MPI4Spark Meets YARN: Enhancing MPI4Spark through YARN support for HPC

Kinan Al-Attar
Department of Computer Science Engineering
The Ohio State University
alattar.2@osu.edu

Aamir Shafi
Department of Computer Science Engineering
The Ohio State University
shafi.16@osu.edu

Hari Subramoni
Department of Computer Science Engineering
The Ohio State University
subramoni@cse.ohio-state.edu

Dhabaleswar K. Panda
Department of Computer Science Engineering
The Ohio State University
panda@cse.ohio-state.edu

2023 IEEE International Conference on Big Data (BigData) | 979-8-3503-2445-7/23/$31.00 ©2023 IEEE | DOI: 10.1109/BigData59044.2023.10386120

Abstract—**The MPI4Spark effort was able to reconcile disparities that existed between High-Performance Computing (HPC) environments and Big Data stacks, by adopting an MPI-based solution inside of Apache Spark's Netty communication layer that was capable of better utilizing high-speed interconnects — such as InfiniBand (IB), Intel Omni-Path (OPA), and HPE Slingshot — across a variety of HPC systems. Apache Spark provides support for several cluster managers, such as YARN, Mesos, and Kubernetes, besides its internal standalone cluster manager. MPI4Spark, however, does not support the YARN cluster manager, instead only relying on Spark's internal standalone cluster manager. The YARN cluster manager is designed for running large-scale clusters up to hundreds of nodes and provides better scalability in an HPC environment. Therefore, support for the YARN cluster manager is needed for MPI4Spark to provide a solution more fitting for HPC in terms of scalability — this paper addresses this problem. We present a new design for MPI4Spark that supports both YARN and the internal standalone cluster manager. The architectural framework of MPI4Spark remains the same in the new YARN design with an MPI-based Netty layer at its core. The new YARN design for MPI4Spark outperforms both regular Spark and RDMA-Spark. Evaluation of MPI4Spark's new YARN design was conducted on two HPC systems, TACC Frontera and TACC Stampede2. On Frontera, looking at *SortByTest* weak-scaling numbers, and cluster size of 64 *NodeManagers* (3584 cores, 896GB), MPI4Spark outperforms in total execution time both Spark by 4.52x and RDMA-Spark by 2.33x. For *GroupByTest* strong scaling numbers, and cluster size of 128 $NodeManagers$ (7168 cores, 1344GB), MPI4Spark performs better than Spark by 3.29x and by 2.32x compared to RDMA-Spark. With Intel HiBench performance evaluations on Frontera, on a cluster size of 32 *NodeManagers* (1792 cores), MPI4Spark fairs better than Spark by 1.91x for the Logistic Regression (LR) benchmark. On Stampede2, Speed-ups for the overall total execution time for 192GB are 1.98x compared to IPoIB for *GroupByTest*, and for *SortByTest*, 1.89x. For strong scaling we see MPI4Spark outperforming Spark, on average, by about 1.73x using OHB benchmarks.**

Index Terms—**YARN, Apache Spark, Netty, MPI**

1

I. Introduction

The intersection of Big Data and High-Performance Computing (HPC) is becoming more pronounced as data continues to grow rapidly, and although solutions such as Big Data frameworks like Apache Spark [1] exist to solve challenges with processing and managing large sets of data in a parallel and distributed fashion, they do not fully utilize the main features that characterize HPC systems and environments such as high-speed interconnects and the Message Passing Interface (MPI) [2] programming model — the lingua franca programming model for developing parallel scientific and engineering applications on HPC.

MPI4Spark [3], developed at the Ohio State University (OSU) by the Network Based Computing Lab (NOWLAB), is a solution that aims at providing a ”Converged Communication Stack“ for HPC and Big Data, by utilizing a MPI-based design that is geared towards HPC — supporting multiple interconnects such as InfiniBand (IB) [4], Intel Omni-Path (OPA) [5], and HPE Slingshot [6]. MPI4Spark was able to outperform Apache Spark [1] and RDMA-Spark [7] by relying on MPI for the communication of shuffle data at the Netty [8] layer of Spark (the communication backend of Spark). However, MPI4Spark has its limitations, as it only supports the standalone internal cluster manager.

When run in a cluster environment, Apache Spark can support a number of different cluster managers, such as YARN [9], Mesos [10], and Kubernetes [11]. Spark also supports an internal standalone cluster manager. As of now, the MPI4Spark effort only supports the standalone cluster manager. In this paper, we present a new design that encompasses both the standalone cluster manager and the YARN cluster manager for MPI4Spark.

A. Motivation

Ideally, MPI4Spark should support all cluster managers that Apache Spark supports. This, however, is not the main motivating aspect for adding support for the YARN cluster manager in MPI4Spark. As mentioned earlier, the YARN cluster manager is able to run and manage large-scale clusters for Spark. The standalone internal cluster manager that Spark provides, and that MPI4Spark currently supports, does not offer the capabilities that YARN provides in terms of scalability. In that regards, support for the YARN cluster manager is needed to provide a more scalable solution in MPI4Spark.

The main motivation behind this work is to provide a solution that better accommodates the intersection of Big Data and HPC. This is realized by expanding the MPI4Spark effort and adding support for the YARN cluster manager to provide better scalability in HPC environments.

B. Problem Statements

In introducing support for the YARN cluster manager in MPI4Spark, several questions were relevant. In this paper, we consider the following questions:

- Is MPI4Spark's standalone cluster manager design ideal? Can support for the YARN cluster manager avoid pitfalls in MPI4Spark's standalone cluster manager design?
- The YARN cluster manager relies on containers and launcher scripts to launch executors for Spark. Can this be mapped to an MPI-based solution where executors are launched through MPI?
- How does the performance of MPI4Spark with YARN compare with regular Spark and RDMA-Spark on different HPC systems such as TACC Frontera and TACC Stampede2?

These questions are later discussed and answered in Section III.

C. Overview

In this paper, we present a new design for MPI4Spark that supports multiple cluster managers, the YARN cluster manager and Spark's internal standalone cluster manager. Support for the YARN cluster manager is needed as it is designed for launching large-scale clusters (up-to hundreds of nodes). MPI4Spark, as a design that accomplishes the goal of a "Converged Communication Stack" for HPC and Big Data, should support the YARN cluster manager to better provide scalable solutions in HPC. We focus on solving this limitation that MPI4Spark has.

We present performance evaluation numbers with larger cluster sizes with MPI4Spark on two HPC systems (TACC Frontera and TACC Stampede2) and note that MPI4Spark overall performs better than Spark and RDMA-Spark. In the MPI4Spark paper [3], we presented numbers for a cluster size up-to 32 worker nodes (using the standalone cluster manager). With the new YARN design, we present numbers for 64 *NodeManager* and 128 *NodeManager* nodes as well. We also use the same benchmarking packages — the OSU HiBD Benchmarks (OHB) package [12] and the Intel HiBench benchmark suite [13] — used in the original MPI4Spark paper in evaluating the performance of our new MPI4Spark YARN design, and compare the performance against Spark and RDMA-Spark.

