Boosting Cloud Computing Frameworks with High Performance Computing Features

by

Jared Ramsey

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
December 10, 2016

Keywords: HPC, MPI, Spark, InfiniBand, Distributed, Parallel

Approved by

Anthony Skjellum, Chair, Professor, Computer Science and Software Engineering
Purushotham Bangalore, Professor, Computer and Information Sciences, UAB
Alvin Lim, Professor, Computer Science and Software Engineering
Xiao Qin, Professor, Computer Science and Software Engineering

Abstract

High Performance Computing (HPC) often employs a set of networked and/or distributed computers working together to solve extremely large problems. Since the early 1990's, HPC has made use of technologies such as the Message Passing Interface (MPI) middleware and InfiniBand networking in order to provide developers with standard means to create solutions that leverage the power of a cluster of computers. However, MPI, InfiniBand, and other HPC-related technologies have gradually been losing their monopoly for applications in HPC as modern cloud computing frameworks, such as Apache Spark, are rising in popularity and are beginning to be used in fields traditionally dominated by HPC. Higher speed and Data-Center Ethernet has also challenged the choice of InfiniBand for certain scalable applications where InfiniBand was once the usual choice.

One of the primary factors advancing the popularity of these new cloud computing frameworks is that they are much more user friendly and do not immediately require an expertise in parallel and/or distributed computing to develop powerful programs. However, when every bit of performance matters, technologies such as MPI and InfiniBand still have an edge because they provide many powerful features that modern cloud computing frameworks, such as Apache Spark, do not offer at present.

I propose to augment Apache Spark with features currently only available in traditional HPC technologies, while maintaining Spark's ease of use that makes it so attractive to its current user base. There has already been some related work in this area, namely adding support for high speed network interconnects (e.g., [24]). In this thesis, I present an extension to Spark that provides easy task-parallel processing through the addition of the novel Multiply Distributed Resilient Distributed Dataset (MDRDD). The existing Spark implementation allows easy data-parallel processing via Resilient Distributed Datasets (RDDs). I also propose several future enhancements to close the gap further between the performance of HPC and the ease-of-use of Apache Spark.

The outcome of this study is that Spark is enhanced to perform user-friendly task-parallel operations in addition to the data-parallel operations it already offers. Several examples are presented to demonstrate how this functionality works and several benchmarks are run to demonstrate that the task-parallel functionality is efficient and can be used for real-world problems. The extension to Spark implemented in this study already provides enough functionality for developers to begin taking advantage of simple task-parallelism via *map* commands. Once more RDD operations, such as *reduce* are implemented in a similar fashion, it would be worthwhile to submit the changes to Apache and try to get them added to a future version of Spark.

Acknowledgments

This thesis would not have been possible without guidance and moral support from several people. I would first of all like to thank each of my committee members for their support and time spent reading over my papers and thesis. Working on a thesis is extremely challenging while working full-time, and I most likely would have quit if not for the support of Anthony Skjellum and Alvin Lim. Dr. Skjellum was kind enough to take me on as a student after some of my previous plans fell through. He was instrumental in directing my research and giving me those extra pushes I needed to motivate me to get my work done. Dr. Lim has been there for me through all of my studies and helped answer many of my concerns and questions.

I would also like to thank some of my co-workers at my place(s) of employment who provided moral support for me. I would like to specifically thank James Bassford, Robert Rutherford, Brian Green, Shirley Selvan, Sergey Yegournov, and Nester Marchenko for letting me bounce ideas off them, proctoring tests, and encouraging me to complete my studies.

Finally, I would like to thank my family, Gregory Ramsey, Deborah Ramsey, Hayden Ramsey, Kyle Ramsey, and Ariel Ramsey for providing me with support through all of my schooling. I would especially like to thank my wife, Sharon Ramsey, for being there for me through all of this and not minding me spending so much of my evenings and weekends on my schoolwork.

This material is based upon work supported by the National Science Foundation under Grants Nos. 1562659 and 1229282. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Table of Contents

List of Figures

Chapter 1

Introduction

Over the past several decades, the volume of data produced by sources such as sensors, machines, appliances, organizations, and individuals has skyrocketed. As the quantity of this data has increased, so has the need for tools and frameworks enabling robust parallel and distributed processing of this data so as to make sense of the noise and provide organizations and individuals with meaningful answers to complex questions in realistic time-frames. One of the first industries that experienced this growth was scientific endeavors such as meteorological and seismological prediction, simulation of natural phenomena, and studies of the human genome, among others. Based on earlier ad hoc message passing approaches of the 1980's, the Message Passing Interface (MPI) [1] gained traction in the 1990's because it provided scientists and engineers working in niche fields such as these the ability to develop complex parallel and distributed algorithms to make sense of their collected data (with both acceptable performance and portability). Over the years, MPI has been refined into a robust technology that is typically used in the context of High Performance Computing (HPC) supercomputers and clusters that well established laboratories and corporations provide to their users.

With the rise of the Internet, ground-breaking improvements in network capabilities, and global adoption of services such as social networking, media streaming, and online commerce, the rate and volume of data being collected is increasing at an exponential rate. Valuable applications that require and also can utilize massive distributed and parallel computation, I/O and datasets are no longer limited chiefly to scientific, niche applications. However, not all organizations and users that seek to harness such data can afford HPC clusters. Nor do they necessarily have scientists and engineers skilled in MPI, which has become notorious for being a complicated programming model that requires a specialized skill set. For instance, one recent blogger noted that “HPC is wedded to a nearly 25-year old technology stack which doesn’t meet the needs of those communities” [5]. For these reasons, cloud computing tools such as Hadoop [10] and Spark [9] (the latter powered by the Akka toolkit[8]) have emerged as alternatives to the high-powered, niche world of MPI and HPC; they strike a stark contrast by building in high-reliability, allowing these models to run on commodity hardware. They also are highly abstracted and have a much lower learning curve, allowing developers who are not specialized in parallel and distributed processing concepts to quickly develop algorithms that can execute in parallel against large data sets. For those application types they support well, they provide effective solutions, and are gaining broad adoption.

While modern cloud computing frameworks, such as Apache Spark, are steadily gaining ground in many industries, traditional HPC technologies still have their place. Specifically, while these modern tools are much more user-friendly and provide a higher level of abstraction that is easier to master, HPC technologies often can provide a higher level of performance, if mastered. While MPI has a relatively steep learning curve and requires a deeper understanding of parallel and distributed computing concepts, if used correctly it allows developers the power to utilize the resources available efficiently. Also, MPI allows for both data, and task level parallelism to be expressed. InfiniBand [6] provides clusters with high network throughput with low latency via Remote Direct Memory Access (RDMA) [43], and MPI empowers developers to leverage InfiniBand without needing to first master InfiniBand verbs. This is a major plus for MPI, as Infiniband verbs do not translate well to application-layer abstractions and developers are required to write complex middleware to be able to access the power of Infiniband verbs.

There is an emerging opportunity here to shift away from legacy HPC technologies completely toward modern, highly abstracted frameworks, such as Apache Spark, if some of the features of these legacy technologies are properly added to these newer frameworks. In this study, I extended Spark with a new Multiply Distributed Resilient Distributed Dataset

(MDRDD) that allows developers the ability to leverage task-parallel processing simply and straightforwardly on top of the existing data-parallel processing offered by Spark. I have prototyped this extension and present several examples of using this prototype and I present benchmarks showing that the prototype provides gains for specific use cases. I also propose several future additions to Spark that can further close the gap between the proven HPC technologies of yesterday with the popular cloud computing frameworks of today.

Currently, cloud computing frameworks, such as Hadoop and Spark, excel at non-communicative, data-parallel processing, especially for associative and commutative problems (under some reduction operation on the data taken as a set of operands). In other words, these frameworks are great at solving large problems that can be solved by partitioning a large set of data into many smaller sets of data, performing identical operations on these partitions of data in parallel, and piecing the results together upon completion. Hadoop solves these problems using the MapReduce paradigm, with many intermediate disk writes required, while Spark uses Resilient Distributed Datasets (RDDs) to perform this data-parallel processing mostly in memory. Frameworks such as Hadoop and Spark are excellent choices for problems that fit this paradigm [15], which is colloquially referred to as *embarrassingly parallel*. Embarrassingly parallel problems are problems that are easily divided into parallel sub-problems, communication is not required between the parallel workers, and the order of the results does not matter. Obvious examples include counting instances of words and summing various values.

