Spark Meets MPI: Towards High-Performance Communication Framework for Spark using MPI

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

Mustafa Abduljabbar
Department of Computer Science Engineering
The Ohio State University
abduljabbar.1@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

***Abstract*—There are several popular Big Data processing frameworks including Apache Spark, Dask, and Ray. The Apache Spark software provides an easy-to-use high-level API in different languages including Scala, Java, and Python. Spark supports parallel and distributed execution of user workloads by supporting communication using an event-driven framework called Netty. Some efforts — including RDMA-Spark and SparkUCX — were made in the past to optimize Apache Spark on High-Performance Computing (HPC) systems equipped with high-performance interconnects like InfiniBand. In the HPC community, Message Passing Interface (MPI) libraries are widely adopted for parallelizing science and engineering applications. This paper presents MPI4Spark which uses MPI for communication in a parallel and distributed setting on HPC systems. MPI4Spark can launch the Spark ecosystem using MPI launchers to utilize MPI communication inside the Big Data framework. It also maintains isolation for application execution on worker nodes by forking new processes using Dynamic Process Management (DPM). It bridges semantic differences between the event-driven communication in Spark compared to the application-driven communication engine in MPI. MPI4Spark also provides portability and performance benefits as it is capable of utilizing popular HPC interconnects including InfiniBand, Omni-Path, Slingshot, and others. The performance of MPI4Spark is evaluated against RDMA-Spark and Vanilla Spark using OSU HiBD Benchmarks (OHB) and Intel HiBench that contain a variety of Resilient Distributed Dataset (RDD), Graph Processing, and Machine Learning workloads. This evaluation is done on three HPC systems including TACC Frontera, TACC Stampede2, and an internal cluster. MPI4Spark outperforms Vanilla Spark and RDMA-Spark by $4.23\times$ and $2.04\times$, respectively, on the TACC Frontera system using 448 processing cores (8 Spark workers) for the `GroupByTest` benchmark in OHB. The communication performance of MPI4Spark is $13.08\times$ and $5.56\times$ better than Vanilla Spark and RDMA-Spark, respectively.**

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

I. Introduction

The global Internet population and unique mobile phone users continue to grow at an accelerated rate [1]. This rise in the digital footprint of the human population is triggering the generation of large amounts of data — the global datasphere is expected [2] to reach 175 ZettaBytes by 2025. It is becoming increasingly challenging for organizations to manage and process this large amount of data, also known as Big Data.

The Apache Spark software [3], like Dask [4] and Ray [5], provides a popular Big Data processing framework targeted for commodity hardware. Spark has also seen wide deployment on High-Performance Computing (HPC) systems including the ones at Texas Advanced Computing Center (TACC), San Diego Supercomputing Center (SDSC), and other HPC centers. Apache Spark improves the performance and scalability of its predecessor, Apache Hadoop, by maintaining user data in-memory as much as possible using Resilient Distributed Datasets (RDDs). RDDs are in-memory partitions of data that are spread out across a Spark cluster. A typical Spark application processes data by applying transformations and actions to these RDDs. The software and the associated programming model are fault-tolerant and have support for wide-ranging libraries for machine learning, graph processing, streaming, and SQL-based workloads.

The performance of Apache Spark, however, does not reap the full performance benefits that can be achieved on High-Performance Computing (HPC) through communication primitives such as MPI [6]. The Message Passing Interface (MPI) is considered the *defacto* standard for writing large-scale parallel applications on HPC systems. There are several production-quality MPI-based messaging libraries including Intel MPI [7], Cray MPI [8], MVAPICH2 [9], Open MPI [10], and MPICH [11]. More recently, MPI libraries have been exploited to scale Deep Learning (DL) training on HPC systems [12].

A. Motivation

Large-scale distributed data processing is typically enabled by running Apache Spark on parallel HPC systems. While the Spark framework is capable of exploiting computing devices

2168-9253/22/$31.00 ©2022 IEEE
DOI 10.1109/CLUSTER51413.2022.00022

like CPUs and GPUs, it fails to exploit high-performance and low latency interconnects provided by these HPC systems. The overall communication performance of the *shuffle* phase — covered in more detail in Section II — becomes a significant bottleneck in distributed execution of Big Data workloads. The reason for this bottleneck is that the Vanilla versions of the Apache Spark software rely on TCP/IP protocol via Java sockets for communication between distributed processes. There have been efforts earlier to address this shortcoming by multiple projects including RDMA-Spark [13], SparkUCX [14], and Spark-RAPIDS [15]. However, none of these efforts utilized MPI communication libraries to exploit a wide range of available high-performance networks (IB, RoCE, OPA, etc.) for maximizing performance, reducing maintenance costs, and building upon decades of research exercised by the HPC community. There are efforts that use MPI libraries such as Spark+MPI [16] and Spark-MPI [17]. Nonetheless, Spark+MPI relies on extending the high-level API for Spark, which requires extra application programming efforts. Also, Spark-MPI just enables MPI workloads to run on Spark worker nodes and does not enhance the communication backend of Spark.

The primary motivation of this work is to utilize the communication functionality provided by production-quality MPI libraries in the Apache Spark framework without having to extend the high-level Spark API. To realize our vision, we design and implement MPI4Spark that aims to extend Spark's communication infrastructure using MPI.

B. Problem Statements

Table I provides an overview and comparison of the main features of MPI4Spark with previous efforts. The proposed MPI4Spark framework optimizes the communication layer, supported by Netty, of the Apache Spark framework by utilizing MPI. Netty [18] is a New I/O (NIO) client/server framework that enables the development of event-based networking applications in Java. However, there are several issues that need to be tackled for this. The communication layers of the Spark software are mainly event-driven while the communication engine of MPI, on the contrary, is application-driven. Also, since MPI follows a Single Program Multiple Data (SPMD) model of MPI, it does not clearly map to Apache Spark where distributed processes are launched manually or through a resource manager.

This paper considers the following design challenges and questions:

- Is it possible to launch execution of Spark ecosystem and associated user workloads using MPI launchers? This is necessary in order to utilize the high-performance MPI communication inside Spark.
- The Apache Spark framework provides isolation for application execution on worker (or compute nodes) by forking new processes — called executors — from the worker processes. How can this execution model be maintained using MPI?
- How can the semantic differences between the Spark software and MPI be resolved? These differences mainly include i) the event-driven communication in Spark compared to the application-driven communication engine , and ii) process naming.
- What are the various performance benefits that can be obtained through different application workloads?

Section III details the design challenges tackled in this paper.