On the Frontera HPC system, *GroupByTest* weak-scaling numbers with 64 *NodeManagers*, we see that MPI4Spark performs better than regular Spark by 3.8x and RDMA-Spark by 2.5x. While for the same configuration for *SortByTest*, MPI4Spark outperforms regular Spark by 4.5x and RDMA-Spark by 2.3x. For *GroupByTest* strong-scaling numbers, 128 *NodeManagers*, MPI4Spark fairs better than regular Spark by 3.3x, and RDMA-Spark by 2.3x. Looking at the same configuration, and for *SortByTest* and 64 *NodeManagers*, MPI4Spark performs better by 2x against regular Spark, and by 1.2x against RDMA-Spark. For performance evaluations with the Intel HiBench suite, on a cluster size of 32 *NodeManagers* (1792 cores), MPI4Spark outperforms Spark by 1.91x for the Logistic Regression (LR) machine learning benchmark. On Stampede2, we see that MPI4Spark performs better than Spark for the *GroupByTest* benchmark by a factor of 1.98x for cluster size 16 *NodeManagers* and data size 192GB. While for the *SortByTest* benchmark, for the same cluster and data sizes, we see MPI4Spark outperforms Spark by 1.89x. For strong scaling we see MPI4Spark outperforming Spark, on average, by about 1.73x using *GroupByTest* and *SortByTest* OHB benchmarks. More information about the performance evaluation can be found in Section V.

D. Contributions

The paper makes the following contributions:

1) The paper presents a new design for the MPI4spark effort carried that is capable of supporting both the standalone and YARN cluster manager. The new YARN-based effort provides a solution that better accommodates the intersection of Big Data and HPC through YARN's inherent scalable design. The architecture of MPI4Spark remains unchanged with Netty MPI at its core.
2) The new YARN design avoids pitfalls in the standalone cluster manager design in MPI4Spark. Since shuffle communication takes place between executors, only executor processes are launched within an MPI environment. This paves the way for future implementations of basic fault-tolerance in MPI4Spark. In the standalone design, the entire Spark cluster is launched inside an MPI environment.
3) The paper evaluates the performance of the new YARN design on the TACC Frontera HPC system against the RDMA-Spark effort and regular regular Spark using the OSU HiBD Benchmarks (OHB) package and the Intel HiBench benchmarks suite. The evaluation is carried out on different sized clusters up-to a $7,168$-core cluster with a total of 128 *NodeManagers*. Detailed performance evaluation can be found in Section V.

The remaining sections of the paper are organized in the following fashion. Section II refers to the background section which covers some concepts in YARN along with MPI4Spark. The next sections, Sections III and IV, refer to the challenges encountered and proposed design and implementation details. Section V presents the performance evaluation of MPI4Spark+YARN. Finally, section VII concludes and covers the future work for the MPI4Spark project.

II. Background

MPI4Spark is an effort carried out at the Ohio State University (OSU) by the Network Based Computing Lab (NOWLAB) that optimizes Apache Spark's shuffle engine by utilizing production-quality MPI libraries at the Netty layer (MVAPICH [14]). **Netty** is a network application framework that is asynchronous and event-based, and is used by Spark to communicate RPC messages and shuffle data. By default, Netty uses a Java New IO (NIO) transport. The NIO transport utilizes a selector thread (single thread) to manage multiple Java socket channels.

Apache Spark is an open-source Big Data framework developed in UC Berkley by the AMPLab. It processes data in-memory and relies on Resilient Distributed Datasets (RDDs) which are fault-tolerant distributed data. User applications in Spark run as JVM independent processes, managed by the *SparkContext* object. The *SparkConext* object resides in the main program — called the driver program — and can connect to different types of cluster managers, such as Mesos, YARN, Kubernetes, and Spark's internal standalone cluster manager.

The role of the cluster manager is to allocate resources across the cluster for user applications. Once the *SparkContext* object is connected to a cluster manager, executors are launched on the cluster to execute a given application. Meta information regarding a given application, such as JAR or Python files, is passed to the executors. Tasks are then created by the *SparkContext* and sent out to the executors for execution.

The YARN cluster manager that Spark supports is part of the Apache Hadoop framework and is used for resource management and job scheduling and monitoring. The centering idea of YARN is to provide a global *ResourceManager* (RM) and an *ApplicationMaster* (AM) for each application — either a single job or a directed cyclic graph (DAG) of jobs. *ApplicationMaster* is a framework specific library, and is responsible for negotiating resources from the *ResourceManager*. The *ApplicationMaster* communicates with *NodeManagers* to execute and monitor the tasks.

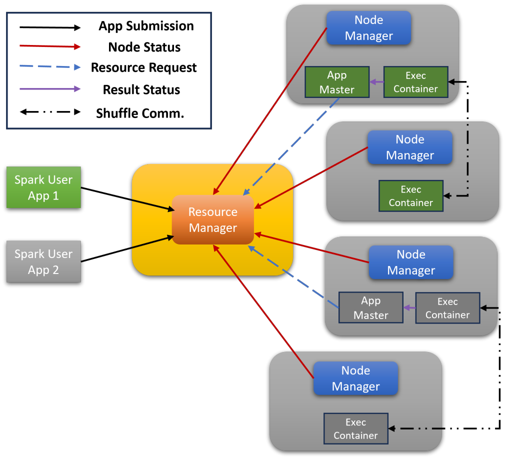

Fig. 1: Spark using YARN cluster manager.

Figure 1 illustrates how Spark relies on the YARN cluster manager. There are several components in the figure: *Spark User App*, *ResourceManager*, *NodeManager*, *ApplicationMaster* (App Master), and *Exec Container*. The *ResourceManager* and the *NodeManager* makeup the framework for data-computation. The *ResourceManager* sets up the resources for all the applications in the cluster. The *NodeManager* (can be thought of as the worker process in Spark's internal standalone cluster manager) is concerned with launching containers (either the *Exec Container* or the *ApplicationMaster*), and monitors each containers resource usage (cpu, memory, disk, network) and reports it back to the *ResourceManager*.

MPI4Spark focuses on optimizing the performance of the shuffle phase utilizing production-quality MPI libraries such as MVAPICH inside of the Netty layer. MVAPICH supports various HPC interconnects, enabling portability for MPI4Spark across multiple HPC systems that use different interconnects.

The shuffle phase is a phase during the runtime of a Spark user application where intensive communication between Spark executors takes place. This phase is considered a performance bottleneck [15] and has been the target for multiple optimization efforts — for example, RDMA-Spark and SparkUCX. The shuffle phase is triggered by RDD transformation operations that produce wide data dependencies across nodes in a Spark cluster.