While MPI can also solve embarrassingly parallel problems, it provides developers with the tools necessary to solve less-obviously parallel problems, where communication between parallel workers is required, the order of the data being processed matters, the order of results matters, or where task-parallel processing should occur. However, all of this power comes at a price. Hadoop and Spark allow developers to think in high-level terms. They can implement solutions involving logical data-structures, with little need to think in terms of how the data is structured in memory. MPI requires developers to decompose logical data

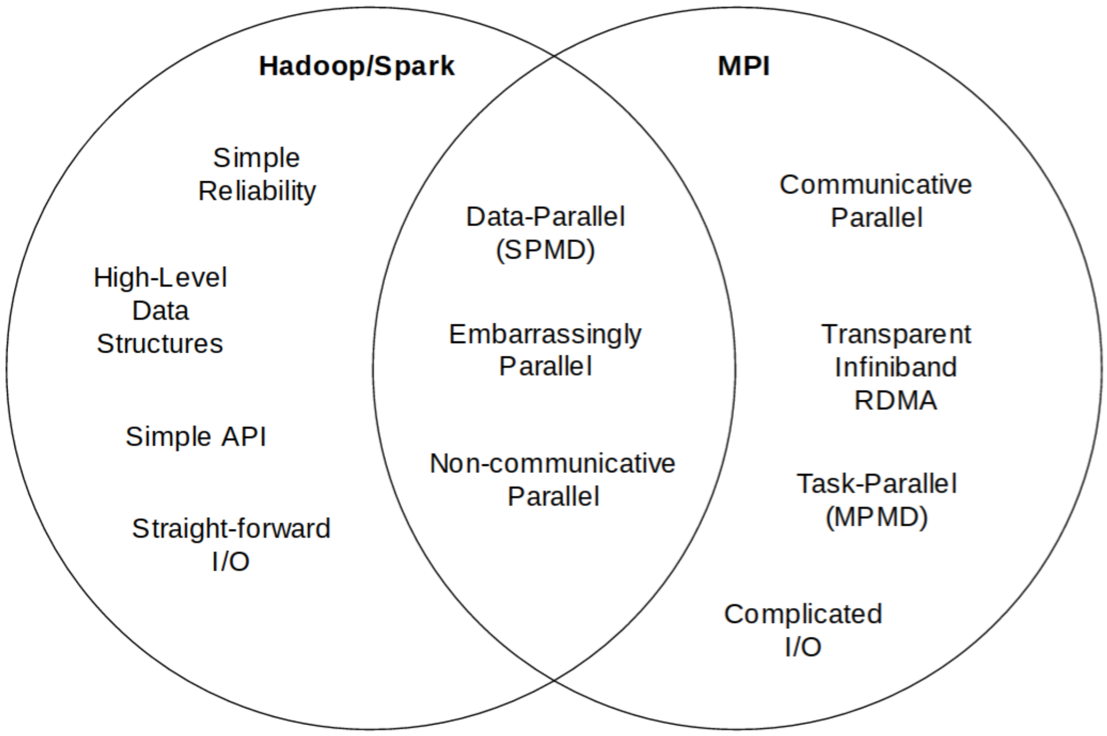

Figure 1.1: Comparison of Hadoop/Spark and MPI

structures into contiguous blocks of memory and design and code at a much lower level than comes naturally to many. See Figure 1.1 for a high-level comparison of the features and classes of problems that can be solved by MPI and Hadoop/Spark.

1.1 Thesis Organization

This thesis is organized as follows: First, an overview of HPC and HPC-related technologies is presented. Next, an overview of cloud computing frameworks is presented. After that, my proposed extension to Spark is presented and several examples using a new API (that I devised) are shown. In the subsequent chapter, I present my experiments on my modified version of Spark to evaluate my extension empirically. Then, I present and analyze the results of those experiments. Finally, I discuss related efforts and suggest further additions to this study that could be undertaken to further this blending of HPC and cloud computing technologies.

Chapter 2

High Performance Computing

High Performance Computing (HPC) is a term that refers to a large collection of computing resources working in collectively to solve some large problem or set of problems. This typically takes place in one of two architectures: a group of disjoint computers connected over a commodity network (distributed computing enabled with parallel middleware) or a group of processors sharing resources and connected by a fast, dedicated interconnect (clusters and supercomputers using similar middleware designed to utilize specialized resources) [4]. This thesis is primarily concerned with the former.

There are many challenges facing both architects and users of a distributed HPC cluster: coordination, resource bottlenecks (memory, disk, CPU, or network), reliability, properly implemented parallel algorithms, and cluster network and memory topologies, only to name a few. Many groundbreaking technologies and processes have been introduced to satisfy these challenging demands. Two of these are MPI [2] and InfiniBand [6].

2.1 MPI

Proposed in 1993 by The MPI Forum (a de facto standards body of universities, companies, and government participants), the MPI is a standard message passing interface for multi-instruction multi-data (MIMD) distributed memory concurrent computers [2]. One of the greatest challenges facing parallel and distributed computing developers is that of coordination. When there are many computers working collectively across the network, properly coordinating state and data between all of the nodes can be extremely difficult depending on the specific application. To address such concerns, The MPI Forum established MPI as a standard message passing interface to aid developers in meeting this demand in a standardized, portable way. Performance was a co-equal goal. However, this approach did not address fault tolerance at all.

MPI provides low-level building blocks that developers can rely on to design and code complex distributed, parallel algorithms. At its most basic level, it provides functions to send (`MPI_Send`) and receive (`MPI_Recv`) messages between processes. It also provides the ability to block (`MPI_Barrier`) until all processes in a collective group have caught up to a specified point. Many distributed algorithms can be developed using just these basic tools[1]. However, MPI provides a rich library of many more functions that allow developers to implement common distributed messaging strategies efficiently. In addition to basic sending, receiving, and blocking, MPI provides functions that aid with data distribution, data collection, and process grouping, among other helpful functions[3]. It also supports moving data asynchronously from the user's thread, with non-blocking send and receive operations (MPI-1), and non-blocking collective operations (MPI-3) [39].

MPI is a message passing standard for APIs (syntax and semantics, but not wire protocols) and is not a tool in and of itself. There are many implementations of the MPI standard in several different languages, and there are several versions of the standard that provide different functionality. Two of the most popular MPI implementations are MPICH and Open MPI. There are three primary version of the MPI standard: MPI-1, MPI-2, and MPI-3. MPI-1 is message-oriented and provides the ability to pass messages on a point-by-point basis. MPI-2 adds parallel I/O operations and remote memory management. MPI-3 adds more collective operations and allows for non-blocking one-sided operations [39].

To aid with data distribution and data-parallel programming MPI provides collective communication (communication over process groups). In particular, MPI provides the `MPI_Bcast` and `MPI_Scatter` functions. The `MPI_Bcast` function distributes the same data to

[1]As described below, `MPI_Allreduce` also is a great primitive for doing parallel reductions and provides a weak concept of barrier too. In minimal MPI programs, this often carries both the semantics of a barrier and the sharing of a globalized value.

all processes in a communicating group. The `MPI_Scatter` function breaks up a large block of data into segments and distributes these segments to the processes in a group. To aid with data collection, MPI provides the `MPI_Gather` and `MPI_Reduce` functions. The `MPI_Gather` function is essentially the opposite of `MPI_Scatter`; it gathers segments of data from all processes in a group and reassembles them into a gathered block of data on a master process. The `MPI_Reduce` function is special in that it performs an associative function on data as it is collected to a designated root or master process, resulting in less work needing to be done on the master process; the `MPI_Allreduce` variant symmetrically shares that result across all the participants.

To assist with process grouping, MPI provides many collective operations to organize processes into logical groups. For example, the `MPI_Cart` family of functions helps group processes into a logical Cartesian coordinate system and the MPI_Graph family of functions helps group processes into a logical graph. A key abstraction of MPI is the ability to work on different subgroups of the entire group of cooperating processes in a parallel program.

