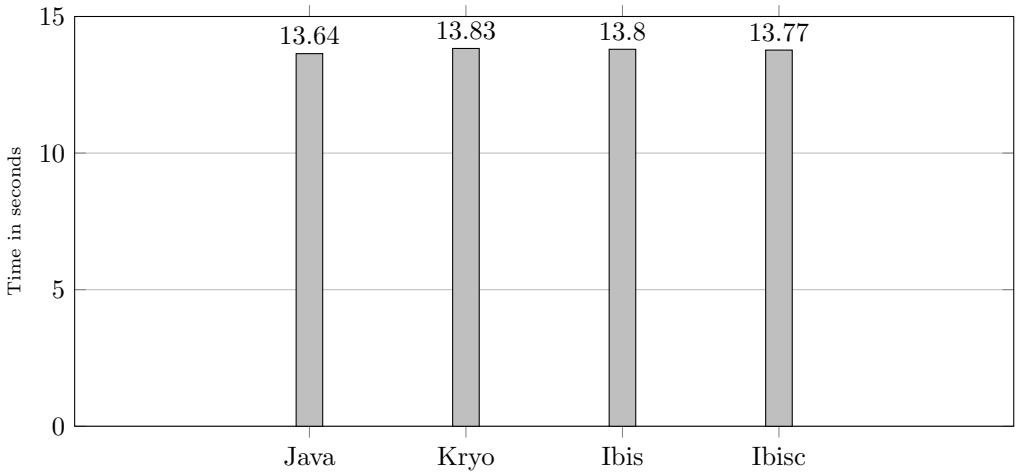
Figure 5: TeraSort memory usage during execution



5.2 SparkPi

The results of the Sparkpi benchmark are shown in figure 6. The results of this benchmark show that the differences between serialization types, when performing a mostly computational workload, are very small. Java serialization seems to be slightly faster. A possible explanation for this could be that Java is the only native serialization method. However, we can not conclusively say whether this is the case, or if Java serialization is faster because the differences are only very slight. The difference could also be caused by deviation. The RSD of the 200 application runs was measured to be between 3.16% and 3.64% depending on the serialization type.

Figure 6: SparkPi time to completion



More detailed results of the SparkPi benchmark are shown in table 4. As well as more precise

Table 4: SparkPi time to completion in milliseconds

	Java	**Kryo**	**Ibis**	**Ibisc**
Mean	13584 ms	13865 ms	14004 ms	13903 ms
STD	428 ms	465 ms	510 ms	430 ms
RSD	3.16%	3.35%	3.64%	3.09%

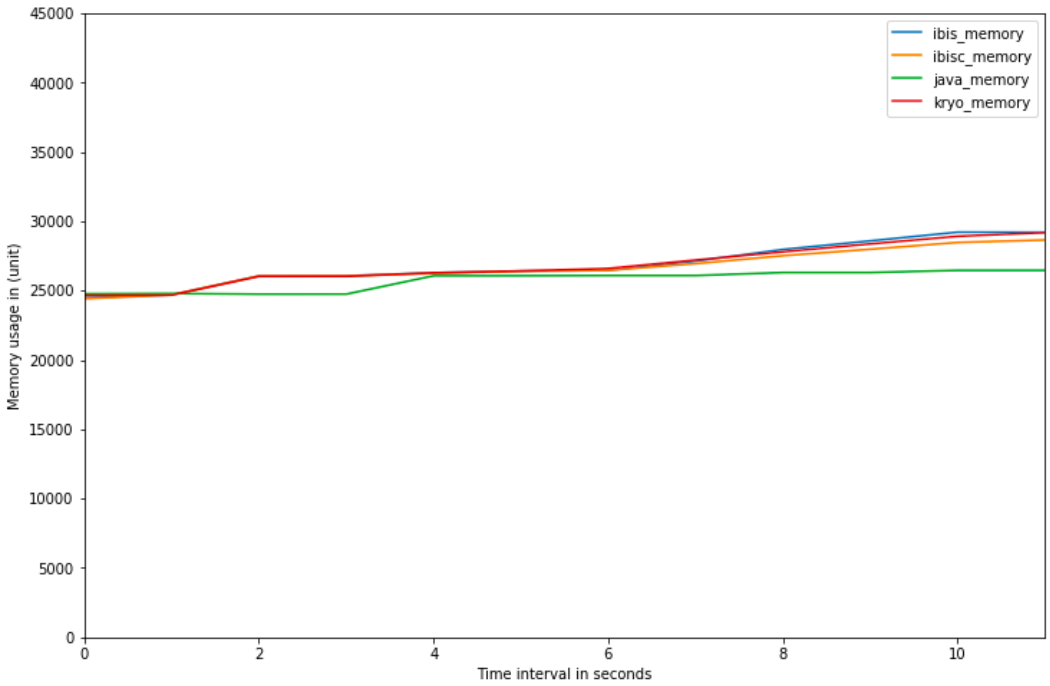
In table 5, the amount of serialization calls per component is shown. We can see that the majority of serialization calls are still being made to Java serialization.