Spark relies on two operations for manipulating RDDs, *action*s and *transformation*s. *Actions* perform computations on data partitions and return the respective values back to the Spark driver. *Transformations* transform existing data partitions into new data partitions. *Transformation* operations create a Directed Acyclic Graph (DAG) for processing dependencies among RDDs. Two types of dependencies among RDDs exist, narrow and wide dependencies. Narrow dependencies are created from using functions such as *Map*, with one parent RDD partition directly mapping to only one child RDD partition. This is different in wide dependencies, whereby partitions in parent RDDs have several RDD child partitions dependencies. Wide dependencies are created by functions such as *GroupByKey* or *SortByKey* and produce the communication intensive phase known as the shuffle phase.

Figure 2 provides an example of the shuffle phase between four executor containers in Spark using the YARN cluster manager, where a data partition depends on another data partition that lives outside of the node that hosts the respective executor container. This shuffling of data blocks occurs between executor container processes. Each *NodeManager*, hosting an executor container, is a separate node on the cluster.

III. Challenges

In this paper, a number of challenges were encountered in providing support for the YARN cluster manager inside of MPI4Spark. This involved redesigning the logic of launching executors used in the "standalone" cluster manager.

Challenge 1: Avoiding pitfalls of MPI4Spark's standalone cluster manager design. In MPI4Spark's standalone cluster manager design, MPI communication is used across the Spark cluster using both intra- and inter-communicators. MPI point-to-point communication is used between executors and with the driver process, along with MPI collective communication in launching executor processes dynamically using Dynamic Process Management (DPM) [16] in MPI. This is not ideal, as the whole Spark cluster needs to be launched within an MPI environment in order for MPI communication to take place. Spark's promise of fault-tolerance is no longer

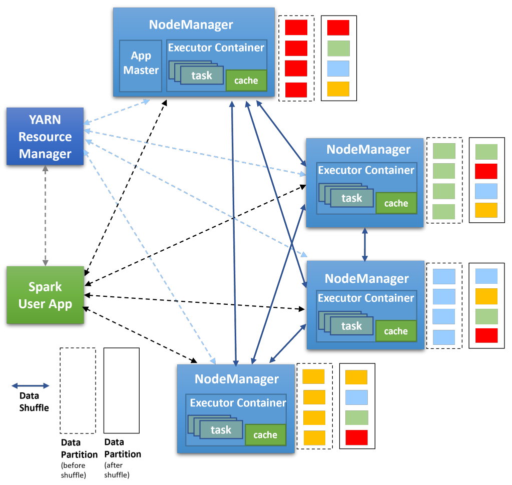

Fig. 2: Shuffle phase example in Spark using YARN cluster manager.

viable in MPI4Spark, as MPI is not capable of conducting fault-tolerance. If any failures occur within the MPI4Spark cluster, the entire cluster will be terminated as it is launched inside of an MPI environment.

With support for the YARN cluster manager, this is avoided. MPI communication is conducted strictly between executors, and there is no collective communication across the entire cluster, as is the case with the standalone design. Spark executor containers are the only components in the cluster to be launched inside an MPI environment.

Challenge 2: Launching Executors with the YARN cluster manager. With the YARN cluster manager, Spark executor processes are launched inside of containers through launcher scripts that are written during runtime. To accommodate this, we rely on the Multiple-Program-Multiple-Data (MPMD) [17] launcher mode that MPI launchers (i.e. *mpirun*) provide. A config file is written that points to the executor launcher scripts, which is then passed to the MPI launcher. The executor containers are then launched within an MPI environment where MPI communication can take place. We avoid using DPM, and so collective MPI communication is not needed here.

IV. Design And Implementation

MPI4Spark only supports Apache Spark's standalone cluster manager, which is Spark's own internal cluster manager. The standalone design utilizes MPI collective communication functions such as *MPI_Allgather()* and Dynamic Process Management (DPM) communication functions such as *MPI_Comm_spawn_multiple()* — used in launching executor processes. The MPI4Spark package also relies on a Java wrapper program that launches the Spark standalone cluster using MPI launchers (i.e. *mpirun*). MPI point-to-point communication also takes place in communicating shuffle data.

Apache Spark provides support for other cluster managers, such as Mesos , YARN, and Kubernetes. In this section, we present a new design for the MPI4Spark effort that supports the YARN cluster manager. The new design relies on the same architectural design of MPI4Spark, that is, using MPI at the Netty communication layer. It also avoids certain aspects of the older, standalone design.

To better accommodate for scalability for HPC, MPI4Spark should support the YARN cluster manager, since YARN is designed for large-scale clusters that consist of hundreds, even thousands of nodes. Support for the YARN cluster manager can also be helpful in workloads that rely on HDFS [18] or Hadoop [19], allowing for easier integration of MPI4Spark in those workloads.

In this new design, which supports YARN as the cluster manager, the Java wrapper program used in launching Spark standalone clusters with MPI is no longer needed, and DPM collective communication functions such as *MPI_Comm_spawn_multiple()* are no longer used. Other MPI collective functions used in the standalone design, such as *MPI_Allgather()* (also used in dynamically launching executors), are not needed either.

Instead, since shuffle data is only communicated between executor processes, the new design only launches executors for Spark directly using the MPI launcher through the Multiple-Program-Multiple-Data (MPMD) launcher mode. Only the executors are launched in an MPI environment, as opposed to the entire Spark cluster, as is the case in the standalone design.

MPI point-to-point communication is carried out only between executor processes, as opposed to also with the driver process, as is the case for the standalone design. Communication between the executor process and the driver was only carried out for meta data, and occurred only once throughout the runtime of a Spark application. For this reason, in the new YARN design, MPI point-to-point communication takes place only between executor processes.

This can be done as shuffle data, and the bulk of the communication, occurs between executors. MPI point-to-point communication continues to be conducted through the MPI-based Netty transport. The new YARN design can be used in future MPI4Spark work, as it allows for executors to launch in an MPI environment without the complications of DPM that are used in the standalone design. The new YARN design can also potentially deliver on Spark's promise of fault-tolerance in a straight-forward fashion (i.e., if an executor fails, all executors are automatically relaunched and the user application is executed again). We plan to implement this in the future.

Figure 3 showcases the YARN new design. RM and NM stand for *ResourceManager* and *NodeManager*, respectively. We call this new design MPI4Spark+YARN.