2.1.1 Pros

MPI has remained popular to this day primarily in scientific research laboratories with large HPC clusters solving large complex problems. MPI is powerful because it provides low-level functions that allow developers to implement nearly any parallel, distributed algorithm they can think of. It is typically implemented in C and/or C++, which allows for fast, minimal programs that do not waste needless space typically required by higher-level languages. Since MPI is so flexible, both task and data parallel algorithms can be implemented, which can be used to solve a wide-variety of problems. MPI has also been around for a relatively long period of time, so it has reached a high level of maturity and a wide set of users. Its feature set and API have undergone many iterations of improvements over the years, resulting in a feature-rich API that enables developers to create powerful parallel programs. In addition, MPI allows for simple access to high-bandwidth network interfaces such as RDMA over InfiniBand without requiring developers to understand the underlying InfiniBand verbs.

2.1.2 Cons

There are several downsides to using MPI. The first is that "programming at the transport layer, where every exchange of data has to be implemented with lovingly hand-crafted sends and receives or gets and puts, is an incredibly awkward fit for numerical application developers, who want to think in terms of distributed arrays, data frames, trees, or hash tables"[5]. MPI forces developers to think in such low-level terms that complex algorithms prove to be difficult to implement. Programming styles and paradigms have seen many changes over the past 20 years, and MPI has not evolved to keep up. For this reason, many new distributed and parallel stacks have come into existence to cater to users who do not want to deal with the steep learning curve of developing complex solutions using MPI.

MPI in and of itself is not terribly difficult to learn. Creating parallel solutions for toy problems can be done in a relatively short amount of time by beginners. What is difficult is developing large scale solutions for extremely large data sets, properly sharing resources among a large number of users, handling I/O in a safe manner, and handling tricky ordering of large numbers of parallel tasks. Many of the complexities of MPI are frequently hidden from developers behind middleware that provides some service to a user or application. For example, HDF5 [40] is a library written on top of MPI that allows for parallel storing and management of data. For this reason, many developers do not realize how complicated MPI is under the hood of these simplified toolkits.

2.2 InfiniBand

One of the difficulties of designing a parallel algorithm is properly handling available resources and avoiding possible bottlenecks. With work being distributed across a set of nodes in a network, network bandwidth is often in short supply and causes these programs to be bandwidth-bound [12]. In other cases, latency may be too high. For this reason, HPC solutions call for network interconnects that provide both low latency and high bandwidth. At present, low latency refers to latencies less than 10 µs and high bandwidth refers to bandwidths of the order of several Gbps [12].

InfiniBand [6] has emerged as one of the de facto standards for low latency, high bandwidth network interconnects in HPC clusters [12][7]. InfiniBand provides a rich feature set, high performance, and scalability, which makes it an attractive interconnect solution for HPC. It provides both basic send and receive operations in addition to remote direct memory access (RDMA) operations through its InfiniBand verbs layer, and has a mode called reliable transmission, typically used by applications and MPI implementations. InfiniBand achieves low latency by bypassing the traditional operating system network stack and uses its own host channel adapters (HCAs). By using InfiniBand instead of the traditional networking interconnects (such as TCP/IP over Ethernet), HPC applications can often times eliminate the network as a potential resource bottleneck. There is already an ongoing effort to bring RDMA InfiniBand support to Apache Spark [24] because of these performance advantages.

Chapter 3

Cloud Computing

Throughout this thesis, the term *cloud computing framework* is used to refer to stacks of technologies that accomplish the same, or similar, goals of HPC, but using commodity hardware and higher levels of abstraction to do so. Three popular examples of this are Apache Hadoop [10], Apache Spark [9], and Akka [8].

As mentioned previously, programming styles and paradigms have shifted several times over the past 20 years, resulting in a more object-oriented, highly abstracted programming style over previous, imperative, function-oriented programming. This allows developers to think more in terms of logical data structures and patterns they need to implement a solution, rather than the low-level details of how the operating system is managing said structures and patterns. This strikes a stark contrast to the programming style of MPI, which has not kept up with these changes and requires developers to decompose complex data structures and logical patterns into blocks of data that can be sent and received by processes. One of the major reasons MPI has not shifted paradigms and styles over the years is to remain upward compatible with new versions. With MPI always sticking to such a low level of abstraction, it is able to continuously add new features and improve various bugs over many years and many users.

In addition, standard implementations of MPI do not provide any application level reliability guarantees and expect all nodes in a cluster to maintain a certain level of health to allow continual processing. There have been many recent efforts to add reliability to MPI as a bolt-on or augmented capability, but it was not a centerpiece of its original conception, and so is difficult to support fully [13] [14]. On the other hand, modern cloud computing frameworks are designed to be run on clusters of commodity hardware, providing high levels of reliability and fault-tolerance for the inevitable case of hardware failure mid-job. For small- to medium-scale applications, this is not a major concern as jobs may only take a number of minutes or hours to rerun and synchronous checkpoint restart [?] is effective where needed. However, this is a major concern for large-scale, massive number-crunching applications that may require days or even weeks to completely process all inputs and for which checkpoint restart is prohibitively expensive. MPI's lack of built-in reliability can be a major hurdle for users that require this kind of scale. For medium-scale applications, users can implement their own means of handling reliability (primarily through checkpoints), but MPI does not currently assist users in coming up with these checkpoint mechanisms. Fortunately, third parties have done so. Even so, MPI's inability to detect failures means that fail stop behavior is not guaranteed even when faults could be detected, and programs supporting checkpoint restart can still hang and have to be restarted manually.

3.1 Apache Hadoop

In 2004, Google released the MapReduce whitepaper [15]. In this paper, the authors laid out the groundwork for many of today's modern cloud computing frameworks while borrowing from concepts that have existed in traditional HPC environments for years. At a high level, MapReduce is a processing flow where distributed Map tasks take series of key-value pairs and perform some user-defined, order-independent processing. The Map tasks' output is then shuffled and sorted by key to pass to Reduce tasks. The Reduce tasks perform more processing on this pseudo-sorted data to produce a final output. As mentioned earlier, MPI already provides Reduce functionality and MapReduce can be easily implemented in MPI. However, what makes MapReduce stand out is Apache's implementation of MapReduce in Hadoop [10].

In 2006, Hadoop was created at Yahoo!. Hadoop combines an implementation of MapReduce with a distributed file system known as the Hadoop Distributed File System (HDFS) to provide both a reliable data store with an efficient parallel and distributed processing framework that operates nicely on top of HDFS. Hadoop's MapReduce implementation leverages data locality awareness using HDFS, and HDFS provides a high level of reliability for both the intermediate and final output of MapReduce programs.

As opposed to MPI, Hadoop's implementation of MapReduce is written in Java and exposes an intuitive and high-level API. This provides developers with a shallow learning curve to learning Hadoop's ins and outs, allowing them to develop distributed, parallel algorithms in a short time without requiring knowledge of Hadoop's inner workings. In addition, Java is highly-portable, enabling Hadoop to be used on nearly every type of platform in existence today.

Over the years 2006 to 2013, industry adoption of Hadoop was quick and widespread. In 2013, more than 50% of Fortune 50 companies reported using Cloudera's popular distribution of Hadoop for some form of their business processing [18]. Since 2013, organizations have been shifting away from Hadoop for several reasons. The key reason for this shift is that Hadoop is disk-intensive and unnecessarily writes all of its intermediate output to disk. This results in a massive, unnecessary slow-down when much, or sometimes all, of this intermediate output could be kept in memory. Another reason is that MapReduce isn't as capable as MPI when it comes to certain classes of data parallel problems. Hadoop is mostly useful for embarrassingly parallel problems, where the data to be processed is not dependent on ordering, the parallel workers do not need to communicate with one another, and where the order of the results do not matter. MPI provides many low-level tools to developers which allow them to come up with relatively more complex parallel algorithms, such as communicative parallelism and task parallelism, which Hadoop has no notion of.

3.2 Akka

Inspired by the Erlang [19] programming language, which is known for its concurrent, event-driven approach, the Akka toolkit [8] was developed in 2009 with its initial version released in 2010. Akka uses an actor-based, concurrent model to provide developers the tools to develop parallel, distributed applications using Scala [20], a functional programming language.

Unlike Hadoop, Akka is a toolkit, not a framework, and provides developers with tools to develop concurrent applications, but does not impose a narrow processing flow like MapReduce. This makes Akka flexible like MPI, but at a slightly higher level of abstraction, thus allowing developers to use higher-level, logical data structures in their programs. Akka's flexibility is a step in the right direction, but it does does not provide a simple, straightforward API, like Hadoop, that newcomers to parallel and distributed computing can latch onto easily.