Table 5: Amount of serialization calls per component in one SparkPi benchmark run

Component	**Ibis serialization calls**	**Java serialization calls**
TaskResultGetter.scala	2	0
TaskResult.scala	3	0
ClosureCleaner.scala	3	0
DAGScheduler.scala	2	0
TorrentBroadcast.scala	1	0
TaskSetManager.scala	2	0
NettyRpcEnv.scala	0	134
Total	**13**	**134**

The average memory utilization of SparkPi during a single benchmark run is shown in figure 7. It shows, just like the performance benchmark that Java serialization has a very slight advantage. The non-native serialization methods are all very close to each other, so we cannot conclusively say which non-native serialization type has a clear advantage.

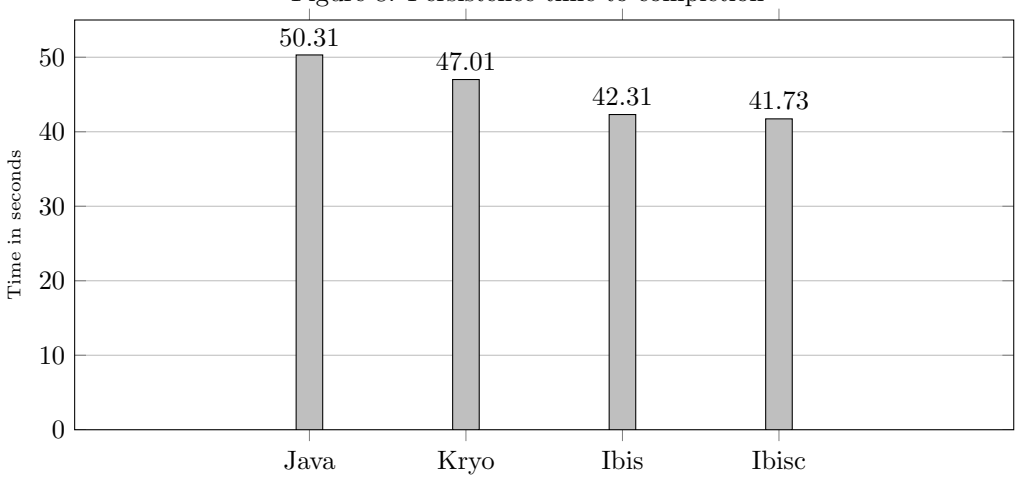
Figure 7: SparkPi memory usage during execution



5.3 Persistence

The results of the persistence test, as described in the approach section, are shown in figure 8. The results show roughly the same distribution as the TeraSort benchmark. Completing a single run of the Spark application using Java serialization took roughly 50 seconds to complete, Kryo 47, while the Ibis versions are both around 42 seconds. Ibis compiled again shows to be only slightly faster than the uncompiled version.

Figure 8: Persistence time to completion



In table 6 the amount of milliseconds it takes for one benchmark to complete is shown. The relative standard deviation (RSD) of Java and Kryo is slightly higher than those of the Ibis variants, however, we can still see that Ibisc is roughly 10 to 12% faster when compared to Java serialization. Kryo, just like in the TeraSort benchmark is slightly faster than Java serialization and slightly slower when compared to Ibis and Ibisc.

Table 6: Persistence benchmark, time to completion in milliseconds.

