Experiences Running and Optimizing the Berkeley Data Analytics Stack on Cray Platforms

Kristyn J. Maschhoff
Michael F. Ringenburg
Cray, Inc.
901 Fifth Avenue, Suite 1000
Seattle, WA 98164
{*kristyn,mikeri*}*@cray.com*

***Abstract*—The Berkeley Data Analytics Stack (BDAS) is an emerging framework for big data analytics. It consists of the Spark analytics framework, the Tachyon in-memory filesystem, and the Mesos cluster manager. Spark was designed as an in-memory replacement for Hadoop that can in some cases improve performance by up to 100X.**

In this paper, we describe our experiences running BDAS on the new Cray Urika-XA extreme analytics platform, on Cray XC systems, and on a prototype Aries-based system with node-local SSDs. We discuss how we configured and optimized the BDAS stack, and describe the execution environment used on each platform. BDAS applications differ significantly from traditional HPC applications: they run in the Java Virtual Machine, and communicate via TCP/IP. We explore how Cray system capabilities, such as the Aries interconnect and SSDs, can be better leveraged to improve performance of these types of applications.

***Keywords*-Spark; Tachyon; Berkeley Data Analytics Stack; Urika-XA; Cray XC; data analytics; big data**

I. Introduction

The Berkeley Data Analytics Stack (BDAS) [1] is quickly gaining traction as a next-generation data analytics software stack. The most widely used piece of the BDAS stack is the Spark in-memory analytics framework [2], [3], which provides APIs for abstracting, transforming, and analyzing large distributed datasets in Java, Scala, and Python. In contrast with Hadoop Map-Reduce, Spark attempts to keep datasets in memory as much as possible, thus avoiding the high overheads of I/O. Spark also includes extensions for graph processing (GraphX [4]), processing streaming data (Spark Streaming [5]), machine learning (MLLib [6], [7]), and performing SQL queries (Spark SQL [8]). The BDAS stack also includes the Tachyon in-memory distributed file system [9], [10]. Tachyon provides the speed of a memory-based filesystem combined with the fault tolerance of a disk-based filesystem, via a combination of asynchronous check-pointing to persistent storage and recording *lineage* information—i.e., the steps necessary to regenerate a file from the last completed checkpoint. The other main component of the BDAS stack is the Mesos cluster and resource manager [11], [12]. Mesos can manage multiple types of jobs, including Hadoop, Spark, and MPI, and can schedule jobs with locality awareness (starting jobs on the nodes where the data lives).

With large memory nodes, many-core processors, and fast interconnects, Cray machines represent ideal platforms for running Spark and the other BDAS components. However, these applications differ in important ways from the HPC jobs that have typically been run on Cray systems:

- They are written in Java, Scala, or Python.
- They execute inside a Java Virtual Machine (JVM) or a Python interpreter.
- They utilize garbage collection (GC) instead of manual memory management.
- An application's workers ("*executors*" in Spark parlance) communicate with each other over TCP sockets.

We must understand these differences if we wish to achieve optimal performance for BDAS applications on Cray platforms.

In this paper, we present our work on understanding and optimizing the performance of Spark and Tachyon on three Cray platforms: the Urika-XA™ analytics appliance, the Cray XC30™, and a prototype Aries™-based system with node-local SSDs. This is ongoing work; as such, we describe the current state of our efforts, as well as potential future directions that we are considering.

A. Outline

The remainder of this paper is organized as follows:

- Section II provides background information on the BDAS stack and the Cray systems that we used for our benchmarking and tuning efforts.
- Section III describes how we configured and ran Spark applications on each of our platforms.
- Section IV details our performance numbers.
- Finally, Section V describes our ongoing efforts to optimize and tune Spark and the other BDAS components on Cray machines.

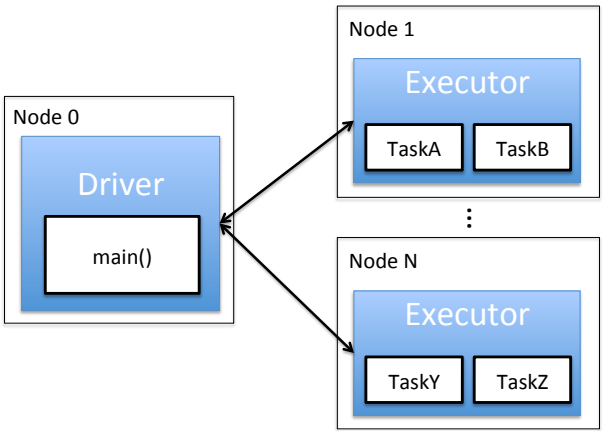

Figure 1. An illustration of a job running in the Spark framework. A driver process runs on the master node, and executes the user's `main` routine. The application partitions the data into RDDs that reside on the worker nodes. Executors on each worker node run tasks that compute operations on partitions of the RDDs.

II. Background

In this section, we first describe the Berkeley Data Analytics Stack in more detail (Section II-A). We focus primarily on the Spark analytics framework and its extensions, as it is the component responsible for building and executing user analytics applications. We then provide details on the three Cray platforms that we used for our benchmarking and tuning efforts (Section II-B).

A. The Berkeley Data Analytics Stack

1) Spark: Apache Spark provides a framework for parallel, distributed, fault-tolerant data analytics [2], [3]. A running Spark job consists of a *driver* (the "master") and a set of *executors* (the "workers"). The driver executes the user's `main` function, and distributes work to the executors. The executors operate on the data in parallel, and return results back to the driver. This is depicted in Figure 1. The driver process and each executor process runs in a separate Java Virtual Machine (JVM). The JVMs communicate via TCP sockets. In some deployment modes, a single executor runs on each compute node and achieves parallelism via using multiple cores. In other deployment modes, parallelism is achieved via a combination of multiple executors per node, and multiple cores per executor.