When YARN is used as a cluster manager for Spark, the driver program executes tasks through executor processes that are launched as containers on separate *NodeManager* processes. *NodeManager* processes can be thought of as worker processes in Spark's standalone mode. The executors are launched using launcher scripts created by the *NodeManagers*. In the new design, the launcher scripts, and therefore the ex-

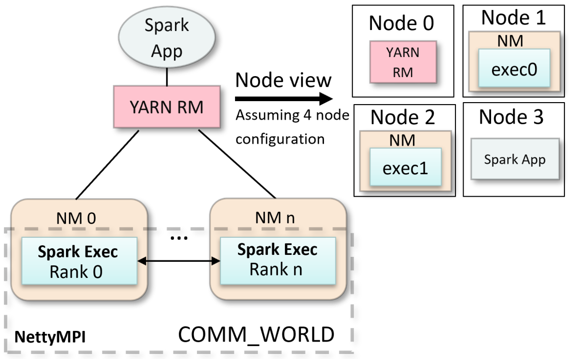

Fig. 3: MPI4spark+YARN Design. The exeuctors are only launched in an MPI environment in the new YARN design.

ecutors, are launched using an MPI launcher (i.e. *mpirun*). This is done using the Multiple-Program-Multiple-Data (MPMD) launcher mode through the MPI launcher.

The launcher scripts created by each *NodeManager* are not visible to other *NodeManagers*. Paths to these launcher scripts, therefore, need to be shared. In MPI4Spark+YARN, the Linux shared file system is used as a sharing mechanism. We modified Hadoop YARN such that each *NodeManager* creates a file *mpmd.config.*{*HOSTNAME*}. Each file is named with a suffix of the respective node manager's host name. This host name suffix is later parsed and used in creating a *hostfile*, which is needed by the MPI launcher. Inside each *mpmd.config.*{*HOSTNAME*} file created, each *NodeManager* writes the respective config information required by the MPMD launcher mode is provided, such as the path to the YARN executor container launcher script and the number of processes to launch the script with.

Since the number of processes will always be 1, because MPI4Spark only supports one executor per *NodeManager*, each *NodeManager* is only concerned with writing to the *mpmd.config.*{*HOSTNAME*} file its respective path to its executor container launcher script.

After all the *NodeManagers* have written their respective *mpmd.*
config. {*HOSTNAME*} files in a location that is visible across all the *NodeManagers* (this is done in a sub-directory labeled *mpmd-config* in the respective Spark directory), the *NodeManager* that launches the *ApplicationMaster* container is then tasked with executing the *setup-and-run-yarn-mpmd.sh* script. The *setup-and-run-yarn-mpmd.sh* script is responsible for creating the files, *mpmd.config* and *hostfile*. These files are then used in launching the executor containers on all the *NodeManager* nodes using MPI. This is illustrated in Figure 4.

Step 1 is concerned with writing the paths of the YARN executor launcher scripts to a shared location visible to all *NodeManagers*. As mentioned earlier, This is done in the *SPARK_HOME* directory inside a sub-directory called *mpmd_config*. Step 2 executes the *setup-and-run-yarn-mpmd.sh* script which creates the *mpmd.config* and *hostfile* files, and step 3 launches the executor containers in their respective *NodeManager* nodes through MPI. Launching of the executors processes on their respective *NodeManager* nodes is taken care by the MPI launcher as long as the *hostfile* and *mpmd.config* file were properly generated.

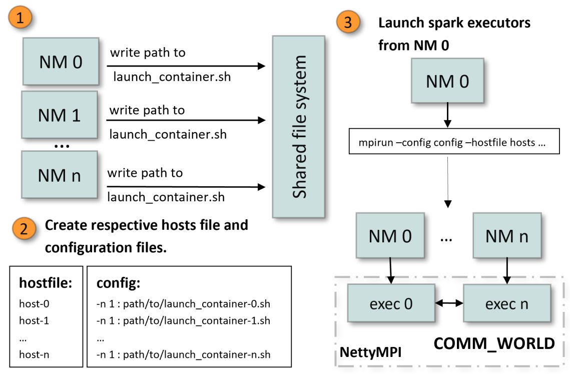
1
NM 0
write path to launch_container.sh
NM 1
write path to launch_container.sh
...
NM n
write path to launch_container.sh
Shared file system
2
Create respective hosts file and configuration files.
hostfile:
host-0
host-1
...
host-n
config:
-n 1 : path/to/launch_container-0.sh
-n 1 : path/to/launch_container-1.sh
...
-n 1 : path/to/launch_container-n.sh
3
Launch spark executors from NM 0
NM 0
mpirun –config config –hostfile hosts ...
NM 0
...
NM n
exec 0
exec n
NettyMPI
COMM_WORLD

Fig. 4: Implementation of MPI4Spark+YARN using a shared file system. Each *NodeManager* writes to the shared file system the path of the launcher script for the executor container. After which, the MPI launcher takes as input the paths of the launcher scripts and launches them inside of an MPI environment using MPMD.

V. Performance Evaluation

Performance evaluation of the new MPI4Spark YARN design was carried out on the Texas Advanced Computing Center (TACC) Frontera system. Information regarding the TACC Frontera HPC system can be found in table I. The performance of the new YARN design was compared against the performance of regular Spark and RDMA-Spark when using the YARN cluster manager.

The OSU HiBD Benchmarks (OHB) package along with the Intel HiBench benchmark suite were used in the evaluation. Both weak and strong scaling numbers were collected on different sized clusters up-to 128 *NodeManager* nodes, or in standalone terminology, up-to 128 worker nodes, a total of 7168 cores. The Intel HiBench [13] benchmark suite was also used in the evaluation.

Specification	TACC Frontera	TACC Stampede2
Cores/Socket	28	24
Sockets	2	2
Interconnect	Infiniband HDR-100	Intel Omni-Path (OPA)
RAM	192GB	192GB
Processor Family	Intel 8280	Intel Xeon Platinum 8160
Clock Speed	2.7GHz	2.1GHz

TABLE I: TACC Frontera and Stampede2 Hardware Specification.

Table II lists the memory configuration used in the evaluation. For cluster size of 32 *NodeManagers* was $70g$. Performance numbers were obtained for two benchmarks in the OHB package, *GroupByTest* and *SortByTest*. More information on these benchmarks can be found in table III.

Spark version 3.3.0-SNAPSHOT was used for both regular Spark and MPI4Spark. For RDMA-Spark, Spark version 2.1.0 was used, as that is the version that RDMA-Spark

Config Parameter	Description
SPARK_DAEMON_MEMORY	1g
SPARK_WORKER_MEMORY	90g
SPARK_EXECUTOR_MEMORY	90g
SPARK_DRIVER_MEMORY	1g

TABLE II: Benchmark input parameter configuration for 64 and 128 *NodeManagers*.

Benchmark	Description
GroupByTest	RDD-level benchmark to group the values for each key in the RDD into a single sequence
SortByTest	RDD-level benchmark to sort the RDD by key

TABLE III: OHB's different benchmark description for Apache Spark.

supports. MPI-based Netty relies on Netty version 4.1.67. Hadoop YARN version 3.3.4 was used for all Spark versions. OSU HiBD Benchmark (OHB) version 0.9.3 was used. Intel HiBench version 8.0 was used, and finally, the MVAPICH MPI library, version 2.3.7, was used in the performance evaluation.