C. Overview

There have been two approaches to enhance the communication performance of the Spark software. The first approach, taken by SparkUCX, was to design and implement a new `ShuffleManager` based on the UCX communication library [19]. The second approach, taken by RDMA-Spark, was to rely on existing shuffle managers like `SortShuffleManager` and only enhance the associated `BlockTransferService`. In MPI4Spark , we go down a level deeper, directly at the communication layer (Netty), and enhance it with MPI. As mentioned, a unique feature of the MPI4Spark framework is that it uses the Dynamic Process Management (DPM) functionality provided by the MPI standard to launch the execution of Spark workloads. This effort allowed MPI-based communication within the Apache Spark framework using two designs — MPI4Spark-Basic and MPI4Spark-Optimized — that are described in detail later in Section IV. Both of these designs enhance the Netty communication layer.

We conduct a detailed performance evaluation of MPI4Spark for up to 32 workers (1792 cores) against Vanilla Spark and RDMA-Spark using OSU HiBD-Benchmarks (OHB) [20] provided by the Ohio State University and the Intel HiBench Benchmark Suite [21] on NVIDIA/Mellanox InfiniBand (TACC Frontera) and Intel Omni-Path (TACC Stampede2) networked systems. As an example of performance gains, MPI4Spark outperforms Vanilla Spark and RDMA-Spark by $4.23\times$ and $2.04\times$, respectively, on the TACC Frontera system using 448 processing cores (8 Spark workers) for the `GroupByTest` benchmark in OHB. The communication performance of MPI4Spark is $13.08\times$ and $5.56\times$ better than Vanilla Spark and RDMA-Spark, respectively. More detailed information regarding the performance evaluation of MPI4Spark can be found in Section VII.

D. Contributions

This paper makes the following contributions:

1) The paper presents the design and implementation of MPI4Spark, which is a novel and modular solution based on Netty for using MPI-based communication inside Spark. This effort realizes the vision of "Converged Communication Stack" for Big Data, Deep Learning, and HPC. MPI4Spark allows Spark to exploit the latest developments, features, and network support provided by production-quality MPI libraries.

Features	MPI4Spark	RDMA-Spark [13]	SparkUCX [14]	Spark+MPI [16]	Spark-MPI [17]
Support for Multiple Interconnects	✓	✗	✓	✓	✓
Adheres to Spark API	✓	✓	✓	✗	✓
Studies with Existing Benchmark Suites	✓	✓	N/A	✓	N/A
Optimization Technique	MPI-Based Netty	RDMA-Based `BlockTransferService`	UCX-Based `ShuffleManager`	Offload to shared memory and use MPI	N/A

TABLE I: Comparison of MPI4Spark with earlier work.

2) MPI4Spark utilizes Dynamic Process Management (DPM) — an advanced feature in MPI libraries — for maintaining the execution model required by the Spark ecosystem where worker processes dynamically launch executor processes for running Big Data workloads.
3) MPI4Spark maintains the event-driven communication progression engine of Netty using MPI's point-to-point communication primitives, intra/intercommunicators.
4) The paper presents the micro-benchmark latency evaluation of MPI4Spark at the Netty layer against the default communication backend. MPI4Spark significantly outperforms the existing communication backend in Netty with speed-ups up to 9× for 4MB messages.
5) We compare the performance of MPI4Spark against Vanilla Spark and RDMA-Spark on two HPC systems: i) TACC Frontera with IB HDR (100 Gbps), and ii) TACC Stampede2 with Omni-Path interconnect (100 Gbps). An internal cluster equipped with InfiniBand (IB) EDR (100 Gbps) was used to evaluate MPI4Spark at the Netty layer.
6) The paper evaluates the performance of MPI4Spark against Vanilla Spark and RDMA-Spark using two popular benchmark suites including OSU HiBD Benchmark (OHB) and Intel HiBench. Detailed performance evaluation can be seen in Section VII.

The rest of the paper is organized as follows. Section II presents the background. Sections III and IV detail the challenges encountered and the design of MPI4Spark , respectively. Launching Spark with MPI is discussed in Section V. Implementation details are introduced in Section VI. The performance evaluation of MPI4Spark is detailed in Section VII. Section VIII presents related work. The paper concludes by covering conclusions and future work in Section IX.

II. Background

This section provides an overview of the Apache Spark software and its communication framework Netty.

A. Overview of Apache Spark: Originally developed by the AMPlab in UC Berkley, Apache Spark [3] is an open-source in-memory Big Data processing engine. Spark has support for running workloads that are both iterative and interactive that include streaming, graph processing, machine learning, and SQL-based workloads. A key abstraction provided by Spark is Resilient Distributed Datasets (RDDs), which are fault-tolerant in-memory distributed data partitions. Applications on Spark run as independent sets of JVM processes managed by the `SparkContext` object inside of the main program or the driver program. Spark provides abstractions like *master*, *workers*, and *executors* to manage distributed execution of user applications. The master process communicates with other processes including the driver to allocate resources across applications and launch executors on worker nodes.

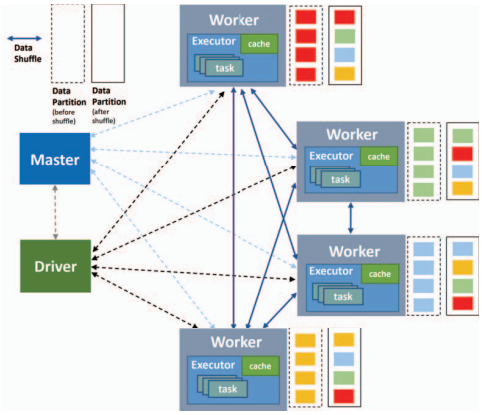


Fig. 1: A simple spark cluster consisting of four worker nodes showcasing the shuffle phase during the runtime of a given Spark application.

B. The Shuffle Phase: There are two types of operations for tasks run by executors. An *action* that carries out a computation on some data partition and returns a value back to the driver, and a *transformation* which creates a new data partition from an existing one. The transformation operation specifies the Directed Acyclic Graph (DAG) processing dependency among RDDs. Narrow dependencies are a result of functions such as `Map` and `Filter`, where each partition of the parent RDD is used by at most one child partition RDD. While, wide dependencies (i.e. Join and GroupByKey) have multiple child partitions depending on the same partition of the parent RDD. Wide dependencies involve data shuffling across the network and are a performance bottleneck for Spark applications. Figure 1 illustrates the communication patterns of the shuffle phase for a Spark cluster with four worker nodes.

Spark uses Netty to communicate RPC and shuffle messages. It does this through a set of message types that are divided into request and response message types. Table II lists the different types of messages and their functions.