To program Spark applications, application developers use Java, Scala, or Python APIs to create *Resilient Distributed Datasets* (RDDs) that are partitioned across the executors, and then operate on them in parallel with a series of Spark operations. These operations come in two flavors: *transformations* and *actions*. Transformations modify the data in the RDDs, and actions return results to the driver. The Spark scheduler operates *lazily*—it tracks dependencies, and only executes transformations when they are needed to generate the data returned by an action. For example, the following lines create an RDD that spans 40 partitions, and contains the integers from 1 to 999,999:

```
val arr1M = Array.range(1,1000000)
val rdd1M = sc.parallelize(arr1M, 40)
```

Here, `sc` is a `SparkContext` object, which represents the Spark cluster and its configuration. Once an RDD is created we can begin transforming the data, for example by filtering out all of the odd elements:

```
val evens = rdd1M.filter(a => (a%2)== 0)
```

We can also perform actions that return values to the driver. For example, the following action returns the first five elements of the RDD:

```
evens.take(5)
```

If we execute this in a Spark shell (an interactive Spark environment), it returns:

```
Array[Int] = Array(2, 4, 6, 8, 10)
```

We can also count the items in the filtered RDD:

```
scala> evens.count()
res4: Long = 499999
```

No processing occurs until the actions (`take` and `count` in this example) are executed.

When a Spark action is executed, the scheduler uses its knowledge of dependencies to build a directed acyclic graph (DAG) of the computational stages. Each stage is then split into tasks, based on the partitioning of the RDD. The scheduler executes tasks on executors as executor resources become available, assuming all of the task's dependencies have been satisfied. If a stage requires communication between partitions, a barrier is inserted, and a shuffle is initiated. Shuffles are divided into two phases: the shuffle write (send), and the shuffle read (receive). Sending/writing tasks are referred to as *map tasks*, and receiving/reading tasks are referred to as *reduce tasks*. During the shuffle write, each map task "sends" data to reduce tasks by writing it to intermediate files (these intermediate files are often, but not always, cached by the OS). Once all of the mappers have written out their shuffle data, the shuffle read commences. During the read, each reducer fetches the data that was sent to it by reading the intermediate files. More recent versions of Spark have optimized this process by sorting map-task output by its intended reduce task, and aggregating data going to the same reducer into a single file.

Spark also comes packaged with specialized modules for graph analytics (GraphX) [4], machine learning (MLLib) [6], [7], processing streaming data (Spark Streaming) [5], and running SQL queries (Spark SQL) [8]. These modules all build on the RDD abstraction, and provide additional specialized abstractions and operations. For example, GraphX

provides a property graph class consisting of a `VertexRDD` and an `EdgeRDD`. Each of these optimize and extend Spark RDDs with graph-specific functionality. The `Graph` class provides methods for operating on graphs, as well as a selection of built-in graph algorithms (as of Spark 1.3.0, these included pagerank, connected components, strongly connected components, and triangle count).

2) Tachyon: Another major component of the BDAS stack is the Tachyon distributed in-memory file system [9], [10]. It allows multi-step workflows to pass data without incurring the overhead of disk I/O. Tachyon caches files in memory, and asynchronously checkpoints them to an underfilesystem (e.g., HDFS or a POSIX-compliant file system). Fault tolerance is provided by synchronously storing *lineage information* to the underfilesystem. Lineage information consists of the steps (e.g., program executions or computation phases) necessary to generate each cached file from data that has already been check-pointed to the underfilesystem. If data which has been cached in the memory of one node is needed on another node, it is sent over the network, and a copy is locally cached on the new node. Tachyon workers are Java processes which run inside of the JVM, but they avoid garbage collection overheads by storing cached data in a RamFS rather than in the Java heap.

In addition to serving as an in-memory file system to share data between jobs without the cost of disk I/O, Tachyon can also serve as an off-heap repository for Spark RDDs. When used in this mode, the primary benefit of Tachyon is reducing the GC overhead of Spark executors. This is most helpful when Spark is used with large datasets, since garbage collection becomes more expensive as the Java heap fills up.

B. Cray Analytics Platforms

We experimented with Spark on three systems: the Urika-XA extreme analytics platform, a Cray XC40, and a prototype Aries-based system with node-local SSDs. We also ran Tachyon on the prototype system. This section briefly describes these three systems.

1) Urika-XA: We used a single rack configuration of the Urika-XA extreme analytics platform, consisting of 48 compute nodes connected with FDR infiniband. Each node has dual sockets, 128 GB of DDR4-2133 memory, a 1 TB hard drive, and an 800 GB SSD. The Urika-XA we tested on was populated with 16-core Haswells, for a total of 1536 cores on the compute nodes.

The Urika-XA comes preloaded with the Cloudera Distribution of Hadoop (CDH) and Apache Spark. Both Hadoop and Spark are configured to to utilize the SSDs for intermediate file storage. The SSDs are also used for HDFS storage. The Urika-XA we tested on had CDH version 5.3 and Spark version 1.2.0.

2) Cray XC systems: Our Spark experiments on Cray XC systems were performed on our large internal XC40 development system. This is a 6 cabinet XC40 system with configurable network bandwidth. The system is populated with a heterogeneous mix of different processor types and speeds. For our experiments we targeted 2.3 GHz dual socket Haswell nodes with 128 GB DDR-4 2133 memory and 16 cores per socket (32 cores per node). We installed Spark 1.3.0 on this system, running in Cray Cluster Compatibility Mode (CCM) (as detailed in Section III-A).

The Cray XC40 system uses the custom Aries interconnect which is implemented with a high-bandwidth, low-diameter network topology called Dragonfly. The Dragonfly network topology is constructed from a configurable mix of backplane, copper and optical links, providing scalable global bandwidth.

3) Prototype Aries-based System With Node-local SSDs: We installed Spark 1.3.0 and Tachyon 0.6.1 on a prototype system with 43 compute nodes and a Cray Aries interconnect (described above). Each node has two sockets populated with 12-core Haswells (for a total of 1,032 compute cores), 128 GB of DDR-4 memory, an 800 GB SSD, and a 1 TB hard drive.