	Java	**Kryo**	**Ibis**	**Ibisc**
Mean	50312 ms	47008 ms	42309 ms	41733 ms
STD	1015.72 ms	936.05 ms	758.75 ms	562.39 ms
RSD	2.02%	1.99%	1.79%	1.35%

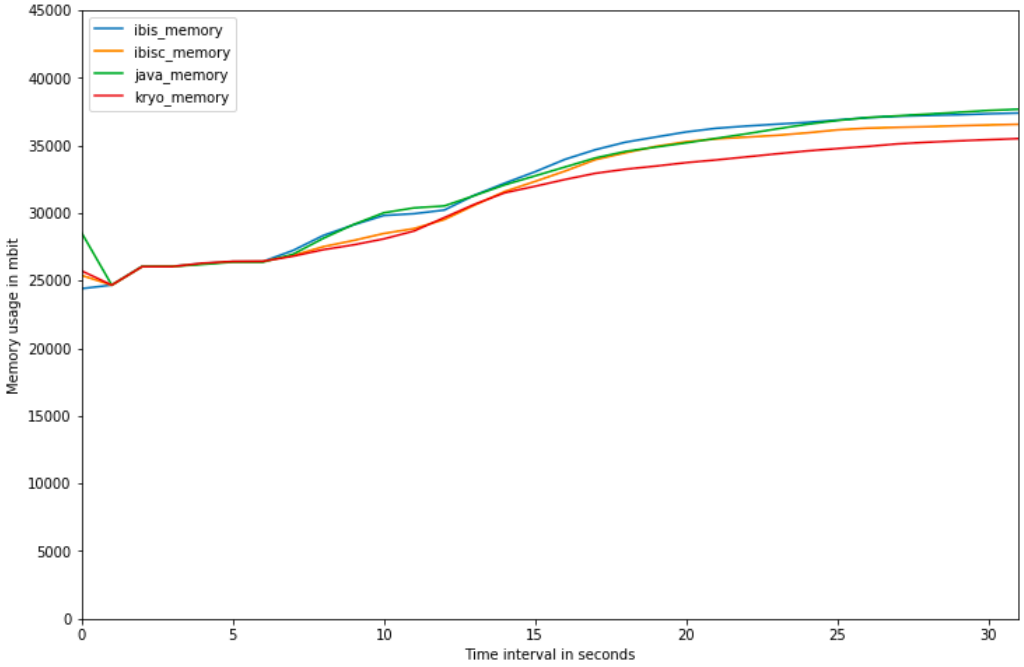
The amount of serialization calls used in one run of the persistence benchmark, is shown in table 7. The persistence benchmark by far makes the most serialization calls. It also shows that even though most components were replaced with Ibis, the Netty backed block transfer service still uses Java serialization due to unresolved incompatibilities discussed in 4.2.4. This use of Java serialization accounts for more than twice the serialization calls.

Table 7: Amount of serialization calls per component in one persistence benchmark run

Component	Ibis serialization calls	Java serialization calls
ClosureCleaner.scala	102	0
DAGScheduler.scala	100	0
TorrentBroadcast.scala	50	0
TaskSetManager.scala	320	0
TaskResultGetter.scala	295	0
TaskResult.scala	679	0
NettyRpcEnv.scala	0	3453
Total	**1546**	**3453**

The memory utilization per serialization type can be seen in figure 9. It shows that Kryo uses the least amount of memory on this benchmark peaking around 35000mbit. Ibisc has a slightly higher memory peak, at around 36000mbit. Both Java and uncompiled Ibis serialization use the most amount of memory, both peaking around 37500mbit.

Figure 9: TeraSort memory usage during execution



6 Discussion

This research has shown that the performance of certain Spark benchmarks could be improved by partially replacing Java and Kryo serialization with Ibis serialization. The modified version of Spark, that used Ibis serialization in most components, completed benchmarks that heavily rely on serialization faster than the original Spark version. However, the time to complete a benchmark is not the only indicator of performance. This research also measured the memory utilization, we can however not conclusive say which serialization type has a clear advantage in terms of memory utilization. This research has not taken the amount of CPU-cycles and the number of bytes sent over the network into account.

Because of the positive impact, without the requirement to register classes like Kryo serialization, Ibis serialization could potentially be used as a default serialization method in Spark. However before this is possible, limitations such as not being able to write to the Hadoop File System should first be resolved. Our research has only measured the performance on a limited part of the Spark API, future research is required to determine the effects of the new serialization methods in different benchmarks and APIs as well.