C. Overview of Netty: Netty [18] is an asynchronous event-driven network application framework. It uses Java New I/O (NIO) transport by default. The NIO transport relies on a selector that utilizes the event notification API, to indicate which, among a set of non-blocking sockets, are ready for

Message Type	Function
StreamRequest	A request to stream data from the remote end
StreamResponse	A response to a StreamRequest when the stream has been successfully opened
RpcRequest	A request to perform a generic Remote Procedure Call (RPC)
RpcResponse	A response to a RpcRequest for a successful RPC
ChunkFetchRequest	A request to fetch a sequence of a single chunk of a stream
ChunkFetchSuccess	A response to ChunkFetchRequest when a chunk exists and has been successfully fetched
OneWayMessage	A RPC that does not expect a reply

TABLE II: Spark Message Types used to communicate RPC and shuffle messages

I/O. Spark relies on the NIO transport for communication of shuffle data along with RPC messages. In Spark, Netty clients and servers are created through the `TransportContext` object, with each component in the Spark cluster having its own set of Netty servers and clients.

III. Challenges and our Approach

As part of this paper, we tackle the following challenges to produce a version of the Apache Spark software that exploits MPI-based communication through the Netty framework.

Challenge 1: Launching Spark in a MPI environment. The MPI standard follows the Single Program Multiple Data (SPMD) programming model that allows executing parallel copies of the same program that communicate with one another using point-to-point and collective communication primitives. In order to utilize MPI-based communication in Spark, all processes — including master, workers, and executors — must be started as MPI processes using launchers like `mpiexec` or `mpirun`. In comparison, Spark relies on different launcher scripts (for the standalone mode) that spawn JVM processes to create the Spark environment or cluster.
Approach: In order to handle this discrepancy between how the two environments are launched, a Java wrapper program was used to launch the MPI processes using `mpiexec` on respective nodes. The MPI processes later fork Spark processes using the launcher scripts that Spark provides.

Challenge 2: Event-driven vs. Application-driven Communication Engines. The communication engine for Spark is event-driven. For instance, Spark relies on Netty that is an event-driven communication framework. In contrast, the MPI runtime is progressed through explicit calls to communication routines by the application. One of our goals in designing MPI4Spark is to enhance the Netty communication framework using MPI without affecting its event-based functionality.
Approach: To achieve this, we carefully maintained Netty's connection establishment functionality while write and read events were handled using MPI point-to-point communication primitives.

Challenge 3: Dynamically Launching Processes. To execute user workloads, the Apache Spark software forks new processes — called executors — on the same node as the worker processes. The worker processes are responsible for forking these processes. The reason for forking a new process to execute the user program is to provide isolation and security for multiple jobs running in parallel on the worker node. In addition, the worker process and the newly launched executor process also communicate with one another.
Approach: We meet the requirement of dynamically launching processes, needed by Big Data stacks, by exploiting Dynamic Process Management (DPM) and intercommunicator features of MPI.

Challenge 4: Process Naming. MPI maintains numeric identifiers called ranks to identify processes involved in a parallel execution. On the contrary, Big Data frameworks use endpoints or channels to identify processes. Endpoints/channels refer to abstractions used to identify distributed entities (Spark master or client).
Approach: The semantic mismatch between endpoints/channels and MPI ranks is handled through designs mapping ranks to endpoints/channels during connection establishment — similar to the concept of mapping virtual to physical addresses. Note that we maintain the connection establishment mechanism of Netty in MPI4Spark and use MPI-based point-to-point communication for reading and writing communication messages between various entities.

IV. The Proposed Designs

Targeting the Netty layer was motivated behind Netty's transparency, flexibility, and popularity, as it has a wide-range of adopters [18]. Modifying the Netty layer would then not only benefit Spark but also other applications. Netty also offers a more transparent code-base that is solely written in Java as opposed to Spark's which is written in both Scala and Java.

Figure 2 illustrates Apache Spark's Netty MPI-based architecture. By default, Spark uses the Netty NIO transport inside of `BlockTransferService`. Our efforts were directed at designing a new MPI transport (Netty+MPI) that uses MPI Java bindings to interface with native MPI communication libraries. The MPI transport layer relies on MPI point-to-point communication primitives as opposed to TCP/IP used by Netty's NIO transport. The design can also support different kinds of network interconnects such as IB, Intel OPA, RoCE, and HPE Slingshot.

We present two designs in this paper, MPI4Spark-Basic , and MPI4Spark-Optimized . The designs both integrate MPI at the Netty level, to keep the Spark level code unchanged as much as possible. Dynamic Process Management (DPM) is used to handle worker processes forking executors and communication between executors.

V. Launching Spark with MPI

Executors in Spark are originally launched using the `ProcessBuilder` class in Java. However, this can no longer work because Spark executors rely on Netty which, due to our design, now relies on MPI. So, instead, DPM here was used to launch the executors. This was done with the DPM collective operation `MPI_Comm_spawn_multiple()` which spawns

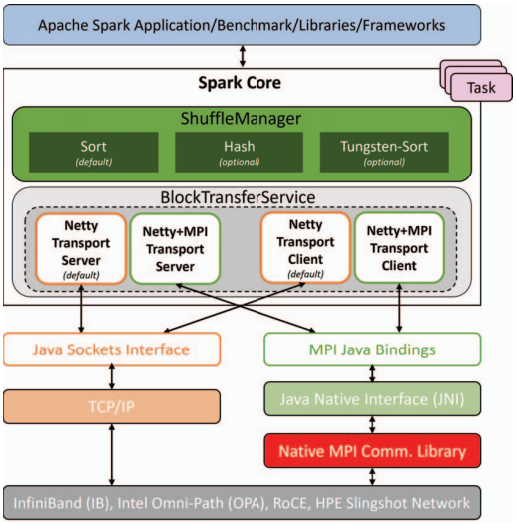


Fig. 2: An overview of the Netty MPI-based Apache Spark architecture.

multiple MPI processes with different executable specifications.

Since the DPM operation is an MPI collective operation, each worker node needs to know all the other different arguments used for launching the executors. To achieve this, an `MPI_allgather` was used across the workers to gather all the different arguments to launch the executors. The executable specifications are then sent to master and driver processes with point-to-point communication to launch the new executors collectively.

A Java wrapper program is used to launch Apache Spark with MPI. The wrapper program uses the launcher scripts that Spark provides. Figure 3 illustrates the steps required in launching Spark with MPI.

In Step A, four wrapper processes are launched with their respective ranks inside of `MPI_COMM_WORLD`. Each process forks Spark processes accordingly, as seen in Step B, where processes 0 and 1 are worker processes and 2 and 3 are master and driver processes, respectively. Step C uses DPM to launch two executor processes, creating a new `DPM_COMM` and an `Intercomm`. The significance of inter- vs. intracommunicators is that intercommunicators allow for process communication between two intracommunicators. Communication between executors is carried out using `DPM_COMM`. The node view is displayed in Step C to detail where each process sits inside of its respective node.