III. Running Spark and Tachyon on Cray Analytics Platforms

The Urika-XA platform comes preinstalled with the Cloudera Distribution of Hadoop, including Apache Spark. Our prototype Aries system is running a full distribution of CentOS 6.4, so installing Spark and Tachyon was straightforward—we simply followed the documented procedures to build, install, and run the software [3], [10]. Section V-A describes some of the configuration parameters we changed to optimize performance on these systems.

In this section, we describe the more involved process of installing and running Spark on a Cray XC system (Section III-A). We also describe the trace analysis tool [13], [14] that we used to profile Spark jobs on all three platforms (Section III-B).

A. Installing and Running Spark on a Cray XC

The simplest way we found to install and run Spark on XC systems is to use Cray's Cluster Compatibility Mode (CCM), and setup Spark to run in standalone deploy mode (as described in [15]). We based our procedure on a process developed for the Edison system at NERSC (thanks to Yushu Yao at Lawrence Berkeley National Lab who shared this with us). We first downloaded and built Spark (see [16]). To setup the Spark environment, we then created a module file. A simplified version of this file (once again, based on the NERSC Spark installation on Edison) is included in Figure 2. This module file uses two scripts, `findmaster.sh` (Figure 3) and `findslave.sh` (Figure 4), which should be added to the top level of the Spark directory.

These scripts assume that we are already in CCM mode, so we first request nodes from the CCM queue. On systems using PBS, this looks like:

```
#%Module1.0

## Required internal variables
set     name        spark
set     version     spark-1.3.0
set     root        /lus/scratch/$USER/$name/$version
set     scratch     /lus/scratch/$USER/SCRATCH

## List conflicting modules here
conflict $name

## List prerequisite modules here
module load ccm
module load java

#If python is available on your system, needed to use PySpark
#module load python

# Add the Spark bin, and sbin directories to your PATH
prepend-path PATH $root/bin:$root/sbin

set workerdir $scratch/$env(PBS_JOBID)
set nodefile $env(PBS_NODEFILE)
setenv SPARK_WORKER_DIR $workerdir
setenv SPARK_SLAVES $workerdir/slaves
setenv SPARK_LOG_DIR $workerdir/sparklogs

#Note that there are several options for setting SPARK_LOCAL_DIRS
#setenv SPARK_LOCAL_DIRS "/dev/shm,$scratch"
#setenv SPARK_LOCAL_DIRS "/tmp,$scratch"
setenv SPARK_LOCAL_DIRS "/tmp"

if { [ module-info mode load ] } {
  puts stderr "Creating Directory SPARK_WORKER_DIR $env(SPARK_WORKER_DIR)"
  puts stderr "Creating $env(SPARK_WORKER_DIR)/slaves file"
  puts stderr "Determining the master node name..."
  set master [exec $root/findmaster.sh]
  puts stderr "Master node is $master"
  exec /bin/mkdir -p $env(SPARK_WORKER_DIR)
  exec  $root/findslaves.sh $master $env(SPARK_WORKER_DIR)/slaves
  setenv SPARKURL spark://$master:7077
  setenv SPARKMASTER $master
}
```

Figure 2. A template module file for setting up the Spark environment (based on a similar file from the NERSC installation on Edison).

```
qsub -I -q ccm_queue -l mppwidth = ...
```

Next, we add the path to the module file to our module path, and load the module. For example, if our module file was `${MY_MODULES}/modulesfiles/spark/1.3.0`, we would use the following commands:

```
module use ${MY_MODULES}/modulefiles
module load spark/1.3.0
```

Once the Spark module is loaded, we log into the head node of the CCM cluster (which our example module file already set as the Spark master node), using the `-V` option to propagate the Spark environment to the slave nodes:

```
ccmlogin -V
```

After we log into the head node, we can start the Spark cluster using Spark's `start-all.sh` script, and then run jobs via `spark-submit` or start an interactive Spark shell via `spark-shell`. Note that by default, Spark will start up one executor per node and set the number of cores per executor to the number of cores available on the node. In practice, it is often useful to limit the number of cores per node by passing the command line option `--total-executor-cores`. When we are done running Spark, we can shut down the cluster via Spark's `stop-all.sh` script.

As noted in the example modulefile template, there are several options for setting the `SPARK_LOCAL_DIRS` environment variable, which specifies the directory (or direc-

```
#!/bin/bash -l
module load ccm >& /dev/null
ccmrun hostname 2> /dev/null | head -n1
```

Figure 3. The `findmaster.sh` script (borrowed from the NERSC installation on Edison).

```
#!/bin/bash l
MASTER=$1
FILE=$2
cat $PBS_NODEFILE | uniq | sort| uniq \
                  | grep -v $MASTER > $2
```

Figure 4. The `findslave.sh` script (borrowed from the NERSC installation on Edison).

tories) to use for "scratch" space in Spark. On Urika-XA and our prototype Aries system, we use the fast local SSDs present on the compute blades. On our existing XC systems, however, the only available fast local storage is a RAM-based filesystem. Although critical for performance, this has the disadvantage of reducing the amount of memory per node available to Spark. It may also cause jobs to fail if the scratch space fills up. We discuss this further in Section V-A, and consider approaches to resolving some of these issues in Section V-B.

B. Profiling Spark Applications

The trace analysis tool from Kay Ousterhout [13], [14] analyzes the event logs produced by Spark, and outputs a *waterfall diagram*. The waterfall diagram depicts when each task starts and finishes, and what they are doing at any given time. Figure 5 shows an example waterfall. The x-axis represents time, and the y-axis indicates task number. Every task is plotted as a horizontal line spanning the execution time of that task. The color of the line at each point indicates the activity the task was engaged in at that time (e.g., I/O, compute, shuffle, etc.) In Section IV-B, we describe how we used the trace analysis tool to better understand Spark behavior on our three platforms.

IV. Benchmarking Results

The two most informative benchmarks we investigated on our three systems were a Spark version of TeraSort (Section IV-A) and a Spark/GraphX PageRank on a Twitter example dataset (Section IV-B). We ran each benchmark with a variety of configurations (memory sizes, executor core counts, data partitionings), and report the optimum results on each platform. Our results are summarized in Table I and Figure 7.