The release of Spark 1.2 had the network layer replaced with a Netty based one and it introduced performance and scalability improvements using techniques like Zero-copy I/O and off-heap network buffer management [34]. We suspect that these techniques, especially the off-heap network buffer management are responsible for the *StreamCorruptedException* we observed when the Ibis serializer is used with the Netty backed block transfer service.

Regarding the incompatibility with retrieving persisted files, we suspect that the root cause of the exception is due to the implementation of both the *writeObject* and *readObject* methods in *SerializableWritable.scala* class. These implementations assume Java serialization is in use and hence they directly make use of *defaultWriteObject* method and *defaultReadObject* of the *ObjectOutputStream* and *ObjectInputStream* respectively. The direct use of the *ObjectWritable* from *Hadoop* is also problematic since its deserialization logic is not aware of the newly introduced Ibis serialization.

This research has made it possible for Ibis serialization to be used in Scala applications. This opens up even more opportunities to experiment with using Ibis serialization in distributed applications. Experimenting with Ibis serialization in other distributed (big data) applications could potentially also show to have a positive impact. Because of the large scale that big data applications operate in, small improvements in areas such as serialization could have a large impact on both the total cost and even possibilities.

7 Conclusion

At the beginning of this research, we set out to see whether it was possible to improve Spark's performance by introducing Ibis serialization. To do this, we first had to understand the components of Spark that can benefit the most from Ibis serialization, then how to integrate Ibis serialization into Spark, and finally observe if there were any performance differences.

Our research has shown that it is possible to integrate Ibis serialization in Spark, as we were able to do so in 15 out of the 17 classes where we found direct serialization calls. Additionally, in the course of our research, we explored how Spark executes programs. The tracing functionality we added to the modified version of Spark, allowed us to see the serialization calls that occur during the execution of a Spark application. We found out that serialization occurs at components involved in the transformation of a Spark application into tasks. Serialization also occurs during two classes of network communications. One is the communication needed to keep the Spark components in a cluster. The other being communication needed to send tasks to worker nodes where they are executed. Hence Spark will benefit the most with a better serialization in these identified places.

We were able to use Ibis serialization in the components involved in the transformation of a Spark application into tasks. This can be seen in the serialization calls per component tables presented in section 5. We did not use Ibis for serialization in one of the network communication layers, due to unresolved compatibility we discussed in section 6.

Regarding the performance measurements, Based on the results of the benchmarks, we can conclude Ibis serialization can reduce the execution time of serialization heavy benchmarks by 10% to 15%. We can also conclude that Ibis serialization does not have a notable impact on the performance of the computational benchmark, SparkPi.

The memory utilization of the serialization types depends on the benchmark. Java serialization used the least amount of memory in the benchmark with the least amount of serialization calls. We cannot conclusively say which serialization method uses the least amount of memory in serialization heavy workloads.

8 Future Work

Our research has shown that Ibis serialization can have a positive impact on the performance of certain Spark jobs, however, we have only measured the impact when using two servers. It would be interesting to perform benchmarks on a larger scale to see if more inter-server communication has an impact on the performance difference. Furthermore, It would be interesting to perform more and different benchmarks to see how Ibis serialization performs in different situations.

It was not possible to successfully use Ibis serialization in the Netty block transfer service and with file persistence to Hadoop. Future research could focus on resolving the incompatibilities that prevented the use of Ibis serialization in this section of Spark and measure if there are any performance differences.

This research has only measured the performance impact of Ibis serialization using the RDD API. As mentioned in section 3.1, it is one of the available APIs in Spark. It would be interesting for future work to compare the performance of the DataSet API, which by default, uses a binary file format [35] for serialization with a modified version that uses Ibis serialization.

This research has measured the time to completion, as well as the memory utilization of the entire Spark application. Future research is necessary to gain insight into other performance indicators, such as network and CPU utilization. It could also help to perform the performance measurements on individual components instead of entire applications.
The Ibis software stack comprises of various components, of which, the serialization layer is one part. It might also be interesting to see how other components of the Ibis stack can be introduced into Spark and measure the performance differences such replacements lead to.

References

[1] *Apache Spark^{TM}; - Unified Analytics Engine for Big Data.* URL: https://spark.apache.org/ (visited on 01/07/2020).

[2] Matei Zaharia et al. "Spark: Cluster computing with working sets." In: *HotCloud* 10.10-10 (2010), p. 95.