VI. Implementation Details for MPI4Spark

We present the implementation details of the proposed MPI4Spark design. This section is divided into several subsections that start with an overview of our Java bindings along with the flow of the design inside the shuffle phase.

A. Java Bindings for MPI

We designed and implemented our own Java bindings for MPI. The Java bindings are inspired by the MPJ Express [22] library. The Java bindings implement wrapper methods to native MPI libraries using the Java Native Interface (JNI). The goal of the Java bindings is to keep the Java layer as slim as possible for several reasons. Mainly, a minimal Java layer will aid in easier maintenance and development of the Java MPI library and helps in achieving better communication performance.

B. Identifying Processes in MPI4Spark

Spark relies on channels that identify client or server processes. For example in Netty the `ChannelId` abstraction is used to signify a unique id for a `Channel` that wraps a Java socket. MPI on the other hand relies on numeric identifiers (ranks) that are used to signify processes that are involved in parallel execution. Handling this semantic mismatch is further complicated with the use of DPM in launching executor processes which necessitates the use of inter and intracommunicators.

To handle the semantic mismatch between channels and MPI ranks and the different communicator types, each `Channel` with its corresponding `ChannelId` was mapped to both an MPI process rank and a communicator type — the communicator type mapping is needed to determine whether the MPI process needs to communicate over an intra- or intercommunicator. The mappings take place at the connection establishment phase. The ranks of MPI processes are identified and communicated through the Netty Java sockets using `PooledDirectByteBufs`. The communicator types are signified using single bytes and are also communicated during the connection establishment phase.

C. Flow of Apache Spark Netty MPI-Based Design

The flow of the MPI4Spark design is illustrated in Figure 4 through the shuffle phase. The shuffle phase is highlighted through two executors in a Spark cluster. One executor performs a reduce task that requires fetching of remote blocks. The reduce task begins by reading records from the underlying `ShuffleManager`, where combined key-values are read by the `ShuffleReader`. Within the `ShuffleReader`, the `ShuffleBlockFetcherIterator` is used to fetch data blocks either locally or remotely. Local blocks are fetched directly through the `BlockManager` without extra communication, while remote blocks (i.e. map outputs) require communication with the remote executor through Netty.

When fetches to a remote block are needed, the `ShuffleBlockFetcherIterator` will send `ChunkFetchRequest` messages to the underlying `BlockTransferService`. If the user decides to use the MPI-Based Netty design by config parameters, fetch requests will be sent through the MPI-Based Netty client to the server using MPI point-to-point communication primitives.

Once the message is received on the remote end, it will be deserialized to get the block ID information from the `BlockManager`. Upon finding the block, the `StreamManager` is used to fetch individual chunks from

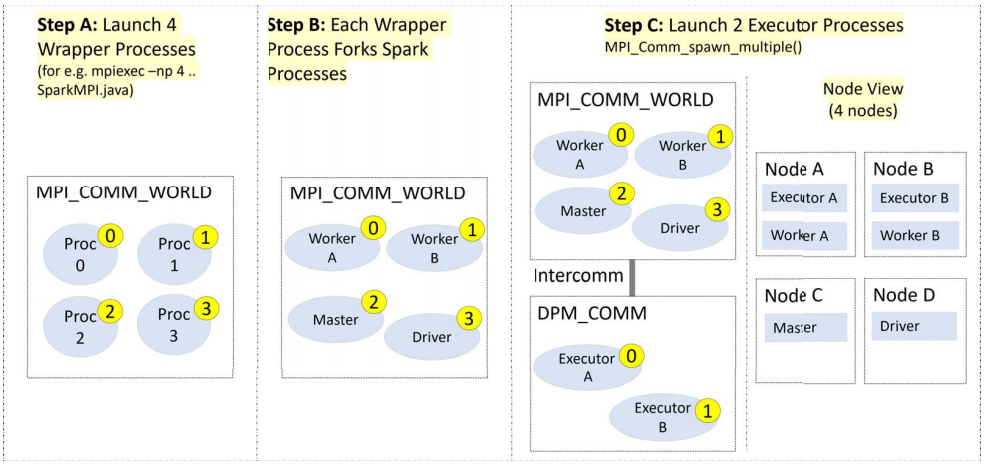


Fig. 3: An example illustrating how a MPI4Spark execution is launched. In **Step A** 4 wrapper processes are launched on separate nodes with their respective MPI ranks. In **Step B** the Spark cluster is created where each process forks Spark processes. In **Step C** the executor processes are launched using the DPM operation `MPI_Comm_spawn_multiple()`. The node view is illustrated under nodes A, B, C, and D.

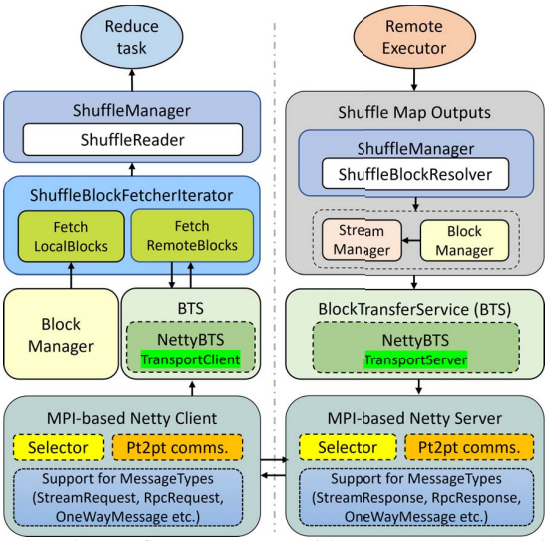


Fig. 4: Flow of MPI4Spark with Netty MPI-based design. The MPI4Spark-Basic sends all message types using MPI, the MPI4Spark-Optimized design only sends `ChunkFetchSuccess` and `StreamResposne` with MPI.

the `BlockManager`. The fetched chunks are then transferred to the underlying `BlockTransferService` where the MPI-based Netty design sends out the corresponding `ChunkFetchSuccess` message back to the client and the process repeats until all remote blocks are received.

D. The MPI4Spark-Basic Design

The MPI4Spark-Basic design modified the Netty NIO selector loop (Figure 5), which polls for channel state changes based on connection, read, or write events. Inside of the selector loop checks were implemented with MPI non-blocking probing method (`MPI_probe`) for `MPI_recv` calls matching `MPI_sends`. The blocking select operation that was used was changed to a non-blocking select to avoid hangs as there were not any messages written or read to/from the Java sockets. Netty `Channels` or simply Java sockets were still being used but only for connection establishment, besides that no messages were being communicated through the Java sockets.