A. *Frontera OHB Weak Scaling*

We varied the data size across different cluster sizes in a weak-scaling fashion. The evaluation was conducted on clusters with 32, 64, and 128 *NodeManagers*. Table IV lists the benchmark input parameter configuration. Both *GroupByTest* and *SortByTest* rely on the same input parameters, mappers and reducers are defined along with a key size and a number of key-value pairs. The mappers and reducers reflect the total number of cores on all the *NodeManagers*.

Cluster Size	Input Parameters	Value	Data Size
32 *NodeManagers* (1792 cores)	Mappers	1792	448GB
	Reducers	1792	
	Key-Value Pairs	65536	
	Key Value	4096	
64 *NodeManagers* (3584 cores)	Mappers	3584	896GB
	Reducers	3584	
	Key-Value Pairs	65536	
	Key Value	4096	
128 *NodeManagers* (7168 cores)	Mappers	7168	1792GB
	Reducers	7168	
	Key-Value Pairs	65536	
	Key Value	4096	

TABLE IV: Benchmark weak-scaling input configurations on Frontera for the OHB benchmarks *GroupByTest* and *SortByTest*.

The following figures provide a breakdown of the performance of Spark (IPoIB), RDMA-Spark (RDMA), and MPI4Spark (MPI) using two OHB benchmarks, *GroupByTest* and *SortByTest*. The performance is split up into multiple stages or jobs. For *GroupByTest*, there are two jobs. *Job0-ResultStage* is concerned with generating data and *Job1-ResultStage* is the stage where intensive shuffle communication takes place. The same applies to *SortByTest*, where *Job2-ResultStage* is the stage where shuffle occurs.

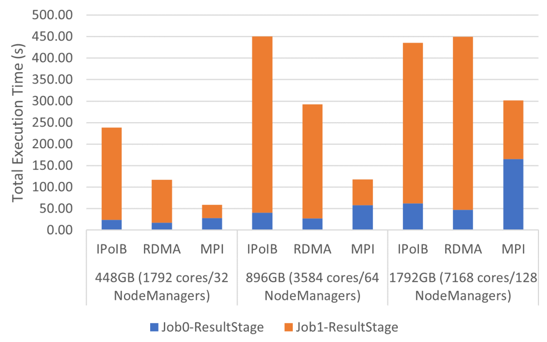

Fig. 5: *GroupByTest* weak-scaling performance numbers on Frontera. The evaluations were run on YARN cluster sizes of 32, 64, and 128 *NodeManagers*.

In Figure 5, considering the 448GB datasize (32 *NodeManagers*), MPI4Spark outperforms Spark (IPoIB) by 4.04x and RDMA-Spark (RDMA) by 2.00x. Moving to the 64 *NodeManagers* cluster size (896GB), MPI4Spark speeds up total job execution compared to Spark by 3.81x and by 2.48x compared to RDMA-Spark. For 128 *NodeManagers* (7168 cores and 1792GB), MPI4spark performs 1.44x times faster than regular Spark, and 1.48x times faster than RDMA-Spark.

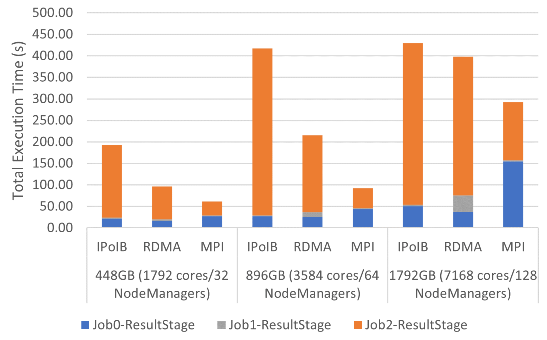

Fig. 6: *SortByTest* weak-scaling performance numbers on Frontera. The numbers were collected on YARN cluster sizes of 32, 64, and 128 *NodeManagers*.

In Figure 6, looking at 32 *NodeManagers*, MPI4Spark outperforms Spark by 3.15x and RDMA-Spark by 1.58x. For 64 *NodeManagers*, MPI4Spark is faster than Spark by 4.52x and RDMA-Spark by 2.34x. Finally, for 128 *NodeManagers*, MPI4Spark performs faster than Spark by 1.47x and RDMA-Spark by 1.36x.

Here the performance benefits that are seen are due in large to the MPI-based Netty transport that MPI4Spark utilizes. The MVAPICH MPI library is used and is capable of utilizing high-speed interconnects such as Intel Omni-Path and InfiniBand to optimize communication during the shuffle phase. We see, for 1792GB, MPI4Spark optimizes the shuffle phase by 2.94x and 2.72x compared to RDMA-Spark and Spark, respectively for the *GroupByTest* benchmark. For *SortByTest*, looking at the same data size and cluster size, MPI4Spark speeds up the shuffle performance by 2.77x and 2.38x, compared against Spark and RDMA-Spark, respectively.

B. *Frontera OHB Strong Scaling*

For this set of performance numbers, the total data size remained constant across the different cluster sizes in a strong-scaling fashion. The evaluation was conducted on cluster sizes 32, 64, and 128 *NodeManagers*. Table V lists the benchmark input parameter configuration. Both *GroupByTest* and *SortByTest* rely on the same input parameters, mappers and reducers are defined along with a key size and a number of key-value pairs. The mappers and reducers reflect the total number of cores on all the *NodeManagers*.

Cluster Size	Input Parameters	Value	Data Size
32 *NodeManagers* (1792 cores)	Mappers	1792	1344GB
	Reducers	1792	
	Key-Value Pairs	196608	
	Key Value	4096	
64 *NodeManagers* (3584 cores)	Mappers	3584	1344GB
	Reducers	3584	
	Key-Value Pairs	98304	
	Key Value	4096	
128 *NodeManagers* (7168 cores)	Mappers	7168	1344GB
	Reducers	7168	
	Key-Value Pairs	49152	
	Key Value	4096	

TABLE V: Benchmark strong-scaling input configurations on Frontera for the OHB benchmarks *GroupByTest* and *SortByTest*.

The following figures are similar to figures 5 and 6, in that they provide a performance breakdown of the performance of three different stacks, regular Spark (IPoIB), RDMA-Spark (RDMA), and MPI4Spark (MPI). The significance of the stages in the figures are explained in the previous subsection V-A.

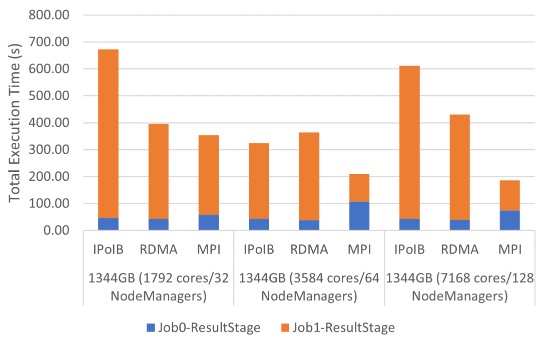

Fig. 7: *GroupByTest* strong-scaling performance numbers on Frontera. The experiments were run on YARN cluster sizes of 32, 64, and 128 *NodeManagers*.