[3] *Tuning - Spark 2.4.4 Documentation.* URL: https://spark.apache.org/docs/latest/tuning.html (visited on 01/07/2020).

[4] *Serialable (Java Platform SE 8).* URL: https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html (visited on 01/16/2020).

[5] *Github - Kryo: Java binary serialization and cloning.* URL: https://github.com/EsotericSoftware/kryo (visited on 01/16/2020).

[6] *Ibis.* URL: https://www.cs.vu.nl/ibis/ipl.html (visited on 01/07/2020).

[7] Xiaoyi Lu et al. "High-performance design of apache spark with RDMA and its benefits on various workloads". In: *2016 IEEE International Conference on Big Data (Big Data)*. IEEE. 2016, pp. 253–262.

[8] Xiaoyi Lu et al. "Accelerating spark with rdma for big data processing: Early experiences". In: *2014 IEEE 22nd Annual Symposium on High-Performance Interconnects*. IEEE. 2014, pp. 9–16.

[9] Rob V Van Nieuwpoort et al. "Ibis: a flexible and efficient Java-based Grid programming environment". In: *Concurrency and Computation: Practice and Experience* 17.7-8 (2005), pp. 1079–1107.

[10] Nicholas Palmer, Thilo Kielmann, and Henri Bal. "Serialization for ubiquitous systems: An evaluation of high performance techniques on java micro edition". In: *2008 The Second International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*. IEEE. 2008, pp. 356–361.

[11] *Java Platform, Micro Edition (Java ME).* URL: https://www.oracle.com/java/technologies/javameoverview.html (visited on 02/08/2020).

[12] *A Tale of Three Apache Spark APIs: RDDs vs DataFrames and Datasets.* URL: https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html (visited on 02/05/2020).

[13] *Catalyst optimizer - Databricks.* URL: https://databricks.com/glossary/catalyst-optimizer (visited on 01/16/2020).

[14] *DAGScheduler Source Code Documentation.* URL: https://github.com/apache/spark/blob/094563384478a402c36415edf04ee7b884a34fc9/core/src/main/scala/org/apache/spark/scheduler/DAGScheduler.scala#L47 (visited on 01/24/2020).

[15] *Task Source Code Documentation.* URL: https://github.com/apache/spark/blob/e1ea806b3075d279b5f08a29fe4c1ad6d3c4191a/core/src/main/scala/org/apache/spark/scheduler/Task.scala#L33 (visited on 01/24/2020).

[16] Michael Armbrust et al. "Spark SQL: Relational Data Processing in Spark". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. 2015, 1383–1394.

[17] *Task Schedular Code Documentation.* URL: https://github.com/apache/spark/blob/7955b3962ac46b89564e0613db7bea98a1478bf2/core/src/main/scala/org/apache/spark/scheduler/TaskScheduler.scala#L25 (visited on 01/24/2020).

[18] *Understand RDD Operations: Transformations and Actions.* URL: https://trongkhoanguyen.com/spark/understand-rdd-operations-transformations-and-actions/ (visited on 01/30/2020).

[19] R Johnson J Vlissides E Gamma R Helm. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[20] *Apache Spark modified to use Ibis serialization*. URL: https://github.com/sne-os3-rp1/spark (visited on 01/27/2020).

[21] *Modified Ibis serialization*. URL: https://github.com/sne-os3-rp1/ipl (visited on 01/27/2020).

[22] *sne-os3-rp1 - Github*. URL: https://github.com/sne-os3-rp1 (visited on 01/24/2020).

[23] *Scala Match error*. URL: https://www.scala-lang.org/api/current/scala/MatchError.html (visited on 01/29/2020).

[24] *Netty*. URL: https://netty.io/ (visited on 02/05/2020).

[25] Owen O'Malley. "Terabyte sort on apache hadoop". In: *Yahoo, available online at: http://sortbenchmark. org/Yahoo-Hadoop. pdf,(May)* (2008), pp. 1–3.

[26] Songze Li et al. "Coded terasort". In: *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2017, pp. 389–398.

[27] *RDD Programming - Spark 2.4.4 Documentation*. URL: https://spark.apache.org/docs/latest/rdd-programming-guide.html#rdd-persistence (visited on 01/16/2020).