A. Spark TeraSort

The Spark TeraSort benchmark is based on the widely used Hadoop TeraSort benchmark. The first phase randomly generates a set of integer-keyed data. The second phase performs a bucket sort. It first divides the data keys into buckets corresponding to value ranges. The sorter then assign a contiguous set of buckets to each node. Finally, it shuffles the data such that all keys belonging to a particular bucket end up on the correct node.

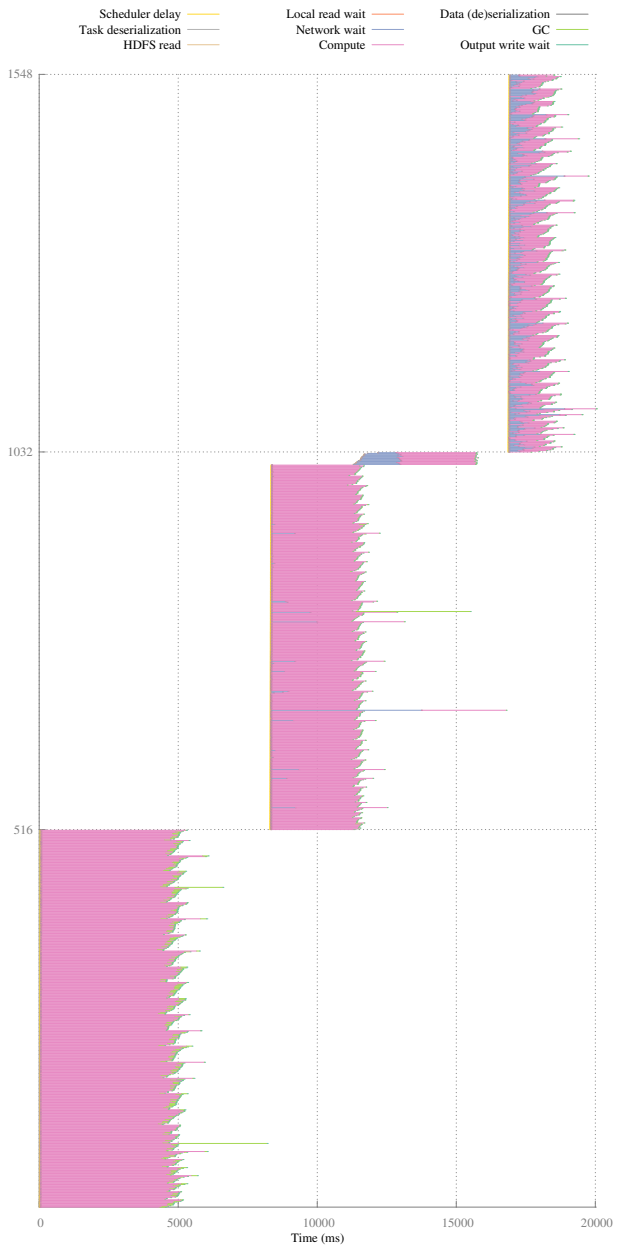

Figure 5. An example waterfall plot, corresponding to one iteration of PageRank on our prototype Aries system. The x-axis represents time, and the y-axis represents the task number. Each horizontal line corresponds to an individual task, and spans the execution time of that task. The colors of the line indicate how much time the task spent on various activities (e.g., compute, network wait, I/O, garbage collection, etc.). The small number of late-starting tasks in the second stage (the small offset at the top of the middle group of tasks) is due to tasks waiting for local executors, and is discussed further in Section V-B.