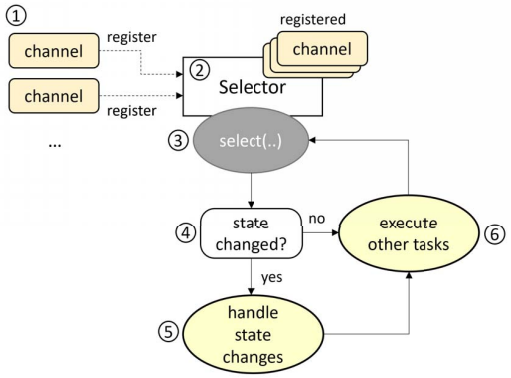


Fig. 5: Overview of how Netty polls channel updates through the selector. 1) New channels register. 2) Selector handles new registration and other channel state changes. 3) Selector.select(..) blocks until new state changes are received or if a given timeout has been reached. 4) Check if a state has changed. 5) If it has, handle it and execute other tasks. 6) Otherwise, execute other tasks and repeat process.

E. The MPI4Spark-Optimized Design

The first design used a non-blocking select and `MPI_Iprobe` in the selector loop. As a result of using these operations the performance of Spark was greatly affected. The operations were too compute-intensive and we needed to avoid using them.

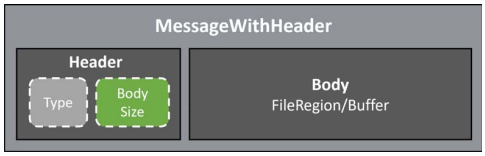


Fig. 6: Format of Spark Messages in `MessageWithHeader`, which is a wrapper class for communicating messages such as `ChunkFetchSuccess` and others. The header contains encoded information of its type and body size.

The MPI4Spark-Optimized design avoids the pitfalls of the MPI4Spark-Basic design and is a lot simpler. In this design, we only target shuffle messages. Since we can not use non-blocking MPI probes and non-blocking select calls, the idea was now to trigger `MPI_recv` calls by parsing the headers of shuffle messages inside of `ChannelHandlers`. This allows us to identify the message type (i.e. whether it is a shuffle message or not) and perform the `MPI_recv` call accordingly. Figure 7 illustrates the flow of communication between client and server `Channels` using `ChannelHandlers` which handle incoming and outgoing events inside of `ChannelPipelines`. Each message that is being received on either side of the communication parses the header inside of `ChannelHandlers` to determine whether a MPI_recv call is to be triggered for a respective MPI_send.

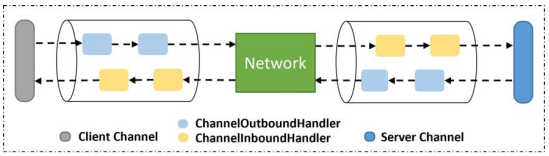


Fig. 7: Flow of Netty client-server communication, detailing the use of inbound and outbound channel handlers in channel pipelines.

Knowing that the shuffle phase was a performance bottleneck and can account for 80% of total execution time, we focused our efforts to solely optimizing it. We noticed that during the shuffle phase message types `ChunkFetchRequest` and `ChunkFetchSuccess` corresponded to shuffle messages and were communicated heavily. The `ChunkFetchRequest` message type is a request to fetch a chunk of data while the `ChunkFetchSuccess` message type is a response to `ChunkFetchRequest` when a chunk exists and has been successfully fetched.

The `ChunkFetchSuccess` message type was to be communicated through MPI as it is the message type that is involved with sending and receiving shuffle data. `ChunkFetchSuccess` consists of a header and a body (Figure 6), with the header containing encoded information about the message's type and body size. The header is sent over Java sockets, while the much larger body is sent and received with MPI.

The message type `StreamResponse` is also sent and received with MPI. `StreamResponse` is a response to `StreamRequest` and is used to communicate metadata such as jar dependencies to the worker nodes. These jars are needed for running the Spark application. Similar to how the `ChunkFetchSuccess` message type is communicated, the header message is sent over Java sockets while the body is sent through MPI.

VII. Performance Evaluation

This section presents the performance evaluation of MPI4Spark against Vanilla Spark and RDMA-Spark using two benchmark suites, the OSU HiBD Benchmarks (OHB) [20] and the Intel HiBench [21]. We could not collect numbers with SparkUCX because hangs were encountered with both benchmark suites. Table IV details the different benchmarks used in the performance evaluation. The experiments were conducted on two prominent TACC clusters, Frontera and Stampede2. An internal cluster was used for Netty-based evaluations. Hardware details for these systems are shown in Table III.

TABLE III: Hardware specification of the Xeon Broadwell system (Internal Cluster) and TACC's Frontera and Stampede2 systems.

Specification	Frontera	Stampede2	Internal Cluster
Number of Nodes	18	10	2
Processor Family	Xeon Platinum	Xeon Platinum	Xeon Broadwell
Clock Speed	2.7 GHz	2.1 GHz	2.1 GHz
Sockets	2	2	2
Cores Per socket	28	28	14
RAM	192 GB	192 GB	128 GB
Hyper-threading	✗	2 threads/core	✗
Interconnect	IB-HDR (100G)	OPA (100G)	IB-EDR (100G)

The following library versions were used for the experiments: Apache Spark v3.3.0-SNAPSHOT, Netty v4.1.67, Apache Hadoop v3.2.3 [23], OSU HiBD Benchmarks (OHB) v0.9.3, Intel HiBench v8.0-SNAPSHOT, and v7.0 — for Spark RDMA as it uses Spark v2.1.0 — and MVAPICH2-X v2.3.6 [9].

A. Evaluating Communication Performance of MPI-based Netty

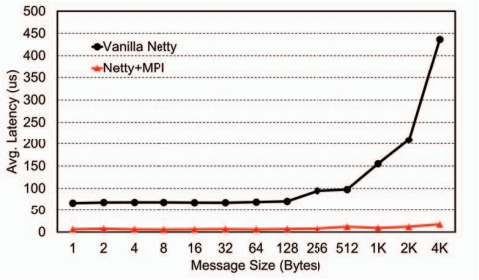
Netty is evaluated using a ping-pong benchmark that reports the latency in microseconds for different sized messages. The evaluation was carried out on two nodes on the internal cluster. Figure 8 showcases the results obtained from the benchmark. Netty+MPI performs considerably better with speedups of up to 9× times for 4MB messages.