3.3 Apache Spark

In 2010, Spark [9] was developed at UC Berkeley, and in 2014 it became a top-level Apache Software Foundation project [21]. Apache Spark is the next logical step in cloud computing frameworks, picking up where Hadoop left off. It is a distributed processing framework built on top of Akka [8] that aims to keep as much data in memory as possible during processing. The key component of Spark that makes this possible is the Resilient Distributed Dataset (RDD). An RDD is a fault-tolerant collection of elements that can be operated on in parallel [9]. Essentially, when an existing data collection is parallelized into an RDD, it is segmented into partitions that can then be assigned to nodes (servers) throughout the cluster. A number of configurable replicas are also assigned to other nodes in the cluster, so that if a node crashes during processing another node can pick up any missing work. If the data being processed already exists in a distributed store such as HDFS, the replicas are only replicated on disk, unless there is a failure, in which case the partition is loaded into memory, at which point it is processed by the replacement node. If the data being processed only exists in memory, the data is replicated across the network to all three nodes, but only one node processes the data unless there is a failure. This doesn't prevent all possible failures,

```
val numbersRDD = sc.parallelize(numbersArray)
val mappedRDD = numbersRDD.map(x => 3*x)
```

Figure 3.1: Spark Data-parallel Example

but it does prevent many common issues, such as hardware failure on a single node in a cluster. If an RDD is created for an existing collection in the driver program, Spark must actually distribute the partitions to the nodes. If an RDD is created for an existing collection on a distributed file system (like HDFS [16] or HBase [17]), the data is not re-distributed. Instead, Spark intelligently assigns partitions to nodes that are local to existing partitions of data throughout the cluster. Spark provides many data-parallel operations that can be performed on RDDs. Users can define arbitrary functions to run in parallel, and Spark takes care of scheduling this work across the cluster on partitions of an RDD. The example in Figure 3.1 demonstrates how easy it is to multiply all values in an array by 3 in parallel across an RDD using Spark.

In addition, Spark provides many useful libraries for data-parallel tasks such as machine learning, streaming, and graphing [22]. Moreover, it can perform these data-parallel computations much faster than its predecessor Hadoop for several reasons. For one, Hadoop must write out all of its intermediate and final results to disk. This is obviously inefficient, especially when results are small enough that they could fit into memory. Spark on the other hand keeps all data in memory unless it absolutely must spill to disk. A second reason Spark is faster than Hadoop is that it is able to cache data in memory so later jobs can access it quickly. With Hadoop, if a later job wishes to access data that was available during a previous job, it must reload this data from disk. Finally, a third reason is that all Spark processes (JVMs) are constantly running the cluster's nodes, so when a job is submitted it is just a matter of issuing some RPC calls to get the job started. With Hadoop, each job and task requires the overhead of starting up new JVMs to host the parallel processes. For this reason, Spark is able to start jobs nearly immediately with little overhead [23] [41].

Chapter 4
Spark Extension

First we discuss the key outcome of this work, a Spark extension called the *Multiply Distributed Resilient Distributed Dataset.* Then I explain the design and implementation work that I undertook to validate the concept and demonstrate its utility.

4.1 Multiply Distributed Resilient Distributed Dataset

Spark currently provides Single Program Multiple Data (SPMD) parallelism. What this means is that if a single program (a set of instructions to run) and a large collection of data are provided, Spark will automatically break down the large dataset into many relatively smaller partitions and run the provided single program on each of the different partitions of data in parallel. This is the only type of parallelism natively provided by Spark. This means that users are not able to provide a set of programs to run in parallel, on a piece of data, which would be Multiple Program Multiple Data (MPMD) parallelism. There are certain classes of problems that, while possible to solve using existing Spark processes, can only be solved in an inefficient manner. I present an example of one of these classes of problems in a later section and show how inefficient existing Spark processes are at solving them.

I have created a novel extension to Apache Spark that allows users to achieve MPMD parallelism using an API that closely matches how they already are achieving SPMD parallelism. Figure 4.1 demonstrates a simple example of using this API. The user simply needs to create an array of functions and pass this array to the map function on the MDRDD. The following is a brief explanation of each line of the example presented in Figure 4.1:

- An MDRDD is created from the existing numbersArray array. The 2 used when constructing the MDRDD specifies that 2 functions will be executed in parallel on this MDRDD.
- A function is defined to multiply each element in the MDRDD by 3.
- A function is defined to multiply each element in the MDRDD by 4.
- An array is created with all of the functions to execute.
- The function array is provided to the map function of the MDRDD and Spark takes care of executing the provided tasks in parallel.

What follow is an explanation of how Spark handles this new MDRDD extension. When an MDRDD is created, two parameters are provided: the collection of data to parallelize and the order of parallelism that must be supported (the number of functions to execute in parallel). For example, if two functions will be executed on the data in parallel, the order of parallelism is two. When when Spark distributes the work to the slave nodes, it does so according to the diagram in Figure 4.2. In this example, there are six slave nodes and two functions to execute in parallel. When Spark divides the work to the slave nodes, it divides the MDRDD into three partitions, and gives each set of three partitions a function to execute. In other words, there are n copies of each partition of the MDRDD where n represents the order of parallelism that is to be achieved (the number of functions to execute in parallel). Not illustrated in the diagram is the fact that each partition is still replicated to other nodes in case one of the nodes crashes and another node needs to pick up the dropped work.

Figure 4.3 is an example of multiplying a collection of numbers by two, three, four, and five using traditional Spark. In this example, the RDD containing the numbers is multiplied by each of these numbers one after the other. Figure 4.4 is an example of multiplying the same collection by the same numbers, only this time the multiplications are done in parallel using my extension of MDRDDs to Spark.

```
val numbersMDRDD = sc.parallelizeMDRDD(numbersArray, 2)
val f1: (Int) => Int = 3 * _
val f2: (Int) => Int = 4 * _
val functions = Array(f1, f2)
val mappedMDRDD = numbersMDRDD.map(functions)
```

Figure 4.1: Spark MPMD Example

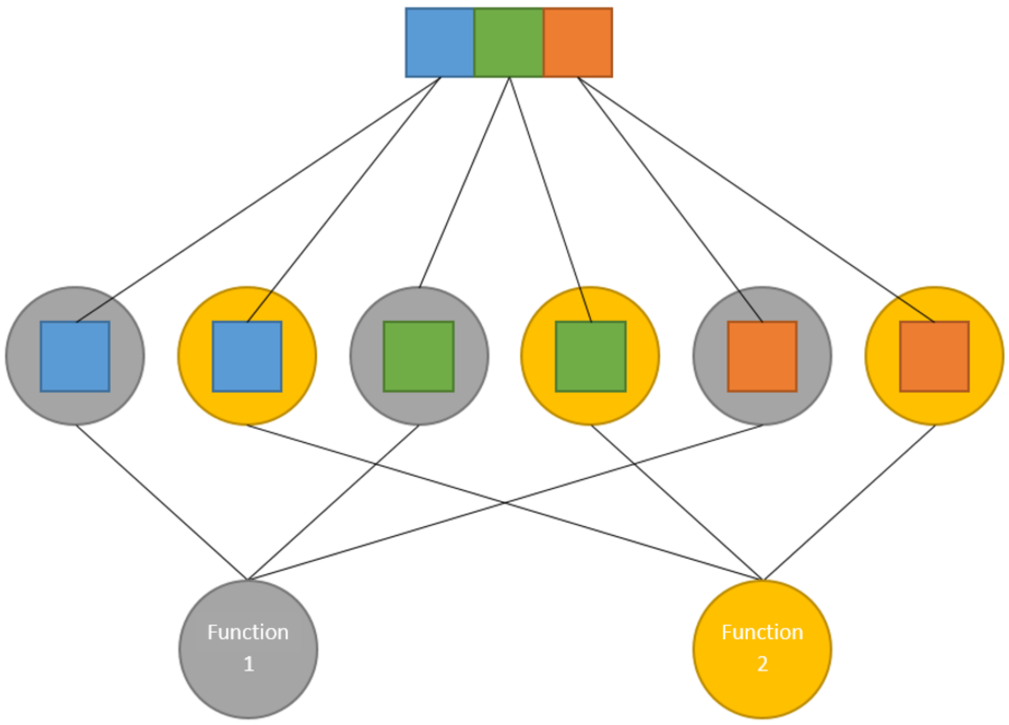

Figure 4.2: MDRDD Concept

```
val listRdd = sc.textFile("hdfs://spark/numbers.txt")
val mapped1 = listRdd.map(x => x * 2)
val mapped2 = listRdd.map(x => x * 3)
val mapped3 = listRdd.map(x => x * 4)
val mapped4 = listRdd.map(x => x * 5)
mapped1.take(size)
mapped2.take(size)
mapped3.take(size)
mapped4.take(size)
```

Figure 4.3: Example Serial (SPMD) Program

```
val listMdrdd = sc.textFileMDRDD("hdfs://randomNumbers.txt")
val f1: (String) => Int = _ * 2
val f2: (String) => Int = _ * 3
val f3: (String) => Int = _ * 4
val f4: (String) => Int = _ * 5
val functions = Array(f1, f2)
val mapped = listMdrdd.map(functions)
mapped.takeParallel(size)
```

Figure 4.4: Example Parallel (MPMD) Program

4.2 Design and Implementation Choices

This section details the design and implementation choices I made to add task-parallelism support to Apache Spark via Multiply Distributed Resilient Distributed Datasets, To begin with, I cloned the latest, stable version of Apache Spark from GitHub. When I began this project, that version was 1.5.2.