In Figure 7, for 32 *NodeManagers*, MPI4Spark is performing better by 1.90x and 1.12x compared to Spark and RDMA-Spark, respectively. For 64 *NodeManagers*, MPI4Spark performs faster than Spark by 1.55x and RDMA-Spark by 1.74x. For 128 *NodeManagers*, MPI4Spark outperforms Spark by 3.29x and RDMA-Spark by 2.32x.

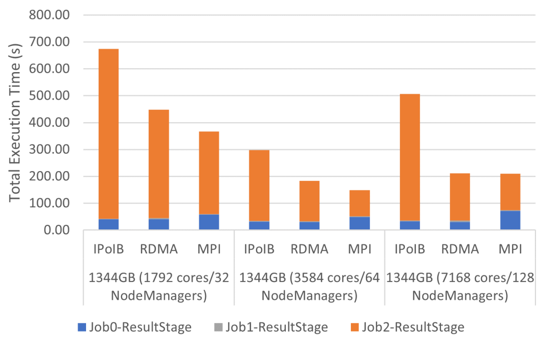

Fig. 8: *SortByTest* strong-scaling performance numbers on Frontera. The evaluations were run on YARN cluster sizes of 32, 64, and 128 *NodeManagers*.

In Figure 8, for 32 *NodeManagers*, MPI4Spark fairs better by 1.84x when compared to Spark and 1.22x when compared to RDMA-Spark. Looking at 64 *NodeManagers*, MPI4Spark is faster than Spark by 2.00x and RDMA-Spark by 1.23x. For 128 *NodeManagers*, MPI4Spark performs similarly to RDMA-Spark but is faster than regular Spark by 2.41x. During the shuffle phase, for 128 *NodeManagers*, it is worth noting that MPI4Spark outperforms RDMA-Spark by 1.28x.

The benefits that are seen in performance are due to the shuffle phase optimization that is carried out using the MPI-based Netty communication backend that relies on the MVAPICH MPI library. The shuffle phase for 128 *NodeManagers* is optimized by 3.52x and 5.11x for *GroupByTest*, respectively compared against RDMA-Spark and Spark. For *SortByTest*, also looking at 128 *NodeManagers*, the shuffle phase for MPI4Spark runs faster by 1.28x and 3.45x, compared to RDMA-Spark and Spark, respectively.

C. *Stampede2 OHB Weak Scaling*

In this section, we evaluate the performance of MPI4Spark+YARN on the TACC Stampede2 HPC system. The experiments were conducted using the OHB benchmarks, *GroupByTest* and *SortByTest*. Table VI details the configuration used for the input parameters for the benchmarks used. Different cluster sizes were used in a weak-scaling fashion using 16, 32, and 64 *NodeManagers*. The values for the the mappers and reducers parameters are set to reflect the total number of cores across all the *NodeManagers*.

Since the Stampede2 HPC system is interconnected with Intel Omni-Path (OPA), it is not possible to run RDMA-Spark. This is the reason why RDMA-Spark nubmers are not included in the following figures.

In Figure 9, MPI4spark is overall performing better than Spark. We see for the *GroupByTest* benchmark, for data

Cluster Size	Input Parameters	Value	Data Size
16 *NodeManagers* (768 cores)	Mappers	768	192GB
	Reducers	768	
	Key-Value Pairs	65536	
	Key Value	4096	
32 *NodeManagers* (1536 cores)	Mappers	1536	384GB
	Reducers	1536	
	Key-Value Pairs	65536	
	Key Value	4096	
64 *NodeManagers* (3072 cores)	Mappers	3072	768GB
	Reducers	3072	
	Key-Value Pairs	65536	
	Key Value	4096	

TABLE VI: Benchmark weak-scaling input configurations on Stampede2 for the OHB benchmarks *GroupByTest* and *SortByTest*.

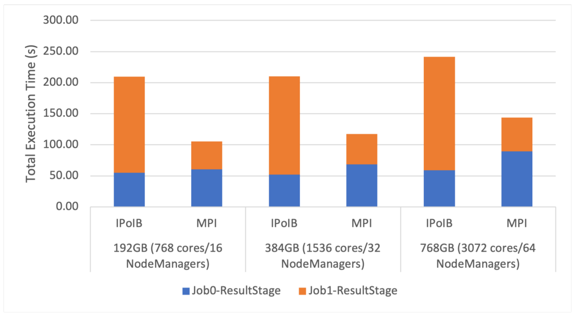

Fig. 9: *GroupByTest* weak-scaling performance numbers on Stampede2. The experiments were run on YARN cluster sizes of 16, 32, and 64 *NodeManagers*.

size 192GB, MPI4Spark performs 1.98x faster. For data size 768GB, on a cluster of size 64 *NodeManagers*, MPI4Spark outperforms Spark by 1.67x.

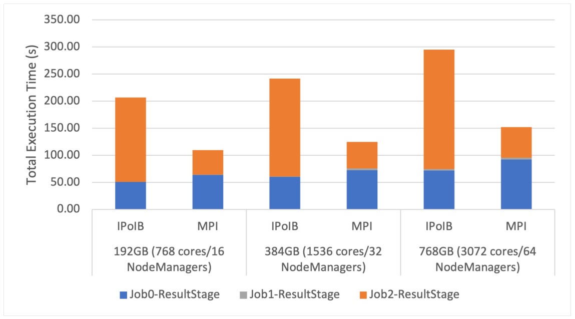

Fig. 10: *SortByTest* weak-scaling performance numbers on Stampede2. The experiments were run on YARN cluster sizes of 16, 32, and 64 *NodeManagers*.

In Figure 6, we see for the *SortByTest* benchmark, MPI4Spark is outperforming Spark. For 384GB and cluster size 32 *NodeManagers*, MPI4Spark is 1.94x faster than Spark. For cluster size 64 *NodeManagers* and data size 768GB, MPI4Spark outperforms Spark by 1.94x.

The performance benefits that we see are largely due to the optimization of the shuffle stage communication using the MVAPICH MPI library. Speed-ups for the shuffle read stage for 768GB are 3.3x compared with IPoIB (Spark) for *GroupByTest*, and for *SortByTest*, 3.9x.

D. Stampede2 OHB Strong Scaling

For strong scaling numbers, we used 3840 mappers and reducers and 65536 key value pairs with key value 4096. For both Figures 11 and 12, we see MPI4Spark performing better than Spark on average by 1.75x and 1.71x, respectively for *GroupByTest* and *SrotByTest*. RDMA-Spark does not run on systems such as Stampede2 that use Intel Omni-Path interconnects. This is why numbers for RDMA-Spark were not collected.