B. MPI4Spark-Basic vs. MPI4Spark-Optimized

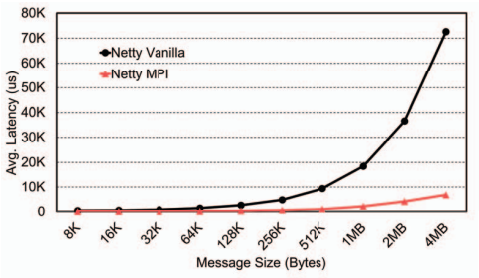
The performance of MPI4Spark-Basic was evaluated using `GroupByTest` and `SortByTest`. The evaluation was carried out for 28GB with 112 cores and 56GB with 224 cores on Fontera. Figure 9 depicts that MPI4Spark-Optimized performs better than the MPI4Spark-Basic . The reason is that the selector loop is constantly calling the non-blocking `select()` function and `MPI_Iprobe` for incoming messages. This constant polling in the selector thread was consuming CPU

Benchmark Suite	Workload	Description	Category
Intel HiBench	Support Vector Machine	Support Vector Machine (SVM) is a standard method for large-scale classification tasks	Machine Learning
	Latent Dirichlet allocation	Latent Dirichlet allocation (LDA) is a topic model which infers topics from a collection of text documents	
	Gaussian Mixture Model	Gaussian Mixture Model (GMM) represents a composite distribution whereby points are drawn from one of k Gaussian sub-distributions	
	Logistic Regression	Logistic Regression (LR) is a popular method to predict a categorical response	
	Repartition	This workload benchmarks shuffle performance	Micro Benchmarks
	TeraSort	A standard benchmark to sort input data	
	Nweight	Computes associations between two vertices that are n-hop away	Graph
OSU HiBD Benchmarks (OHB)	GroupBy	RDD-level benchmark to group the values for each key in the RDD into a single sequence	RDD Benchmarks
	SortBy	RDD-level benchmark to sort the the RDD by key	

TABLE IV: Benchmark Suite Type with Workload Description and Category



(a) Latency (Small)



(b) Latency (Large)

Fig. 8: Average latency numbers (in microseconds) collected using a ping-pong Netty benchmark on the internal cluster for large and small message sizes.

time hence starving the actual compute tasks. ***Note that for the remainder of the evaluation, the paper only utilizes MPI4Spark-Optimized design for MPI4Spark .***

C. MPI4Spark-Optimized Performance Evaluation (OHB)

The following configurations for Spark were used in carrying out the performance evaluation, spark_worker_memory=120GB, spark_daemon_memory=6GB, spark_executor_memory=120GB, spark_driver_memory=6GB. The configuration variables spark_worker_cores and spark_executor_core represent the number of threads the CPU can run concurrently, and were, by default, set to the number of cores available on a single node. For Frontera, the variables were set to 56 cores, while for Stampede2, they were set to 96, as a result of hyper-threading.

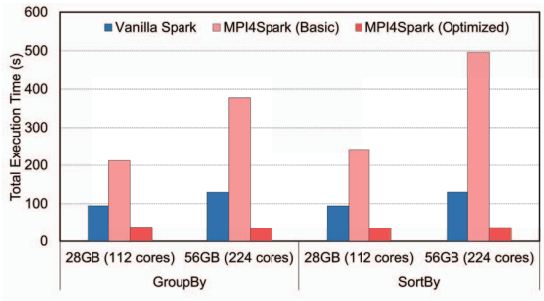
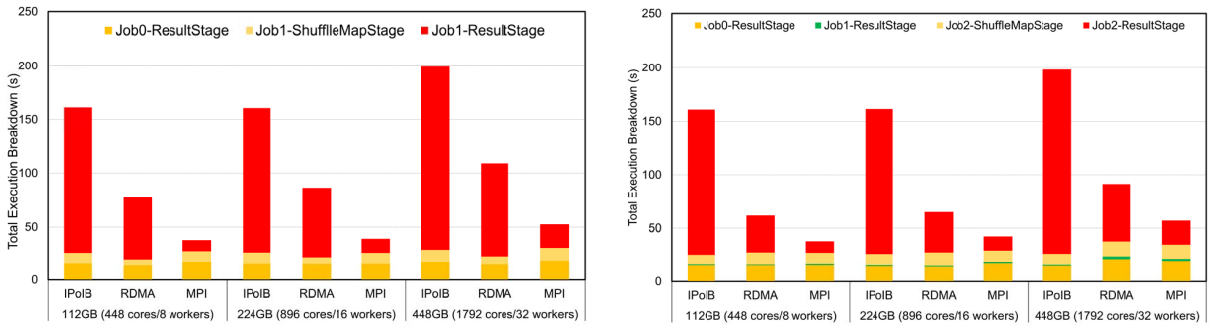


Fig. 9: Performance evaluation between MPI4Spark-Basic and MPI4Spark-Optimized using OHB's GroupByTest and SortByTest against Vanilla Spark on Frontera.

Figures 10 and 11 present the strong and weak scaling performance breakdown for OHB's `GroupByTest` and `SortByTest` RDD-benchmarks using 8 (112GB), 16 (224GB), and 32 (448GB) worker nodes. For `GroupByTest` performance is broken down into three main stages. Job0-ResultStage, Job1-ShuffleMapStage, and Job1-ResultStage. Job0-ResultStage is the stage that generates data, and Job1-ShuffleMapStage (shuffle-write) is the stage of mapping the output data into local storage (RAM disk) for the shuffle read stage. Job1-ResultStage is the shuffle read stage where the heavy communication takes place. Similarly, for `SortByTest`, Job0-ResultStage refers to the data generation stage, Job2-ShuffleMapStage refers to the shuffle write stage, and Job2-ResultStage refers to the shuffle read stage.