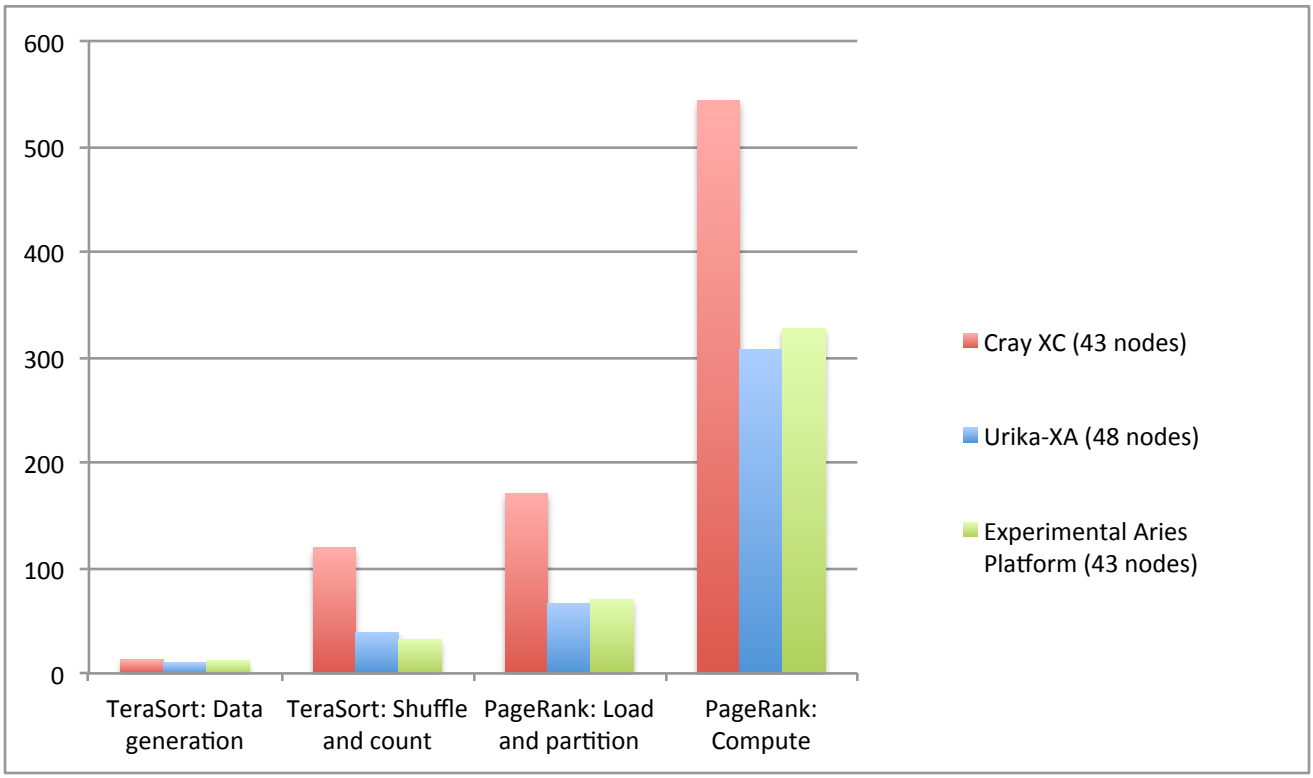

Figure 7. The performance of Spark/GraphX PageRank on the Twitter dataset and Spark TeraSort on our three systems. On the XC40 system, we used 43 compute nodes in order to match the size of the prototype Aries system.

	Cray XC40 (43 nodes, 1376 cores)	**Urika-XA (48 nodes, 1536 cores)**	**Prototype Aries System (43 nodes, 1032 cores)**
Spark Terasort (1TB): Data Generation	13 seconds	10 seconds	12 seconds
Spark Terasort (1TB): Shuffle and Count	119 seconds	39 seconds	32 seconds
PageRank (Twitter dataset): Load and Partition	171 seconds	66 seconds	70 seconds
PageRank (Twitter dataset): Compute	544 seconds	307 seconds	327 seconds

Table I
RESULTS OF RUNNING SPARK/GRAPHX PAGERANK ON THE TWITTER DATASET AND SPARK TERASORT ON OUR THREE SYSTEMS. ON THE XC40 SYSTEM, WE USED 43 COMPUTE NODES IN ORDER TO MATCH THE SIZE OF THE PROTOTYPE ARIES SYSTEM.

All of our runs were performed on a one terabyte dataset. Results are summarized in Table I and Figure 7.

B. GraphX PageRank: Twitter dataset

The PageRank algorithm we used for our studies is the static GraphX PageRank algorithm provided with the Spark distribution. We wrote our own version of the PageRank driver (rather than reusing the driver packaged with Spark) so that we could time each stage of the computation. We also inserted correctness checks into our driver. We ran PageRank on the Twitter dataset (41,652,230 vertices and 1,468,365,182 edges) from the WebGraph framework [17]

Figures 5 and 6 provide waterfall plots corresponding to individual PageRank iterations. It was helpful for our performance study to map the various regions in the waterfall diagram to the operations being performed during the algorithm. For this reason we go into some detail here on the inner workings of the algorithm implementation.

The GraphX PageRank code [18] is presented in Figure 8. Briefly, the code first (lines 5–11) creates a graph data structure (`rankGraph`), and initializes its edge attributes to the inverse of the out-degree of the source vertex and its vertex attribute to the default probability (`resetProb`). The code then enters a loop that executes iterations of PageRank.

Each iteration consists of three stages, which can be seen in the waterfall plots referenced above. The first stage (lines 21–22) sends the rank contributions for each source

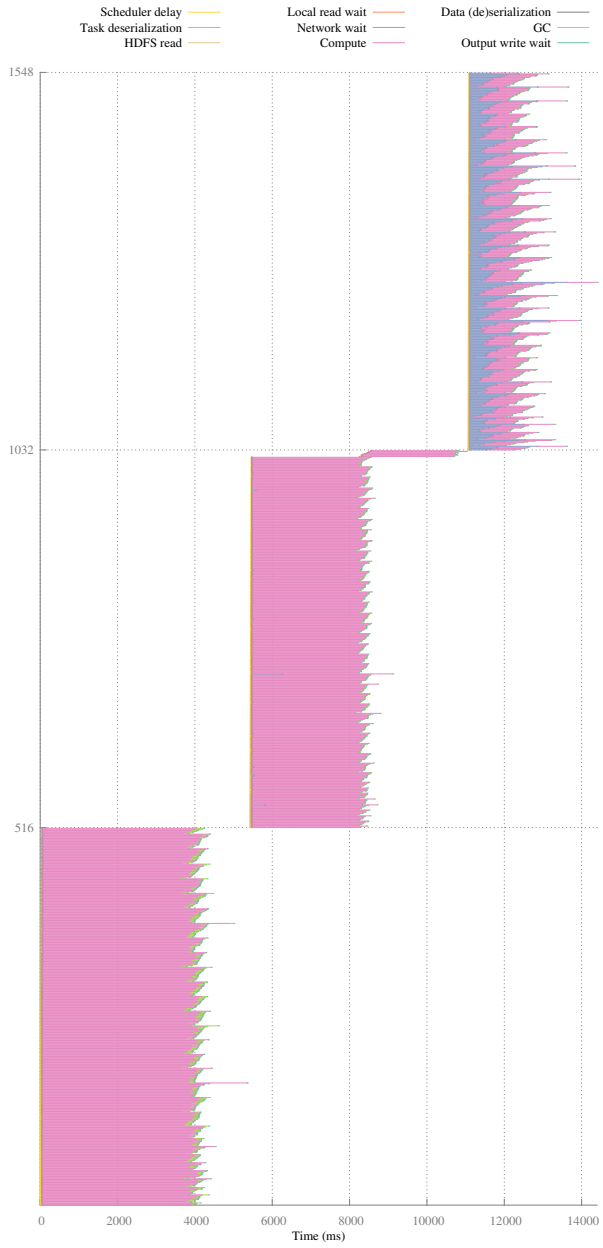

Figure 6. A waterfall plot corresponding to one iteration of PageRank on an XC30. The x-axis represents time, and the y-axis represents the task number. Each horizontal line corresponds to an individual task, and spans the execution time of that task. The colors of the line indicate how much time the task spent on various activities (e.g., compute, network wait, I/O, garbage collection, etc.). The small number of late-starting tasks in the second stage (the small offset at the top of the middle group of tasks) is due to tasks waiting for local executors, and is discussed further in Section V-B.