[28] *Spark/SparkPi.scala - Github*. URL: https://github.com/apache/spark/blob/master/examples/src/main/scala/org/apache/spark/examples/SparkPi.scala (visited on 01/29/2020).

[29] Nicholas Metropolis and Stanislaw Ulam. "The monte carlo method". In: *Journal of the American statistical association* 44.247 (1949), pp. 335–341.

[30] *Apache Hadoop 3.2.1 - Yarn*. URL: https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html (visited on 01/16/2020).

[31] *Apache Hadoop*. URL: https://hadoop.apache.org/ (visited on 01/16/2020).

[32] *Oracle Java SE Support Roadmap*. URL: https://www.oracle.com/technetwork/java/java-se-support-roadmap.html (visited on 02/05/2020).

[33] *Build and Run Spark on JDK11*. URL: https://issues.apache.org/jira/browse/SPARK-24417 (visited on 02/05/2020).

[34] Michael Armbrust et al. "Scaling spark in the real world: performance and usability". eng. In: *Proceedings of the VLDB Endowment* 8.12 (2015-08-01), pp. 1840,1843. ISSN: 21508097.

[35] *Introducing Apache Spark Datasets*. URL: https://databricks.com/blog/2016/01/04/introducing-apache-spark-datasets.html (visited on 02/06/2020).

9 Appendix

9.0.1 Location of direct serialization calls in Apache Spark

Table 8: Components replaced with Ibis

Source file	Description
Task.scala	Takes care of serializing and deserializing metrics for executing tasks
Accumulable.scala	Represents a data type that can be accumulated
RangePartitioner.scala	Partition sortable records of RDD by range into roughly equal ranges
TorrentBroadcast.scala	A torrent like implementation of a broadcast variable
FileSystemPersistenceEngine.scala	This part of the code takes care of allowing Master to persist any state that is necessary to recover from a failure
ZooKeeperPersistenceEngine.scala	Same as FileSystemPersistenceEngine but when using Zookeeper
CoarseGrainedExecutorBackend.scala	An RPC endpoint that defines what functions to trigger given a message
Executor.scala	The executor that uses a given scheduler to run tasks
BlockStoreShuffleReader.scala	Fetches partitions in ranges from a shuffle
DiskBlockObjectWriter.scala	A class for writing JVM objects directly to a file on disk
UnsafeShuffleWriter.scala	An optimized shuffle manager that powers the tungsten-sort
expressions.objects.scala	Serialization that is part of the Catalyst SQL optimizer
MemoryStore.scala	Stores blocks in memory, as Arrays of deserialized Java objects or as serialized ByteBuffers
ClosureCleaner.scala	A cleaner that renders closures serializable if they can be done so safely
ExternalAppendOnlyMap.scala	An append-only map that spills sorted content to disk
ExternalSorter.scala	Sorts and potentially merges a number of key-value pairs
TaskSetManager.scala	Schedules tasks in a single TaskSet in the TaskSchedulerImpl

Table 9: Components not replaced with Ibis

Source file	Description
Netty backed network subcomponent	Netty related components for internal RPC communications
Hadoop subcomponent for deserializing/serializing	Hadoop related components in the sql/core module

9.0.2 Apache Spark's serialization interfaces and Ibis implementations

Listing 4: Serializer and Ibis implementation

```
abstract class Serializer {
  @volatile protected var defaultClassLoader: Option[ClassLoader] = None
  def setDefaultClassLoader(classLoader: ClassLoader): Serializer = {
    defaultClassLoader = Some(classLoader)
    this
  }
  def newInstance(): SerializerInstance
  @Private
  private[spark] def supportsRelocationOfSerializedObjects: Boolean = false
}

class IbisSerializer(conf: SparkConf) extends Serializer with Externalizable {
  private var counterReset = conf.getInt("spark.serializer.objectStreamReset", 100)
  private var extraDebugInfo = conf.getBoolean("spark.serializer.extraDebugInfo",
      true)
  protected def this() = this(new SparkConf())
  override def newInstance(): SerializerInstance = {
    val classLoader = defaultClassLoader.getOrElse(Thread.currentThread.
        getContextClassLoader)
    new IbisSerializerInstance(counterReset, extraDebugInfo, classLoader)
  }
  override def writeExternal(out: ObjectOutput): Unit = Utils.tryOrIOException {
    out.writeInt(counterReset)
    out.writeBoolean(extraDebugInfo)
  }
  override def readExternal(in: ObjectInput): Unit = Utils.tryOrIOException {
    counterReset = in.readInt()
    extraDebugInfo = in.readBoolean()
  }
}
```