After cloning 1.5.2, I explored the source code to determine the best method of adding task-parallelism to Spark. Spark is primarily written in Scala, although it makes use of several Java utilities, and uses Akka and Netty to handle various communication and messaging concerns. I was presented with two obvious choices at this juncture: I could add task-parallel functionality completely from scratch, essentially re-engineering major building blocks of Spark, or I could identify if there were a way to wrap already implemented pieces of Spark in a new way and expose this to users. After investigating how difficult the first option would be and how feasible it would be to accomplish this addition in a reasonable time-frame, I decided instead to move forward with the second option. As can be seen in the next section, this option not only worked out in terms of implementation effort, it also performs well at least for relatively average loads and so far has proven to be a good solution. Thus, I see no reason this solution cannot scale to a much larger space, but it would be a worthwhile effort for future students with access to a much larger cluster to try this.

After determining that I would go about wrapping existing Spark classes with new classes of my own, I began looking at the best strategy to accomplish this. Two of the most critical classes to Spark's infrastructure are the SparkContext and RDD classes. When a Spark application or shell is instantiated, a SparkContext object is provided to the user. This object provides all necessary Spark API calls and is provided as the *sc* object. For example, to parallelize a collection that already exists in memory, the user would issue a *sc.parallelize(collection)* call. This call returns an RDD object that the user can operate on. There are two types of operations that can be performed on RDDs: transformations and actions. A transformation is an operation that results in another RDD being created. An action is an operation that returns a concrete result from an RDD. Transformations are lazy and are not actually executed until an action requires a transformation to take place. For example, a call such as *rdd.map(x =¿ 3*x)* is a transformation and results in a new RDD being created that represents every element in the first RDD being multiplied by 3. On the other hand, a call such as *rdd.take(3)* is an action that requires all transformations to take place and returns the first 3 results in the final array of results.

I decided to implement an MDRDD as a wrapped collection of RDDs. I added a *parallelizeMDRDD* method to the SparkContext class. While the *parallelize* method returns an RDD on which to perform data-parallel operations, the *parallelizeMDRDD* method returns an MDRDD upon which to perform task-parallel operations. The *parallelizeMDRDD* method takes two required parameters and one optional parameter. The first parameter is the collection that is to be parallelized, the second parameter is the number of parallel tasks that will be performed on this MDRDD, the third (optional) parameter is the number of partitions to split the MDRDD into. The third parameter is for optimization purposes and is application-specific. Users can explore with this parameter to find the optimal number of partitions for their datasets. When the *parallelizeMDRDD* method is invoked on a collection, the following occurs:

- The number of partitions per RDD is calculated. This is the number of partitions that were specified (or the default) divided by the number of tasks that are to be performed in parallel.

- An array of RDDs of size number of tasks is created, where each element of the array is a copy of the same RDD, with the number of partitions for each RDD determined from the previous step.

- An MDRDD object is constructed using the array of RDDs created from the previous step and is returned to the caller.

I implemented the MDRDD class as follows. An MDRDD is essentially a wrapper around an array of identical RDDs. As mentioned earlier, the array of RDDs and the partition size of the RDDs are determined primarily from the nubmer of tasks that are going to be executed in parallel. For basic testing purposes, the only functionality I implemented on MDRDDs are *map* and *take*. As opposed to the RDD version of *map*, the MDRDD *map* method takes an array of functions. Each of these functions is applied to an RDD wrapped by the MDRDD. When an action is performed on the MDRDD, such as *take*, the action is applied to each RDD in parallel using Scala Futures. A Scala Future is an object that runs a provided function asynchronously. The creator of the Future can choose to wait (block) for the Future to complete, or continue processing and be notified via an event when the Future has completed processing. I made a choice while creating the MDRDD just to block until all parallel RDD actions are completed. This seemed the logical choice because Spark actions on an RDD block already, so it makes sense that parallel actions on an MDRDD would also block until completed. I feel that blocking until all parallel processes is right choice for the future of this extension and matches choices made throughout the rest of the Spark framework. When an action, like *take*, is performed on an MDRDD, the same action is performed on all wrapped RDDs. If there are n RDDs wrapped by an MDRDD, n Scala Futures will be spun up that perform the action via the SparkContext object. The MDRDD will block until all RDD actions have completed, and then return the results to the caller. The results are returned in an array to the client, where each index of the results corresponds to the index of the function that was mapped in the previous step.

MDRDD's take advantage of the already existing Spark scheduler to run the actions in parallel. I began exploring creating a custom scheduler, designed from the ground up for parallel operations on MDRDDs, but after experimenting with the existing Spark scheduler, I determined that the scheduler can already perform parallel operations adequately. When each Scala Future is created with a Spark action, each Future asynchronously submits its action to the Spark scheduler via the Spark Context. The reason the scheduler is able to perform the actions in parallel is the partitioning choice made when creating the MDRDD. Recall that when an MDRDD is created, the number of functions that will be performed in parallel is provided as a parameter. The reason this is necessary is to divide the number of partitions that the RDD would normally be divided into by the number of functions that will run in parallel. This ensures that all of the underlying RDDs can be scheduled at the same time on the cluster because all n functions can fit on the cluster at the same time. Following is a concrete example that will make this even more clear:

- An MDRDD is created on an existing dataset. Four functions will be run on this MDRDD in parallel, so the value 4 is passed in the constructor.
- Inside of the MDRDD, four RDDs are created on the existing dataset. If the cluster layout would normally default to dividing this RDD into 20 partitions, instead each RDD is now divided into 5 partitions.
- A *map* transformation is run on the MDRDD. This map transformation consists of four functions that are each run on the RDDs contained within the MDRDD.
- A *take* action is run on the MDRDD. A scala Future is created for each of the underlying RDDs, where each Future issues a take command to the Spark scheduler via the SparkContext.
- The Spark scheduler is able to execute all of these actions in parallel because of the way the RDDs have been partitioned.

- The MDRDD blocks until all RDD actions have returned.
- An array of results is returned to the caller, containing each RDDs results in the same order as the functions of the map operations.

I believe that I made the correct design and implementation decisions while adding MDRDDs to Spark to support user-friendly task-parallel processing. In the next section, I show empirical evidence that my choices were correct and that the performance of MDRDDs exceed that of RDDs when processing tasks in parallel on a large collection of data.

Chapter 5
Experimental Results

In order to measure the performance of the new MDRDD extension to Spark empirically, I created several benchmarks designed to measure two straightforward use cases: processing data distributed from the driver/master process and processing data located on HDFS. In addition, these same benchmarks were compared to traditional Spark with serially run tasks. If the parallel tasks run in a smaller amount of time than the tasks take to run serially in traditional Spark with RDDs, then my extension can be considered a success. This section presents the results of my experiments to demonstrate that my extension to Spark is successful in achieving this end.

5.1 Cluster Details

All experiments were run on a cluster comprised of 6 nodes with CentOS version 7 installed. Each of the nodes have 40 Intel Xeon E5-2650 V3 2.30GHz CPUs and 132 GB of RAM. Each node has 240 GB of disk space available for storage. My modified version of Spark (based on version 1.5.2) was installed on each of the nodes. One of the nodes was chosen as a master node and the rest ran as workers. The Spark processes were each allocated 32 GB of heap space for storing RDD information before off-loading to disk.