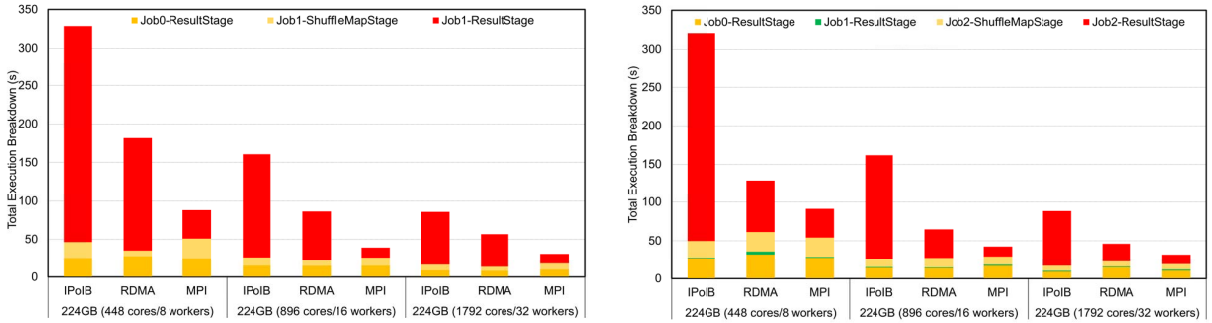
In Figure 10(a), for 448GB with 1792 cores, MPI4Spark outperforms Vanilla Spark by 3.78× and RDMA-Spark by 2.07×. For 448 cores, MPI4Spark is faster by 4.23× and 2.04× compared to Vanilla Spark and RDMA-Spark, respectively. In Figure 10(b), for the same data size and number of cores, we also perform better than Vanilla Spark by 3.44× and, for RDMA-Spark, by 1.66×. For 448 cores, MPI4Spark fairs better by 4.31× and 1.60× compared to Vanilla Spark and RDMA-Spark, respectively. For strong scaling in Figure 11(a) (GroupByTest), with 448 cores, we perform better than Vanilla Spark by 3.72× and 2.06× compared to RDMA-Spark. In SortByTest, also with 448 cores, we find that MPI4Spark



(a) GroupByTest Execution Breakdown Evaluation

(b) SortByTest Execution Breakdown Evaluation

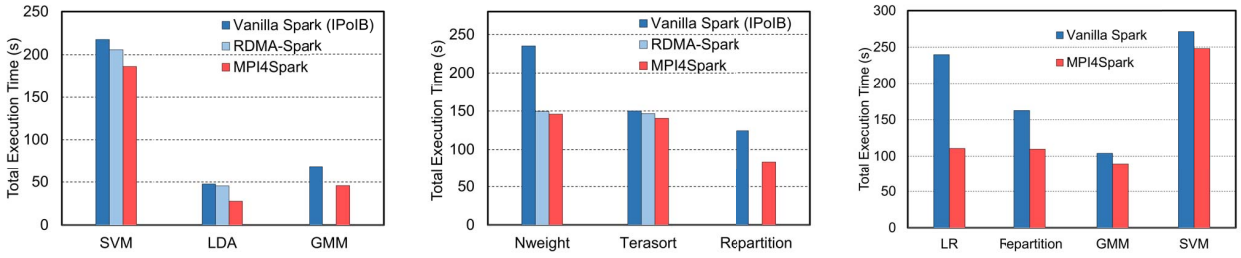
Fig. 10: Weak scaling performance breakdown for RDD Benchmarks on TACC Frontera. MPI here refers to MPI4Spark . IPoIB and RDMA refer to Vanilla Spark and RDMA-Spark, respectively. Job1-ResultStage in (a) and Job2-ResultStage in (b) refer to the shuffle read time.



(a) GroupByTest Execution Breakdown Evaluation

(b) SortByTest Execution Breakdown Evaluation

Fig. 11: Strong scaling performance Breakdown for OHB RDD Benchmarks on TACC Frontera using 224GB. MPI here refers to MPI4Spark . IPoIB and RDMA refer to Vanilla Spark and RDMA-Spark, respectively. Job1-ResultStage in (a) and Job2-ResultStage in (b) refer to the shuffle read time.



(a) ML Workloads (Frontera)

(b) Micro/Graph Workloads (Frontera)

(c) Micro/ML Workloads (Stampede2)

Fig. 12: Performance comparison of Vanilla Spark (IPoIB) and RDMA-Spark against MPI4Spark under Intel Hibench ML and graph processing workloads along with micro-benchmarks using the Huge data size and 896 cores on TACC Frontera, interconnected with InfiniBand. For Stampede2, interconnected with Intel Omni-Path, the performance evaluation was carried on 384 cores (768 threads).

performs better than Vanilla Spark and RDMA-Spark by $3.51\times$ and $1.41\times$, respectively.

D. MPI4Spark-Optimized Performance Evaluation (Intel HiBench)

Figure 12 presents the performance evaluation of MPI4Spark using the Intel Hibench suite. The evaluation was conducted on two systems, Frontera and Stampede2. On Frontera, 16 worker nodes were used in full subscription mode (896 cores). On Stampede2, 8 worker nodes were used in full subscription mode (384 cores - 768 threads). The Huge data size provided in Intel HiBench was used in all evaluations.

In Figure 12(a), for Latent Dirichlet Allocation (LDA), we can see that MPI4Spark outperforms both Vanilla Spark and RDMA-Spark by $1.74\times$ and $1.66\times$, respectively. For Singular Value Decomposition (SVM), we fair better than Vanilla Spark by $1.17\times$ and $1.10\times$ for RDMA-Spark. We could not collect numbers for RDMA-Spark for Gaussian Mixture Model (GMM) and Repartition, as HiBench 7.0 did not support it. MPI4Spark outperforms Vanilla Spark by $1.50\times$ and $1.49\times$ for GMM and Repartition, respectively. Figure 12(b) presents evaluations with the graph processing workload, Nweight, and the TeraSort micro-benchmark. For Nweight, our design performs comparably to RDMA-Spark and $1.61\times$ faster than Vanilla Spark. We see that for TeraSort we are also performing comparably to both Vanilla and RDMA. In Figure 12(c) we present numbers obtained on TACC's Stampede2 cluster. RDMA-Spark numbers were not collected here because Stampede2 does not use IB interconnects. For the Logistic Regression (LR), GMM, SVM and Repartition, MPI4Spark speeds up execution time by $2.17\times$, $1.09\times$, $1.16\times$, and $1.48\times$, respectively.

E. Discussion

The performance benefits seen in MPI4Spark are a result of optimizing the shuffle phase using MPI compared to the unified communication runtime (UCR) of RDMA-Spark. This is seen in Figure 10(a) and 10(b) where MPI4Spark outperforms both Vanilla Spark and RDMA-Spark by $13.08\times$ and $5.56\times$ and $12.78\times$ and $3.19\times$ for 448 cores, respectively. For strong scaling, similarly can be said about MPI4Spark 's shuffle communication performance optimizations.

VIII. Related Work

Several efforts were carried out to optimize Apache Spark like RDMA-Spark [13], SparkUCX [14], Spark-RAPIDS [15], and Spark+MPI [16]. SparkUCX and RDMA-Spark designs use RDMA. However, the implementation of each design is different. For SparkUCX, a new `ShuffleManager` was developed, while for Spark RDMA, the design was carried out at a lower level, where a new `BlockTransferService` was implemented. Spark-RAPIDS uses the open-source RAPIDS [24] libraries to accelerate Spark performance on GPUs. Other work such as Spark-MPI [17], provides the capability for MPI applications to run on Spark workers, while Spark+MPI, utilizes the Linux shared memory to offload Spark work to an MPI environment using HDFS [25].