Listing 5: SerializerInstance and Ibis implementation

```
abstract class SerializerInstance {
  def serialize[T: ClassTag](t: T): ByteBuffer
  def deserialize[T: ClassTag](bytes: ByteBuffer): T
  def deserialize[T: ClassTag](bytes: ByteBuffer, loader: ClassLoader): T
  def serializeStream(s: OutputStream): SerializationStream
  def deserializeStream(s: InputStream): DeserializationStream
}

private[spark] class IbisSerializerInstance(counterReset: Int,
                                            extraDebugInfo: Boolean,
                                            defaultClassLoader: ClassLoader)
  extends SerializerInstance {

  override def serialize[T: ClassTag](t: T): ByteBuffer = {
    val bos = new ByteArrayOutputStream()
    val out = serializeStream(bos)
    out.writeObject(t)
    out.close()
    ByteBuffer.wrap(bos.toByteArray)
  }

  override def deserialize[T: ClassTag](bytes: ByteBuffer): T = {
    val bis = new ByteArrayInputStream(byteBufferToByteArray(bytes))
    val in = deserializeStream(bis)
    in.readObject()
  }

  override def deserialize[T: ClassTag](bytes: ByteBuffer, loader: ClassLoader): T={
    val bis = new ByteArrayInputStream(byteBufferToByteArray(bytes))
    val in = deserializeStream(bis, loader)
    in.readObject()
  }

  override def serializeStream(s: OutputStream): SerializationStream = {
    new IbisSerializationStream(s, counterReset, extraDebugInfo)
  }

  override def deserializeStream(s: InputStream): DeserializationStream = {
    new IbisDeserializationStream(s, defaultClassLoader)
  }

  def deserializeStream(s: InputStream, loader: ClassLoader): DeserializationStream
      = {
    new IbisDeserializationStream(s, loader)
  }

  private[this] def byteBufferToByteArray(bytes: ByteBuffer):Array[Byte] = {
    if (bytes.hasArray) {
      bytes.array()
    } else {
      val bytesArray= Array.fill[Byte](bytes.remaining())(0)
      bytes.get(bytesArray, 0, bytesArray.length)
      bytesArray
    }
  }
}
```

Listing 6: SerializationStream and Ibis implementation

```
abstract class SerializationStream extends Closeable {
  def writeObject[T: ClassTag](t: T): SerializationStream
  def writeKey[T: ClassTag](key: T): SerializationStream = writeObject(key)
  def writeValue[T: ClassTag](value: T): SerializationStream = writeObject(value)
  def flush(): Unit
  override def close(): Unit
  def writeAll[T: ClassTag](iter: Iterator[T]): SerializationStream = {
    while (iter.hasNext) {
      writeObject(iter.next())
    }
    this
  }
}

private[spark] class IbisSerializationStream(out: OutputStream,
                                             counterReset: Int,
                                             extraDebugInfo: Boolean) extends
                                                 SerializationStream {

  private val objOut = new IbisSerializationOutputStream(
    new BufferedArrayOutputStream(out)
  )
  private var counter = 0
  override def writeObject[T: ClassTag](t: T): SerializationStream = {
    try {
      objOut.writeObject(t)
      flush()
    } catch {
      case e: NotSerializableException if extraDebugInfo =>
        throw SerializationDebugger.improveException(t, e)
    }
    counter += 1
    if (counterReset > 0 && counter >= counterReset) {
      objOut.reset()
      counter = 0
    }
    this
  }

  def flush(): Unit = { objOut.flush() }
  def close(): Unit = {
    objOut.realClose()
  }
}
```