vertex the to destination vertices. A local *preaggregation* combines messages destined to the same vertex. Spark then performs a shuffle write to send the preaggregated data to their destinations. The second stage (lines 27–30) aggregates the rank contributions from all source vertices at each destination vertex. Spark implicitly inserts a barrier (seen in the waterfall diagrams in Figures 5 and 6) at the start of the stage to ensure all shuffle writes have completed. Since all tasks must synchronize at this barrier, any long running tasks or load imbalances get exposed here (as seen in the waterfall diagrams). After the aggregation completes, we update the vertex attribute with the updated rank computation. Finally, the third stage (line 32) *materializes* the replicated vertex view stored in the EdgeRDD by shipping all changed vertex attributes to their incident edges. For PageRank, since we are computing a new double precision rank at each iteration, we do not see a reduction in the number of updated vertex attributes shipped during the materialize phase. In other graph algorithms, such as connected components or label propagation, only shipping vertex information for the vertices whose properties have changed can provide a significant performance improvement. The second implicit barrier we see in the waterfall diagrams (Figures 5 and 6) corresponds to waiting for the write portion of this materialization shuffle to complete. We see the most network wait activity in the waterfall plots during the tasks that immediately follow this barrier. One possible explanation for this observed network latency is that this is tied to the efficiency of the routing tables used to route the updated vertex properties as part of the incremental view maintenance described in [4].

Finally, line 38 returns the ranked graph to the caller.

C. Spark Results Discussion

The results in Table I and Figure 7 show the impact of the local SSDs on Urika-XA and our prototype Aries system. The XC30 fell well short of our prototype Aries system on shuffle-intensive benchmarks, despite having nearly identical processors, memory, and interconnect[1]. The primary difference is that on our prototype system (and on Urika-XA), we were able to configure Spark to use local SSDs for its shuffle scratch directory. The XC30 does not have directly attached SSDs on the compute blades, so we instead configured the a RAM-based filesystem for scratch space, as described in Section V-A. This helps performance, but the RAM-based filesystem can fill up quickly. Thus we were forced to allocate additional space in the Lustre filesystem in order to ensure that jobs were able to complete. We discuss a proposal to mitigate this issue in Section V-B.

We also note that our prototype Aries system was competitive with (and in some cases exceeded) the performance of the Urika-XA, despite having fewer nodes and cores. The primary difference between the systems is the interconnect (Urika-XA uses FDR Infiniband). This demonstrates the suitability of the Aries interconnect for analytics applications.

[1]The Aries configuration on the XC30 actually has slightly *lower* latency than the Aries configuration on our prototype system.

```
def run[VD: ClassTag, ED: ClassTag](graph: Graph[VD, ED], numIter: Int, resetProb:
                                    Double = 0.15): Graph[Double, Double] = {
  // Initialize the PageRank graph with each edge attribute having
  // weight 1/outDegree and each vertex with attribute 1.0.
  var rankGraph: Graph[Double, Double] = graph
    // Associate the degree with each vertex
    .outerJoinVertices(graph.outDegrees) { (vid, vdata, deg) => deg.getOrElse(0) }
    // Set the weight on the edges based on the degree
    .mapTriplets( e => 1.0 / e.srcAttr, TripletFields.Src )
    // Set the vertex attributes to the initial pagerank values
    .mapVertices( (id, attr) => resetProb )

  var iteration = 0
  var prevRankGraph: Graph[Double, Double] = null
  while (iteration < numIter) {
    rankGraph.cache()

    // Compute the outgoing rank contributions of each vertex, perform local
    // preaggregation, and do the final aggregation at the receiving vertices.
    // Requires a shuffle for aggregation.
    val rankUpdates = rankGraph.aggregateMessages[Double](
      ctx => ctx.sendToDst(ctx.srcAttr * ctx.attr), _ + _, TripletFields.Src)

    // Apply the final rank updates to get the new ranks, using join to preserve
    // ranks of vertices that didn't receive a message. Requires a shuffle for
    // broadcasting updated ranks to the edge partitions.
    prevRankGraph = rankGraph
    rankGraph = rankGraph.joinVertices(rankUpdates) {
      (id, oldRank, msgSum) => resetProb + (1.0 - resetProb) * msgSum
    }.cache()

    rankGraph.edges.foreachPartition(x => {})//also materializes rankGraph.vertices

    prevRankGraph.vertices.unpersist(false)
    prevRankGraph.edges.unpersist(false)
    iteration += 1
  }
  rankGraph
}
```

Figure 8. The PageRank code [18] from GraphX.

D. Tachyon

We also configured a Tachyon file system on our prototype Aries system. We allocated 32 GB of memory per node, and pointed it to HDFS as the underfilesystem. We then benchmarked loading an edgelist file from Tachyon into GraphX using the `GraphLoader` class. When generating the Lustre results, we alternated our runs with other jobs that read in large datasets in order to flush the OS file caches. Our results are detailed in Table II. We saw at least 2x speedups when using Tachyon instead of Lustre.

V. Optimizing the BDAS Stack for Cray Systems

In this section, we describe our ongoing efforts to improve performance of the Berkeley Data Analytics Stack on Cray platforms. We first describe what we learned about tuning Spark performance on our platforms (Section V-A). We then discuss possible optimizations that could be undertaken in the future (Section V-B).