One of the benefits of our design is that it adheres to the high-level API that Apache Spark provides. This is in contrast to Spark+MPI where additional API was introduced to offload RDDs to shared memory.

IX. Conclusions and Future Work

The paper presented MPI4Spark that utilizes MPI communication in the Spark framework. The paper discussed two design alternatives, MPI4Spark-Basic , and MPI4Spark-Optimized , to optimize Apache Spark for HPC at the Netty level. The performance analysis of MPI4Spark-Basic revealed overheads because of repetitive calls to `MPI_Iprobe`, coupled with non-blocking `select()` inside the selector loop — this approach led to high CPU usage for communication threads. To alleviate these performance overheads, MPI4Spark-Optimized was designed to avoid constant polling of incoming message and focused only on communicating Spark shuffle messages through MPI.

MPI4Spark outperformed both Vanilla Spark and RDMA-Spark by $3.78\times$ and $2.07\times$, respectively, for the `GroupByTest` in OHB benchmarks (weak scaling) running on 1792 cores. With strong scaling, for 224GB and 448 cores, our design was faster by $3.72\times$ and $2.06\times$. While for `SortByTest` (weak scaling), our design was faster by $3.44\times$ and $1.60\times$ compared to Vanilla Spark and RDMA-Spark, respectively for 1792 cores. In strong scaling numbers, for `SortByTest` with 224GB and 448 cores, MPI4Spark outperformed Vanilla Spark and RDMA-Spark by $3.51\times$ and $1.41\times$, respectively. For machine learning workloads like Latent Dirichlet Allocation (LDT), MPI4Spark performed better than Vanilla Spark and RDMA-Spark by $1.74\times$ and $1.66\times$, respectively for 896 cores with the Huge data size. While, for the Singular Value Decomposition (SVM) workload, we saw speed-ups of $1.17\times$ and $1.10\times$ compared to Vanilla Spark and RDMA-Spark, respectively for 896 cores. On Stampede2, we found that MPI4Spark-Optimized performed better than Vanilla Spark with speed-ups up to $2.17\times$.

The performance evaluation, on TACC's Frontera and Stampede2, showcased the portability of our design on both InfiniBand and Intel Omni-Path interconnects and the performance benefits gained through MPI4Spark . We plan to release MPI4Spark soon along with our Java MPI Bindings and incorporate fault-tolerance (using `MPI_Comm_connect` and `MPI_Comm_accept` functionality) along with support for GPU communication in the future.

X. Acknowledgement

This research is supported in part by NSF grants #1818253, #1854828, #1931537, #2007991, #2018627, and XRAC grant #NCR-130002.

REFERENCES

[1] "Data Never Sleeps 9.0," urlhttps://www.domo.com/learn/infographic/data-never-sleeps-9, 2022, Accessed: August 1, 2022.

[2] David Reinsel, John Gantz, and John Rydning, "Data Age 2025: The Digitization of the World, From Edge to Core," urlhttps://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf, 2018, Accessed: August 1, 2022.

[3] 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

[4] M. Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling," in *Proceedings of the 14th Python in Science Conference*, K. Huff and J. Bergstra, Eds., 2015, pp. 130 – 136.

[5] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A distributed framework for emerging AI applications," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 561–577. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/moritz

[6] The MPI Forum, "The Message Passing Interface (MPI) 4.0 Standard," urlhttps://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf, 2022, Accessed: August 1, 2022.

[7] Intel MPI, "Multifabric message-passing library that implements the open-source MPICH specification," urlhttps://www.intel.com/content/www/us/en/developer/tools/oneapi/mpi-library.html#gs.0s5e9z, 2022, Accessed: August 1, 2022.

[8] CrayMPI, "CUDA-aware MPI implementation," urlhttps://docs.nersc.gov/development/programming-models/mpi/cray-mpich/, 2022, Accessed: August 1, 2022.

[9] 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

[10] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.

[11] "MPICH: High-Performance Portable MPI," http://www.mpich.org. Accessed: August 1, 2022.

[12] A. Coates, B. Huval, T. Wang, D. J. Wu, A. Y. Ng, and B. Catanzaro, "Deep learning with cots hpc systems," in *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ser. ICML'13. JMLR.org, 2013, p. III–1337–III–1345.

[13] 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.

[14] SparkUCX, "A high-performance, scalable and efficient ShuffleManager plugin for Apache Spark, utilizing UCX communication layer," urlhttps://github.com/openucx/sparkucx, 2022, Accessed: August 1, 2022.

[15] "RAPIDS Accelerator For Apache Spark," urlhttps://nvidia.github.io/spark-rapids/, 2020, Accessed: August 1, 2022.

[16] M. Anderson, S. Smith, N. Sundaram, M. Capotă, Z. Zhao, S. Dulloor, N. Satish, and T. L. Willke, "Bridging the gap between hpc and big data frameworks," *Proc. VLDB Endow.*, vol. 10, no. 8, p. 901–912, apr 2017. [Online]. Available: https://doi.org/10.14778/3090163.3090168

[17] N. Malitsky, R. Castain, and M. Cowan, "Spark-mpi: Approaching the fifth paradigm of cognitive applications," *CoRR*, vol. abs/1806.01110, 2018. [Online]. Available: http://arxiv.org/abs/1806.01110

[18] "The Netty Project," urlhttps://netty.io/, 2022, Accessed: August 1, 2022.

[19] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss, Y. Shahar, S. Potluri, D. Rossetti, D. Becker, D. Poole, C. Lamb, S. Kumar, C. Stunkel, G. Bosilca, and A. Bouteiller, "Ucx: An open source framework for hpc network apis and beyond," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, 2015, pp. 40–43.

[20] "OSU HiBD-Benchmarks (OHB)," http://hibd.cse.ohio-state.edu/static/media/ohb/changelogs/ohb-0.9.3.txt. Accessed: August 1, 2022.

[21] Intel HiBench Suite, "Big Data Benchmark Suite," urlhttps://github.com/Intel-bigdata/HiBench, 2022, Accessed: August 1, 2022.

[22] A. Shafi, B. Carpenter, and M. Baker, "Nested parallelism for multi-core HPC systems using Java," *Journal of Parallel and Distributed Computing*, vol. 69, no. 6, pp. 532–545, 2009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731509000252

[23] Apache Software Foundation, "Apache Hadoop," urlhttps://hadoop.apache.org, 2022, Accessed: August 1, 2022.

[24] RAPIDS, "RAPIDS suite of open source software libraries and APIs," urlhttps://github.com/rapidsai, 2022, Accessed: August 1, 2022.

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