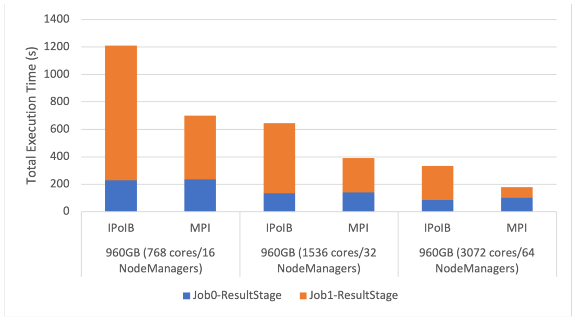

Fig. 11: *GroupByTest* strong-scaling performance numbers on Stampede2. The experiments were run on YARN cluster sizes of 16, 32, and 64 *NodeManagers*.

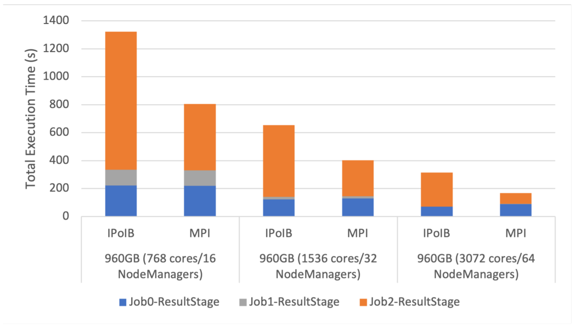

Fig. 12: *SortByTest* strong-scaling performance numbers on Stampede2. The experiments were run on YARN cluster sizes of 16, 32, and 64 *NodeManagers*.

The performance benefits seen here are in part due to the shuffle engine in MPI4Spark. The shuffle engine utilizes MPI point-to-point communication to optimize Spark shuffle performance. On average shuffle performance in optimized by 2.45x.

E. Performance Evaluation with Intel HiBench on Frontera

In this sub-section, performance evaluation of Spark and MPI4Spark were carried out using the Intel HiBench suite. A number of benchmarks were run on a Hadoop YARN cluster of size 32 *NodeManagers*. Spark's memory configuration was changed such that *SPARK_EXECUTOR_MEMORY* and *SPARK_WORKER_MEMORY* were set to 120g and *SPARK_DRIVER_MEMORY* was set to 6g.

Table VII lists other configuration parameters used in the MPI4Spark+YARN performance evaluation using the Intel HiBench benchmark suite. RDMA-Spark numbers were not collected here as the HiBench version used (v8.0) does not support RDMA-Spark's Spark version (v2.1.0), and certain benchmarks like Repartition do not exist for older versions of HiBench that support Spark version 2.1.0.

Config Parameter	Description
spark.driver.maxResultSize	5g
spark.executor.memory	120g
spark.driver.memory	6g
hibench.yarn.executor.num	32
hibench.yarn.executor.cores	56
hibench.scale.profile	huge
hibench.default.map.parallelism	1792
hibench.default.shuffle.parallelism	1792

TABLE VII: Benchmark input parameter configuration for the Intel HiBench performance evaluation on the Frontera TACC HPC system. The evaluation was carried out on a YARN cluster of 32 *NodeManagers*.

In Figure 13, several benchmarks were run using the new YARN design and regular Spark. Table VIII provides a quick description for the benchmarks used here. Each benchmark has its own set of configure parameters too. For this evaluation, the default configurations that HiBench provides were used.

Looking at the Logistic Regression (LR) benchmark, it can be noted that MPI4Spark is able to perform better than Spark by a factor of 1.91x. Moving to the Singular Value Decomposition (SVD), MPI4Spark is faster by a factor of 1.32x. For Repartition, MPI4Spark outperforms Spark by a 1.15x. Finally, for the Gradient Boosted Trees (GBT) benchmark, MPI4Spark fairs better by 1.14x than Spark.

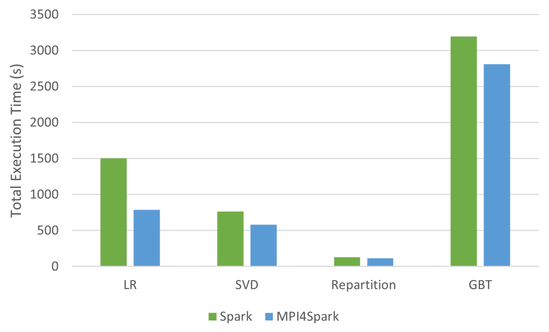

Fig. 13: Performance numbers for several Intel HiBench benchmarks on the TACC Frontera HPC system. The evaluation was conducted on a YARN cluster size of 32 *NodeManagers*.

VI. Related Work

There have been many efforts in enhancing the performance of the shuffle phase in Apache Spark, which is known as a performance bottleneck [15] due to the intensive communication that takes place between executor processes in collecting data in an all-to-all fashion.

Efforts like RDMA-Spark [7] and SparkUCX [20] enable communication using RDMA technology provided by InifiniBand interconnects. The difference between the two efforts, however, is their implementation. RDMA-Spark is implemented through a new *RDMABlockTransferService*, while SparkUCX relies on an *RDMAShuffleManager*.

MPI solutions also exist for Spark, such as MPI4Spark [3] and Spark+MPI which uses the Hadoop Distributed File System (HDFS) [18] along with Linux shared memory to offload Spark computations to an MPI environment. However, this is implemented by adding high-level API. MPI4Spark does not introduce any new high-level API for Apache Spark.

GPU solutions for Spark also exist. Spark-RAPIDS relies on the RAPIDS [21] open source libraries in accelerating the performance of Spark using GPUs. This paper discusses certain limitations of the MPI4Spark effort developed at the Ohio State University and introduces a new and leaner design that supports the YARN [9] cluster manager which paves the way for basic fault-tolerant solutions for MPI4Spark. We also present performance evaluations using OHB benchmarks on up to $7,168$-cores using a 128 *NodeManagers* cluster.

VII. Conclusion

In this paper, a new design is presented for MPI4Spark that supports both the standalone and YARN cluster managers. Support for the YARN cluster manager is needed as YARN is designed for launching and managing large-scale clusters (up-to hundreds of nodes). This better serves the intersection of Big Data and HPC that MPI4Spark attempts to achieve in terms of scalability.

Through support for the YARN cluster manager, new design decisions had to be made in regards to launching executors. Certain pitfalls were avoided that were present in the standalone design, and Dynamic Process Management (DPM) was no longer needed for launching executor processes dynamically. Executors in the YARN cluster manager were launched directly using an MPI launcher using the Multiple-Program-Multiple-Data (MPMD) launcher mode.

This allowed for a design that could potentially support a simple fault-tolerant solution in the future, whereby failed executor processes are relaunched automatically through MPI. In the new YARN design, MPI point-to-point communication is strictly between executors, as shuffling only occurs between executor processes.