A. Configuring and Tuning Spark

The shuffle phase of Spark applications is responsible for moving data between partitions, and is the bottleneck in many Spark applications. As described in Section II-A, shuffle mappers (the senders) write data to intermediate files, and shuffle reducers (the receivers) read the data from those files (potentially fetching them over the network, if they are not local). Ideally, the intermediate files will all be cached by the OS file cache, but profiling of real jobs on all three of our systems has shown that this is often not the case for larger shuffles. The OS cache is limited by the unused memory. We have also empirically observed that the OS has increasing difficulty effectively managing the cache as the number of open file handles grows. Thus I/O rates often

Dataset	Size	(a) Lustre to GraphRDD	(b) Tachyon to GraphRDD	Tachyon Load Speedup
LiveJournal dataset	1.0 GB	13.8 seconds	5.4 seconds	2.6x
Twitter dataset	24.3 GB	41.1 seconds	19.4 seconds	2.1x

Table II
THE TIME TO LOAD TWO EDGELIST FILES INTO A SPARK GRAPHX GRAPHRDD FROM (A) LUSTRE AND (B) TACHYON ON OUR PROTOTYPE ARIES SYSTEM.

gate the performance of shuffles, and thus the performance of Spark operations that require data movement.

To mitigate the impact of shuffle I/O, we moved the Spark shuffle intermediate file directory[2] to a node-local SSD-mounted directory on Urika-XA and on our prototype Aries system. On XC systems, local SSDs are not available on the compute nodes, so we instead directed intermediate files to a RAM-based local file system (`/tmp` in CCM mode). Unfortunately, this suffers from some of the the same problems as the OS cache, in that it is size-limited to the unused RAM on the node. In order to ensure that large shuffles are still able to successfully complete, we also designate a Lustre directory as a second shuffle file location.[3] These changes resulted in speedups ranging from 1.2-1.5x on shuffle-dependent benchmarks.

We also looked at reducing the amount of data that was spilled from the OS file cache. As mentioned above, we observed that increasing the number of open file handles reduced the effectiveness of the file cache, even with the same total amount of file data. We were able to reduce the number of open files by switching from Spark's hash-based shuffle to its sort-based shuffle (sort-based shuffle was available as an option starting in Spark 1.1, and is the default in Spark 1.2 and 1.3).[4]

Spark also offers a number of job-specific configurations, including number of cores to use, and amount of memory to allocate to the Java heap of each executor. [5] As usual, increased core counts allow more data parallelism, but also increase the amount of shuffle traffic (which hits both network and I/O). In addition, increased core counts result in additional memory utilization on the compute nodes. To avoid spills to disk, we need to ensure that every executor has sufficient heap space for its data. In addition, garbage collection overhead is reduced when there is extra space available in the heap—a fuller heap triggers more frequent garbage collections, and it requires the garbage collector to work harder to find room to move objects. On the other hand, the OS cache is critical to shuffle performance, and (as mentioned above) the OS cache capacity is gated by the amount of memory *not* allocated to other applications.

Thus we need to balance the needs of the executor heaps with the needs of the OS cache (and possibly the RAM-based filesystem if we are using it for intermediate files). For moderate or larger sized datasets, we've generally found the sweet spot on all three of our systems to be around 10-12 cores per node and approximately half the node memory dedicated to executors. For instance, our 1 TB TeraSort peaked on both Urika-XA and our prototype Aries system at 10 cores per node with 6GB of heap space per core (so 60 GB allocated out of the 128 GB available on the node). For smaller datasets, we often do better allocating fewer cores—for example, running PageRank on the relatively small LiveSocial example graph dataset (4.8 million vertices, 69 million edges) that comes with Spark stopped scaling at about 128 total cores, regardless of the system size. The larger Twitter dataset (42 million vertices, 1.5 billion edges), on the other hand, continued scaling up to 12 cores per node on Urika-XA and on our prototype Aries system (576 cores on a 48 node system).

B. Future Work

We are currently exploring four directions for future improvements to Spark and Tachyon performance on Cray systems: replacing TCP sockets with native network communication protocols, retuning the balance between network and processor usage, utilizing Cray-optimized libraries for certain common operations such as linear algebra and graph algorithms, and reducing file system usage in the shuffle phase. This section explores these directions in more depth.

All inter-node communication in the BDAS framework is currently handled via TCP sockets. TCP is not a particularly performant protocol, and does not take advantage of the features of high performance interconnects like Aries and Infiniband. Previous work from the High-Performance Big Data project (HiBD) at Ohio State [19], [20], [21] and Auburn University and Mellanox [22], [23] have shown the benefits of replacing TCP-based communication with RDMA over infiniband in the context of big data analytics. We are exploring the possibilities of using the RDMA capabilities of the Aries interconnect in similar ways to accelerate Spark and Tachyon communications.

[2]This is configurable via the `spark.local.dir` configuration parameter and the `SPARK_LOCAL_DIRS` environment variable.

[3]Spark allows a comma-separated list of local intermediate file directories.

[4]This is settable via the `spark.shuffle.manager` configuration parameter.

[5]The details of how to allocate cores vary between Spark deployment types—see the `spark-submit` documentation for details.

The default configuration of Spark assumes a minimally-capable commodity interconnect. This assumption informs many of the parameter choices. For example, Spark enables shuffle compression in the default configuration, which may spend many compute cycles compressing data in order to reduce network usage. Spark's defaults also allow the scheduler to wait up to three seconds for an executor with a task's preferred locality to become available, and limit the amount of data in flight at any time. Our preliminary investigations have indicated that many of these choices may not be ideal for more capable interconnects like Aries. For instance, turning off shuffle compresssion on our prototype Aries system reduced the running time of communication-intensive workloads by 20–25%. Eliminating the locality wait also removed the late starting tasks in the second stage of the waterfall diagrams in Figures 5 and 6, and improved PageRank execution times by 5–10%. Note that neither of these results are included in the numbers reported in Section IV, as these investigations are still ongoing.

Shivaram Venkataraman at AMPLab has had success with replacing the linear algebra libraries underlying MLLib with cloud-optimized versions [24], [25]. In addition, the Cray Graph Engine (CGE) project [26] contains built-in, native graph algorithms that in some cases outperform GraphX's built-in algorithms by 10x or more. For example, CGE runs its version of Twitter PageRank on 43 nodes of XC40 in just 27 seconds (compared to a Spark PageRank running time of 307–544 seconds on our three systems, as presented in Section IV). We plan to investigate whether we can bring similar performance gains to MLLib and GraphX by integrating Cray's optimized linear algebra and graph libraries. We also plan to explore tighter integration between GraphX and CGE, such that Spark/GraphX applications could directly call the Cray Graph Engine.