5.2 Benchmarks

The results shown in Figure 5.1 are from the driver-distributed benchmark run on traditional Spark and Spark with my MDRDD extension. In this benchmark, a collection of one million random integers is created and an RDD or MDRDD is created from this set. In each run of the benchmark, a number of tasks are run on each element of the set of random

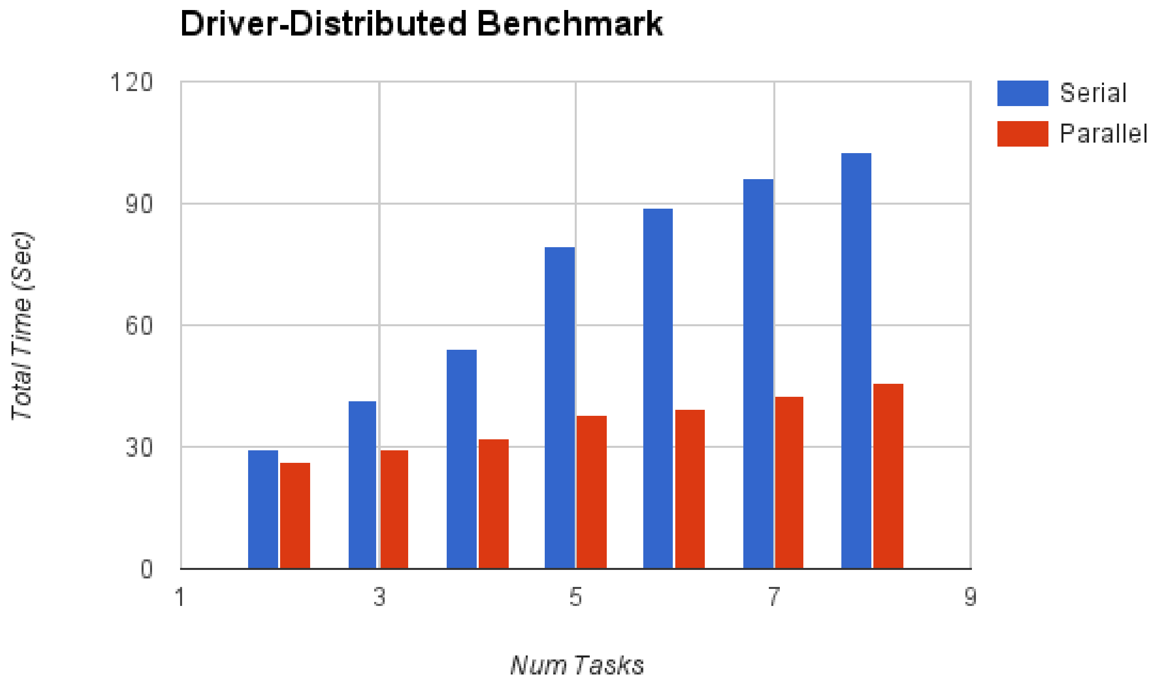

Figure 5.1: Driver-Distributed Benchmark Results

integers. Specifically, each element is multiplied by another integer. This benchmark is repeated fifty times, and the total time to run all fifty instances of the benchmark is recorded. Finally, this entire process is run three times, and the average of all three runs is recorded. This final average is what is displayed in Figure 5.1.

With reference to Figure 5.1, the performance gain by using MDRDDs is evident. As the number of tasks increase, the total time taken by traditional Spark, utilizing RDDs, appears to increase at a steep linear rate. Meanwhile, Spark with my MDRDD extension increases at a much lower rate. Traditional Spark with RDDs took 102 seconds to run all eight tasks serially 50 times and Spark with MDRDDs took 46 seconds to run all eight tasks in parallel 50 times. This is an improvement of 54%. To be clear, this is not an increase in overall Spark performance. This is an improvement for specific cases where multiple tasks need to be run on the same set of data at the same time. Another thing that should be noted is that this is a relatively small example where the work being performed is trivial

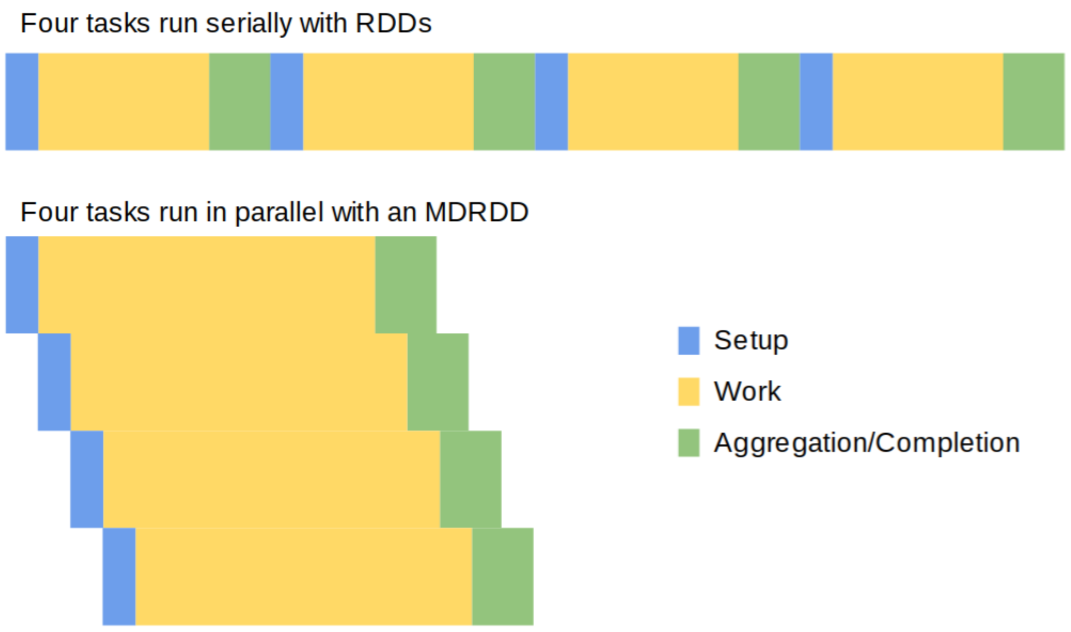
Four tasks run serially with RDDs
Four tasks run in parallel with an MDRDD
Setup
Work
Aggregation/Completion

Figure 5.2: Parallel Work Timeline

(multiplying by single numbers) and that if the work being done was larger, the performance gains would be even more pronounced.

The reason for this improvement is that the parallel work being performed for each function need not wait for the master to piece together the results of the previous operation before getting started. Figure 5.2 illustrates why parallel computing with MDRDDs is faster than serial work with traditional RDDs. While the actual work being performed for each task actually takes slightly longer because the partitions need to be larger for MDRDDs, the throughput of jobs over time is significantly higher because the work is done in parallel and the setup for a job does not need to wait for the completion of the previous task.

The results shown in Figure 5.3 are from the benchmark run on data located in HDFS, a configuration in which many Spark applications are designed to be run. This benchmark is essentially the same as the other one, but all fifty sets of one million random integers were pre-generated and placed on HDFS, which was co-located with the Spark worker nodes.

The results in Figure 5.3 closely match the results of the driver-distributed benchmark that was just discussed. Again, the total time taken to run the benchmark increases at a