Listing 7: DeserializationStream and Ibis implementation

```
abstract class DeserializationStream extends Closeable {
  def readObject[T: ClassTag](): T
  def readKey[T: ClassTag](): T = readObject[T]()
  def readValue[T: ClassTag](): T = readObject[T]()
  override def close(): Unit
  def asIterator: Iterator[Any] = new NextIterator[Any] {
    override protected def getNext() = {
      try {
        readObject[Any]()
      } catch {
        case eof: EOFException =>
          finished = true
          null
      }
    }
    override protected def close() {
      DeserializationStream.this.close()
    }
  }
  def asKeyValueIterator: Iterator[(Any, Any)] = new NextIterator[(Any, Any)] {
    override protected def getNext() = {
      try {
        (readKey[Any](), readValue[Any]())
      } catch {
        case eof: EOFException =>
          finished = true
          null
      }
    }
   override protected def close() {
      DeserializationStream.this.close()
    }
  }
}

 private val objIn = new IbisSerializationInputStream(new BufferedArrayInputStream(
     in)) {
     def resolveClass(desc: ObjectStreamClass): Class[_] =
     try {
       Class.forName(desc.getName, false, loader)
     } catch {
       case e: ClassNotFoundException =>
         JavaDeserializationStream.primitiveMappings.getOrElse(desc.getName, throw e
             )
     }
    }
  override def readObject[T: ClassTag](): T = {
    SLogger.log("ibis","read")
    objIn.readObject().asInstanceOf[T]
  }
  override def close(): Unit = {
    objIn.close()
  }
}
```

Listing 8: File persistence stack trace

```
WARN scheduler.TaskSetManager: Lost task 0.0 in stage 1.0 (TID 1, localhost,
    executor driver): java.io.IOException: ibis.io.SerializationError: require byte
    []: org.apache.hadoop.mapred.FileSplit4file:/Us
       at org.apache.spark.util.Utils$.tryOrIOException(Utils.scala:1333)
       at org.apache.spark.SerializableWritable.readObject(SerializableWritable.
           scala:41)
       at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
       at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java
           :62)
       at sun.reflect.DelegatingMethodAccessorImpl.invoke(
           DelegatingMethodAccessorImpl.java:43)
       at java.lang.reflect.Method.invoke(Method.java:498)
       at ibis.io.AlternativeTypeInfo.invokeReadObject(AlternativeTypeInfo.java:532)
       at ibis.io.IbisSerializationInputStream.alternativeReadObject(
           IbisSerializationInputStream.java:1402)
       at ibis.io.AlternativeTypeInfo$SerializableReader.readObject(
           AlternativeTypeInfo.java:245)
       at ibis.io.IbisSerializationInputStream.doReadObject(
           IbisSerializationInputStream.java:1716)
       at ibis.io.IbisSerializationInputStream.readFieldObject(
           IbisSerializationInputStream.java:1205)
```

Listing 9: File persistence stack trace

```
Caused by: java.lang.RuntimeException: java.io.StreamCorruptedException: invalid
    stream header: 00005300
       at java.io.ObjectInputStream.readStreamHeader(ObjectInputStream.java:862)
       at java.io.ObjectInputStream.<init>(ObjectInputStream.java:354)
       at org.apache.spark.serializer.JavaDeserializationStream$$anon$1.<init>(
           JavaSerializer.scala:64)
       at org.apache.spark.serializer.JavaDeserializationStream.<init>(
           JavaSerializer.scala:64)
       at org.apache.spark.serializer.JavaSerializerInstance.deserializeStream(
           JavaSerializer.scala:126)
       at org.apache.spark.serializer.JavaSerializerInstance.deserialize(
           JavaSerializer.scala:111)
       at org.apache.spark.rpc.netty.
           NettyRpcEnv$$anonfun$deserialize$1$$anonfun$apply$1.apply(NettyRpcEnv.
           scala:271)
   ...
       at org.apache.spark.network.server.TransportRequestHandler.processRpcRequest(
           TransportRequestHandler.java:180)
       at org.apache.spark.network.server.TransportRequestHandler.handle(
           TransportRequestHandler.java:103)
       at org.apache.spark.network.server.TransportChannelHandler.channelRead(
           TransportChannelHandler.java:118)
       at io.netty.channel.AbstractChannelHandlerContext.invokeChannelRead(
           AbstractChannelHandlerContext.java:362)
       at io.netty.channel.AbstractChannelHandlerContext.invokeChannelRead(
           AbstractChannelHandlerContext.java:348)
       at io.netty.channel.AbstractChannelHandlerContext.fireChannelRead(
           AbstractChannelHandlerContext.java:340)
       at io.netty.handler.timeout.IdleStateHandler.channelRead(IdleStateHandler.
           java:286)
```