As mentioned in Section IV-C, our Spark performance on XC systems appears to be limited by space constraints on the local RAM-based filesystem we are using for shuffle files. In order to ensure that jobs are able to complete, we must also allocate shuffle file space in Lustre. When jobs exhaust the RAM file space and hit Lustre, performance suffers. We believe there may be opportunities to improve the Spark `ContextCleaner` in order to reduce the amount of file system space used by Spark jobs that require many shuffles. This would in turn allow more of the shuffle data to be stored locally. We may also investigate the feasibility of new shuffle architectures that bypass the filesystem entirely.

VI. Conclusion

The Berkeley Data Analytics stack has shown great promise as an emerging big data analytics framework. Cray is actively pursuing efforts to install, run, and optimize Spark and the rest of the BDAS stack on our systems. In this paper, we described our ongoing efforts on three systems: the Urika-XA extreme analytics platform, Cray XC systems, and a prototype Aries-based system with node-local SSDs. Our preliminary results are encouraging, and we are currently focused on further performance optimizations.

Acknowledgment

The authors would like to thank Bill Sparks and Yushu Yao for their assistance with setting up Spark on the XC30, and James Clough for his assistance with our prototype Aries system. We would also like to thank Rob Vesse and Rolland Waters for their assistance with Tachyon, and Richard Korry for his assistance with Spark benchmarking. Thank you as well to Kay Ousterhout for providing assistance with her trace analysis tool.

References

[1] *Software—AMPLab*, 2015 (accessed March 30, 2015). [Online]. Available: https://amplab.cs.berkeley.edu/software/

[2] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)*, San Jose, CA, 2012, pp. 15–28.

[3] *Apache Spark—Lightning-fast Cluster Computing*, 2015 (accessed March 30, 2015). [Online]. Available: http://spark.apache.org/

[4] R. S. Xin, D. Crankshaw, A. Dave, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "GraphX: Unifying Data-Parallel and Graph-Parallel Analytics," *ArXiv e-prints*, Feb. 2014.

[5] *Spark Streaming*, 2015 (accessed March 30, 2015). [Online]. Available: https://spark.apache.org/streaming/

[6] *Spark MLLib*, 2015 (accessed March 30, 2015). [Online]. Available: https://spark.apache.org/mllib/

[7] T. Kraska, A. Talwalkar, J. Duchi, R. Griffith, and M. J. Michael J. Franklin, "MLbase: A distributed machine learning system," in *Conference on Innovative Data Systems Research*, 2013.

[8] *Spark SQL*, 2015 (accessed March 30, 2015). [Online]. Available: https://spark.apache.org/sql/

[9] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Tachyon: Reliable, memory speed storage for cluster computing frameworks," in *Proceedings of the ACM Symposium on Cloud Computing*, Seattle, WA, Nov. 2014, pp. 1–15.

[10] *Tachyon Overview*, 2015 (accessed March 30, 2015). [Online]. Available: http://tachyon-project.org/

[11] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: a platform for fine-grained resource sharing in the data center," in *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI '11)*, Boston, MA, March 2011.

[12] *Apache Mesos*, 2015 (accessed March 31, 2015). [Online]. Available: http://mesos.apache.org

[13] K. Ousterhout, *Spark Performance Analysis*, 2015 (accessed March 30, 2015). [Online]. Available: http://www.eecs.berkeley.edu/~keo/traces/

[14] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making sense of performance in data analytics frameworks," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*, Oakland, CA, May 2015, pp. 293–307.

[15] *Spark Standalone Mode*, 2015 (accessed March 30, 2015). [Online]. Available: http://spark.apache.org/docs/latest/spark-standalone.html

[16] *Building Spark*, 2015 (accessed March 30, 2015). [Online]. Available: http://spark.apache.org/docs/latest/building-spark.html

[17] P. B. S. Vigna, "The WebGraph framework I: Compression techniques," in *13th International WWW Conference*, New York, NY, 2004.

[18] *PageRank.scala*, 2015 (accessed April 2, 2015). [Online]. Available: https://github.com/apache/spark/blob/v1.3.0/graphx/src/main/scala/org/apache/spark/graphx/lib/PageRank.scala

[19] X. Lu, W. Rahman, N. Islam, D. Shankar, and D. Panda, "Accelerating Spark with RDMA for big data processing: Early experiences," in *International Symposium on High Performance Interconnects (HotI'14)*, August 2014.

[20] N. Islam, W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. Panda, "High performance RDMA-based design of HDFS over InfiniBand," in *International Conference on Supercomputing (SC '12)*, November 2012.

[21] W. Rahman, X. Lu, N. Islam, R. Rajachandrasekar, and D. Panda, "High-performance design of YARN MapReduce on modern HPC clusters with Lustre and RDMA," in *International Parallel and Distributed Processing Symposium (IPDPS '15)*, May 2015.

[22] *Unstructured Data Accelerator (UDA)*, 2015 (accessed April 3, 2015). [Online]. Available: http://www.mellanox.com/page/products_dyn?product_family=144

[23] *Efficient MapReduce for Big Data Analytics*, 2015 (accessed April 3, 2015). [Online]. Available: http://pasl.eng.auburn.edu/doku/doku.php?id=hadoopa

[24] *amplab/ml-matrix*, 2015 (accessed April 3, 2015). [Online]. Available: https://github.com/amplab/ml-matrix

[25] E. Sparks, *ML Pipelines*, 2014 (accessed April 6, 2015). [Online]. Available: https://amplab.cs.berkeley.edu/ml-pipelines/

[26] K. Maschhoff, R. Vesse, and J. Maltby, "Porting the Urika-GD graph analytic database to the XC30/40 platform," in *Cray User Group Conference (CUG '15)*, Chicago, IL, 2015.