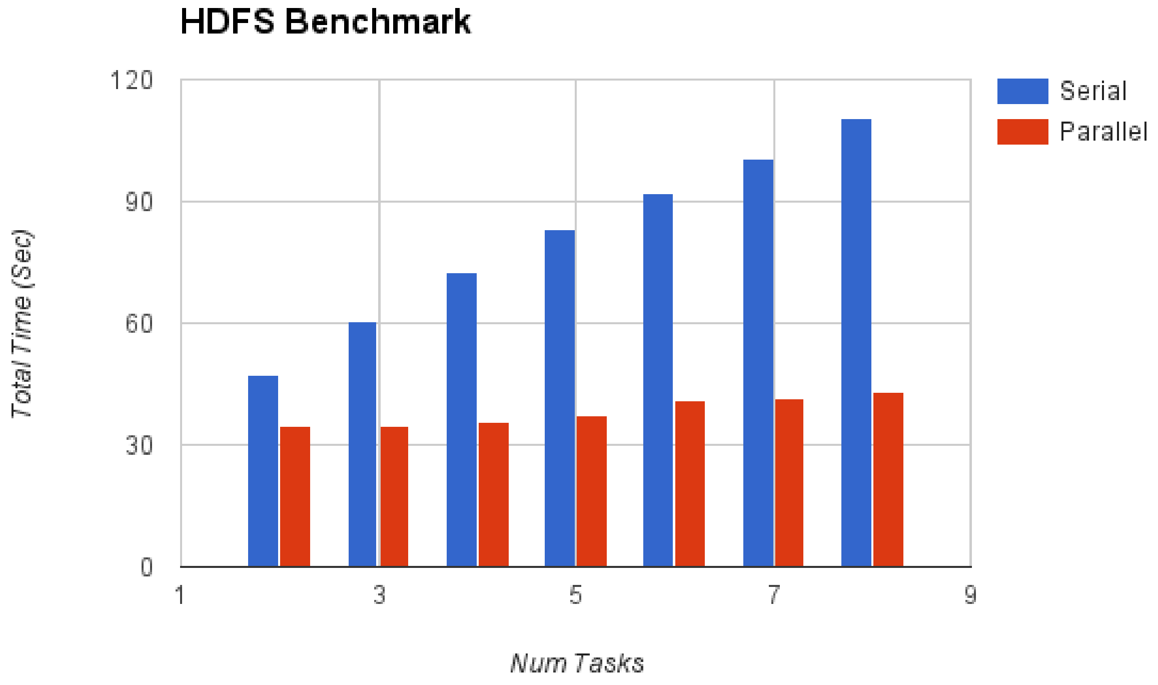

Figure 5.3: HDFS Benchmark Results

steep linear rate as more tasks are executed. The version of Spark with MDRDD increases at a much lower rate. Traditional Spark with RDDs took 110 seconds to run all eight tasks serially 50 times and Spark with MDRDDs took 43 seconds to run all eight tasks in parallel 50 times. This is an improvement of about 60%. Again, this is not an increase in overall Spark performance. This is an improvement for specific cases where multiple tasks need to be run on the same set of data located in HDFS at the same time.

In conclusion, my extension to Spark is successful in providing efficient, user-friendly task-parallel processing. In a relatively small number of lines of code, a developer is able to direct Spark to execute multiple tasks in parallel on a large collection of data in a distributed fashion. Not only is it simple to implement, but my experiments have shown that it provides a large speed up over serially running these same tasks using traditional Spark RDDs.

Chapter 6

Related Work

In this chapter, we consider related work to this thesis. There have been several similar studies to blend HPC technologies with cloud computing frameworks. If many of the technologies coming out of these studies are put together into a single framework or common set of tools, users who are locked in to MPI and traditional HPC technologies may see enough reason to begin using these tools instead.

6.1 RDMA Shuffle

The most directly applicable work similar to the one conducted here is The Ohio State University's Network Based Computing Lab's [26] high-performance big data [27] group of studies. In these studies, they bring technologies typically used in HPC to cloud computing frameworks such as Hadoop and Spark. Two efforts worth noting are their addition of RDMA shuffle to Spark via InfiniBand [24] and their addition of RDMA shuffle to Hadoop [28]. As mentioned earlier, technologies such as InfiniBand allow HPC solutions to enjoy low latency, high bandwidth network interconnects, which typically result in the network not being the primary resource bottleneck. In these studies, the benefits of RDMA via InfiniBand have been successfully added to Spark and Hadoop, giving considerable speedups for many use cases.

6.2 MPI Reliability

There have been a number of studies aiming to accomplish the opposite of this study: bring cloud computing features to HPC technologies. While MPI will always remain a low-level, API-based transport layer library that is relatively difficult to master. It is missing some key features that frameworks like Spark and Hadoop have gotten right from the beginning. The most notable feature is high-reliability. MPI does not handle scenarios where hardware (or other resources) fail mid-job. If this occurs, jobs must be restarted from the beginning, which can be a major concern for large-scale, complex jobs. on the other hand, Hadoop and Spark are built on the underlying premise that they will be run on commodity hardware that will fail quite often. They are able to detect hardware failures quickly mid-job and reassign portions of work to other workers quickly. There are several ongoing efforts to bring some level of reliability like this to MPI [30] [31] [32] [29] [42].

Chapter 7

Possible Extensions

The main contribution of this thesis is the addition of an API to Spark to allow simple task-parallel (MPMD) processing capabilities. If this version of Spark is further extended, it can begin to be used in place of traditional HPC frameworks and tools for both academic and industry applications. In this section I propose several follow-up studies that could bring Spark closer to this goal.

7.1 Parallel Map Reduce

In this study, the only task-parallel extension I added to Spark was the ability to execute functions in parallel via the *map* function. While this is the first, most obvious, extension, it would be worthwhile also to extend Spark to allow parallel map-reduce functionality. Users can already perform map-reduce operations, similar to Hadoop, in Spark by utilizing the pluggable shuffle mechanism. If multiple map-reduce operations could be performed in parallel, similarly to how I extended Spark to perform several maps in parallel, this would comprise an even more powerful tool.

7.2 Extend MLLib

One popular use case for Spark is machine learning. The MLLib library provided with releases of Spark provides users with an easy-to-use API to execute popular machine learning algorithms on their datasets. It would be a worthwhile study to speed up these MLLib algorithms with the task-parallel functionality provided by my extension to Spark.

7.3 InfiniBand Shuffling

The group at Ohio State's High-Performance Big Data Laboratory [27] have already extended Spark to include an RDMA (over InfiniBand) version of the shuffler [24]. This shuffler allows all shuffling that takes place between the map and reduce phases of map-reduce Spark programs to occur over high-speed RDMA communication via InfiniBand. It would be a worthwhile study to add and revalidate this study alongside the task-parallel processing added via MDRDDs.

7.4 Topology-Aware Communication

One the key-features of MPI is the ability to distribute data and collect results in a topologically aware manner. This means that the actual work of distributing data among processes and the actual work of collecting results (for example in a reduce operation) can be distributed among processes, drastically reducing the amount of work the master node needs to do in common use cases. Spark, on the other hand, distributes and collects data in a naive way. If data needs to be distributed, the designated master processes does so in a simple loop. If data needs to be collected and/or reduced, this is done solely at the master. It would be an interesting study to extend Spark further in order to provide a smarter, more distributed distribution/collection mechanism that is topologically aware.

Chapter 8

Conclusion

Traditional HPC technologies provide the tools needed to perform high-performance, parallel computations on a distributed set of resources. However, these tools are outdated and often difficult to master. MPI specifically forces developers to think in low-level terms and not at a higher level of abstraction. In recent years, modern cloud computing frameworks, such as Apache Spark, have begun to overtake these legacy HPC technologies in both academic and commercial settings. While these modern frameworks are much more user-friendly and allow developers to think and operate at a much higher level of abstraction, they don't presently provide all of the same functionality as some of the older HPC tools and many times cannot achieve the same level of performance. MPI provides developers with the tools to execute both data-parallel and task-parallel processing. It also provides relatively easy integration with high speed network interconnects such as InfiniBand, without requiring developers to be experts at InfiniBand verbs. Modern cloud computing frameworks such as Apache Hadoop and Apache Spark are great tools for specific use cases, namely non-communicative, embarrassingly parallel problems, but they are lacking when it comes to other classes of problems that call for more flexibility and power. If a problem requires communicative parallelism or calls for task-parallelism in order to be efficient, Hadoop and Spark are poor choices. This is a shame because these tools allow developers to work in much higher levels of abstraction and typically require less code and less expertise in parallel computing.

The eventual goal of this study is to extend Apache Spark to be a complete, or nearly complete, replacement for MPI in HPC settings. In order to achieve this outcome, there are many deficiencies of Spark that need to be improved upon so that all classes of problems that MPI can solve efficiently are also efficiently solved in Apache Spark. In this thesis, I took on one of thse deficiencies: I extended Spark to be capable of task-parallel processing via a simple API that matches the existing API that allows for straightforward data-parallel processing. I wrapped existing structures within Spark to introduce the novel Multiply Distributed Resilient Distributed Dataset (MDRDD). With MDRDDs, developers can clearly and efficiently prepare data for task-parallel processing and execute said processing in a very small amount of code that is clear, highly-abstracted from primitive data structures, and easy to maintain. I demonstrated that my extension to Spark is efficient at processing data in parallel. When compared with running the same operations serially, my extension saw a speed up of about 60%.