Performance evaluation of the new design was conducted on the TACC Frontera System using different data and cluster sizes. Cluster sizes ranging from 32 *NodeManagers* (1792 cores) up-to 128 *NodeManagers* (7168 cores) were used. The OSU HiBD Benchmarks (OHB) package was used in the evaluation step, mainly two benchmarks: *GroupByTest* and *SortByTest*. Both benchmarks were used in the evaluation in a weak- and strong-scaling fashion. The evaluation was carried out against regular Spark and RDMA-Spark.

Benchmark	Description
Logistic Regression (LR)	Logistic Regression (LR) is a popular method to predict a categorical response.
Singular Value Decomposition (SVD)	Singular value decomposition (SVD) factorizes a matrix into three matrices.
Repartition	The workload randomly selects the post-shuffle partition for each record, performs shuffle write and read, evenly repartitioning the records.
Gradient Boosted Trees (GBT)	Gradient-boosted trees (GBT) is a popular regression method using ensembles of decision trees.

TABLE VIII: Intel HiBench benchmarks description used in the performance evaluation of the new YARN MPI4Spark design.

On the TACC Frontera HPC system, for both weak- and strong-scaling numbers, MPI4Spark was overall performing better than both Spark and RDMA-Spark. For 64 *NodeManagers* weak-scaling numbers, with *GroupByTest*, MPI4Spark was outperforming both Spark by 3.8x and RDMA-Spark by 2.5x. While for *SortByTest*, also for 64 *NodeManagers* (weak-scaling), MPI4Spark performed better by 4.5x compared to Spark and 2.3x compared to RDMA-Spark.

For strong-scaling, for 128 *NodeManagers* and *GroupByTest*, MPI4-Spark fairs better in performance by 3.3x against Spark, and 2.3x against RDMA-Spark. While for *SortByTest*, considering 64 *NodeManagers* and strong-scaling, MPI4Spark outperforms Spark by 2x, and RDMA-Spark by 1.2x. For HiBench performance evaluations, on a cluster size of 32 *NodeManagers*, MPI4Spark performed better than Spark by 1.91x for the Logistic Regression (LR) benchmark and by 1.32x for the Singular Value Decomposition (SVD) benchmark.

On the Stampede2 HPC system, looking at cluster size of 32 *NodeManagers* and data size of 384GB, MPI4Spark outperforms Spark by 1.94x for the *SortByTest* benchmark. While for the *GroupByTest* benchmark, for cluster size 64 *NodeManagers* and data size of 768GB, we see that MPI4Spark performs faster by 1.67x compared to Spark. For strong-scaling, MPI4Spark outperforms Spark by 1.73x on average using the OHB benchmarking package. The performance benefits we see are largely in part due to the MPI-based Netty transport that MPI4Spark utilizes.

MPI4Spark's new YARN design has been release and is available on the High-Performance Big Data (HiBD) page (http://hibd.cse.ohio-state.edu/downloads/). We plan to extend the MPI4Spark effort in the future by implementing fault-tolerance, allowing for a solution that better suits both HPC environments and Big Data programming models.

VIII. Acknowledgement

This research is supported in part by NSF grants #1818253, #1854828, #2007991, #2018627, #2311830, #2312927, and XRAC grant #NCR-130002.

References

[1] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache spark: A unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, p. 56–65, oct 2016. [Online]. Available: https://doi.org/10.1145/2934664

[2] The MPI Forum, "The Message Passing Interface (MPI) 4.0 Standard," urlhttps://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf, 2023, Accessed: November 26, 2023.

[3] K. A. Attar, A. Shafi, M. Abduljabbar, H. Subramoni, and D. Panda, "Spark meets mpi: Towards high-performance communication framework for spark using mpi," September 2022.

[4] Nvidia, "InfiniBand Interconnect," 2023, Accessed: November 26, 2023. [Online]. Available: https://network.nvidia.com/pdf/whitepapers/IB_Intro_WP_190.pdf

[5] Intel, "Intel Omni-Path Interconnect," 2023, Accessed: November 26, 2023. [Online]. Available: https://www.intel.com/content/www/us/en/products/network-io/high-performance-fabrics.html

[6] HPE, "HPE Slingshot Interconnect," 2023, Accessed: November 26, 2023. [Online]. Available: https://www.hpe.com/us/en/compute/hpc/slingshot-interconnect.html

[7] X. Lu, D. Shankar, S. Gugnani, and D. K. Panda, "High-performance design of apache spark with rdma and its benefits on various workloads," pp. 253–262, 2016.

[8] "The Netty Project," urlhttps://netty.io/, 2023, Accessed: November 26, 2023.

[9] Apache Software Foundation, "Apache Hadoop YARN," 2023, Accessed: November 26, 2023. [Online]. Available: https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html

[10] ——, "Apache Mesos," urlhttps://mesos.apache.org/, 2023, Accessed: November 26, 2023.

[11] Kubernetes, "open-source system for automating deployment," 2023, Accessed: November 26, 2023. [Online]. Available: https://kubernetes.io/

[12] "OSU HiBD-Benchmarks (OHB)," 2023, accessed: November 26, 2023. [Online]. Available: http://hibd.cse.ohio-state.edu/static/media/ohb/changelogs/ohb-0.9.3.txt

[13] Intel HiBench Suite, "Big Data Benchmark Suite," 2023, Accessed: November 26, 2023. [Online]. Available: https://github.com/Intel-bigdata/HiBench

[14] D. K. Panda, H. Subramoni, C.-H. Chu, and M. Bayatpour, "The MVAPICH project: Transforming research into high-performance MPI library for HPC community," *Journal of Computational Science*, vol. 52, p. 101208, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877750320305093

[15] A. Davidson, "Optimizing shuffle performance in spark," 2013. [Online]. Available: https://api.semanticscholar.org/CorpusID:16322742

[16] W. Gropp and E. Lusk, "Dynamic process management in an mpi setting," pp. 530–533, 1995.

[17] Intel, "MPMD Launch Mode," 2023, Accessed: November 26, 2023. [Online]. Available: https://www.intel.com/content/www/us/en/docs/mpi-library/developer-guide-linux/2021-6/mpmd-launch-mode.html

[18] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," pp. 1–10, 2010.

[19] Apache Software Foundation, "Apache Hadoop," url-https://hadoop.apache.org, 2023, Accessed: November 26, 2023.

[20] SparkUCX, "A high-performance, scalable and efficient ShuffleManager plugin for Apache Spark, utilizing UCX communication layer," 2023, Accessed: November 26, 2023. [Online]. Available: https://github.com/openucx/sparkucx

[21] RAPIDS, "RAPIDS suite of open source software libraries and APIs," urlhttps://github.com/rapidsai, 2022, Accessed: November 26, 2023.