While this thesis was a success inasmuch as it brought a popular cloud computing framework (Apache Spark) closer to the power of MPI for HPC scenarios, there is still productive work to do. I only extended Spark to execute functions non-communicatively in parallel. It would be an interesting study to further extend Spark to be capable of reducing (aggregating) in parallel, so that MapReduce-type operations can be done in parallel. A group at Ohio State has extended Spark to be capable of MapReduce shuffling over InfiniBand [24]. It would be a worthwhile effort to add this extension alongside the extensions of this thesis to learn what gains it may produce. The MLLib [22] Spark library provides easy-to-use distributed machine learning algorithms that are powered by Spark. It would be a worthwhile project to boost this library with the task-parallelism introduced in this thesis.

Once a few of these follow-on projects are completed, Spark will become an ideal framework for HPC users. It already offers data and task parallel processing with my extension. It can offer RDMA Infiniband shuffling with the extension developed at Ohio State [24]. It can be further extended to support parallel reductions, and further still to offer boosted machine learning capabilities. At that point, it would be great to partner with an HPC user who is willing to move away from MPI and see if it can prove itself in the industry, empowering them with easy-to-use, modern tools that will make their jobs easier, for quicker turn around on complex problems.

Bibliography

[1] ”Message Passing Interface Forum.” Message Passing Interface (MPI) Forum Home Page. MPI Forum, n.d. Web. 28 Oct. 2015. http://www.mpi-forum.org/.

[2] CORPORATE The MPI Forum. 1993. MPI: a message passing interface. In Proceedings of the 1993 ACM/IEEE conference on Supercomputing (Supercomputing ’93). ACM, New York, NY, USA, 878-883.

[3] Pacheco, Peter S. Parallel Programming with MPI. San Francisco, CA: Morgan Kaufmann, 1997. Print.

[4] Radu Prodan and Thomas Fahringer. 2007. Grid Computing: Experiment Management, Tool Integration, and Scientific Workflows. Springer-Verlag, Berlin, Heidelberg.

[5] Dursi, Jonathan. ”HPC Is Dying, and MPI Is Killing It.” Jonathan Dursi. N.p., 03 Apr. 2015. Web. 28 Oct. 2015.

[6] InfiniBand Trade Association. InfiniBand Architecture Specification: Release 1.0. InfiniBand Trade Association, 2000.

[7] Liu, Jiuxing, Jiesheng Wu, and Dhabaleswar K. Panda. ”High performance RDMA-based MPI implementation over InfiniBand.” International Journal of Parallel Programming 32.3 (2004): 167-198.

[8] ”Akka Documentation.” Akka. Typesafe, n.d. Web. 28 Oct. 2015. http://akka.io/docs/.

[9] ”Apache Spark - Lightning-Fast Cluster Computing.” Apache Spark - Lightning-Fast Cluster Computing. Apache, n.d. Web. 28 Oct. 2015. http://spark.apache.org/.

[10] ”Welcome to Apache Hadoop!” ¡i¿Welcome to Apache Hadoop!¡/i¿ Apache, n.d. Web. 30 Oct. 2015. https://hadoop.apache.org/.

[11] William Gropp, Ewing Lusk, Nathan Doss, Anthony Skjellum, A high-performance, portable implementation of the MPI message passing interface standard, Parallel Computing, Volume 22, Issue 6, 1996

[12] Jiuxing Liu et al., ”Performance Comparison of MPI Implementations over InfiniBand, Myrinet and Quadrics,” Supercomputing, 2003 ACM/IEEE Conference, 2003, pp. 58-58.

[13] R. T. Aulwes et al., "Architecture of LA-MPI, a network-fault-tolerant MPI," Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International, 2004, pp. 15-.

[14] Aurlien Bouteiller, Franck Cappello, Thomas Herault, Graud Krawezik, Pierre Lemarinier, and Frdric Magniette. 2003. MPICH-V2: a Fault Tolerant MPI for Volatile Nodes based on Pessimistic Sender Based Message Logging. In Proceedings of the 2003 ACM/IEEE conference on Supercomputing (SC '03). ACM, New York, NY, USA, 25-.

[15] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. Commun. ACM 51, 1 (January 2008), 107-113.

[16] Apache Software Foundation. "HDFS Architecture Guide", https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html. Accessed 06-2016.

[17] Apache Software Foundation. "Apache HBase Home", https://hbase.apache.org/. Accessed 06-2016.

[18] "Altior's AltraSTAR Hadoop Storage Accelerator and Optimizer Now Certified on CDH4 (Cloudera's Distribution Including Apache Hadoop Version 4)" (Press release). Eatontown, NJ: Altior Inc. 2012-12-18.

[19] "Erlang Programming Language" Erlang, 20 June 2016. https://www.erlang.org/.

[20] "The Scala Programming Language" Ecole Polytechnique, 20 June 2016. http://www.scala-lang.org/.

[21] "Welcome to The Apache Software Foundation!" Apache Software Foundation, 20 June 2016. http://www.apache.org

[22] Meng, Xiangrui, et al. "Mllib: Machine learning in apache spark." JMLR 17.34 (2016): 1-7.

[23] Zaharia, Matei, et al. "Spark: cluster computing with working sets." HotCloud 10 (2010): 10-10.

[24] Lu, Xiaoyi, et al. "Accelerating spark with RDMA for big data processing: Early experiences." 2014 IEEE 22nd Annual Symposium on High-Performance Interconnects. IEEE, 2014.

[25] Flynn, Michael J. "Some computer organizations and their effectiveness." IEEE transactions on computers 100.9 (1972): 948-960.

[26] The Ohio State University. "NOWLAB :: Home". http://nowlab.cse.ohio-state.edu/

[27] The Ohio State University. "High-Performance Big Data". http://hibd.cse.ohio-state.edu/

[28] Performance Modeling for RDMA-Enhanced Hadoop MapReduce. W. Rahman, X. Lu, N. Islam, D. Panda. 43rd International Conference on Parallel Processing (ICPP), Sep 2014.

[29] Joshua Hursey, Thomas Naughton, Geoffroy Vallee, and Richard L. Graham. A log-scaling fault tolerant agreement algorithm for a fault tolerant MPI. In EuroMPI, pages 255263, 2011.

[30] Wesley Bland. User level failure mitigation in MPI. In Euro-Par 2012: Parallel Processing Workshops, pages 499504, 2013.

[31] A. Hassani, A. Skjellum and R. Brightwell, "Design and Evaluation of FA-MPI, a Transactional Resilience Scheme for Non-blocking MPI," 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Atlanta, GA, 2014, pp. 750-755.

[32] Hassani, Amin, et al. "Practical resilient cases for FA-MPI, a transactional fault-tolerant MPI." Proceedings of the 3rd Workshop on Exascale MPI. ACM, 2015.

[33] SchedMD. "Simple Linux Utility for Resource Management". http://slurm.schedmd.com/.

[34] IBM. "IBM Spectrum LSF - Overview". http://www-03.ibm.com/systems/spectrum-computing/products/lsf/.

[35] Adaptive Computing Inc. "TORQUE Resource Manager". http://www.adaptivecomputing.com/products/open-source/torque/.

[36] Oracle. "Oracle Grid Engine". http://www.oracle.com/us/products/tools/oracle-grid-engine-075549.html.

[37] Apache Software Foundation. "Running Spark on YARN". https://spark.apache.org/docs/1.5.2/running-on-yarn.html

[38] Apache Software Foundation. "Apache Mesos". http://mesos.apache.org/

[39] Hoefler, T., Kambadur, P., Graham, R.L., Shipman, G., Lumsdaine, A.: A Case for Standard Non-Blocking Collective Operations. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface, EuroPVM/MPI 2007. Volume 4757., Springer (October 2007) 125134

[40] The HDF Group. "HDF Group - HDF5". https://www.hdfgroup.org/HDF5/

[41] Gu, Lei, and Huan Li. "Memory or time: Performance evaluation for iterative operation on hadoop and spark." High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing $HPCC\_EUC$, 2013 IEEE 10th International Conference on. IEEE, 2013.

[42] Teranishi, Keita, and Michael A. Heroux. "Toward local failure local recovery resilience model using MPI-ULFM." Proceedings of the 21st European MPI Users' Group Meeting. ACM, 2014.

[43] Liu, Jiuxing, Jiesheng Wu, and Dhabaleswar K. Panda. "High performance RDMA-based MPI implementation over InfiniBand." International Journal of Parallel Programming 32.3 (2004): 167-198.