Characterizing the Performance of Analytics Workloads on the Cray XC40

Michael F. Ringenburg
Shuxia Zhang
Kristyn J. Maschhoff
Bill Sparks
Cray, Inc.
{*mikeri,szhang,kristyn,jsparks*}*@cray.com*

Evan Racah
Prabhat
Lawrence Berkeley National Laboratory
{*eracah,prabhat*}*@lbl.gov*

***Abstract*—This paper describes an investigation of the performance characteristics of high performance data analytics (HPDA) workloads on the Cray XC40™, with a focus on commonly-used open source analytics frameworks like Apache Spark. We look at two types of Spark workloads: the Spark benchmarks from the Intel HiBench 4.0 suite and a CX matrix decomposition algorithm. We study performance from both the bottom-up view (via system metrics) and the top-down view (via application log analysis), and show how these two views can help identify performance bottlenecks and system issues impacting data analytics workload performance. Based on this study, we provide recommendations for improving the performance of analytics workloads on the XC40.**

***Keywords*-Spark; Cray XC40; data analytics; big data**

I. Introduction

Big Data Analytics is rapidly becoming a critical capability required by organizations in our digital world. Analysts expect the "digital universe" to grow from approximately 3.2 zettabytes (2.3 billion terabytes) in 2014 to over 40 zettabytes by 2020, with many organizations expecting their data volumes to increase by 50x over that time period [1]. Up to 85% of this growth is likely to come from new data sources, including the "Internet of Things"—wireless sensors remotely connected to public and private networks [1]. The technology market intelligence firm ABI research expects that over 30 billion sensors will be wirelessly connected by 2020 [2].

Cray systems, with their high compute and memory densities, fast parallel I/O, and highly scalable interconnects, make excellent platforms for running data analytics applications. Thus, it is not surprising that many users of Cray systems are becoming interested in running common data analytics frameworks like Spark [3], [4] and Hadoop [5]. In particular, the widely-used Apache Spark open source analytics framework is generating a great deal of interest because its flexible, mostly in-memory computational model is more amenable to performance and scaling. For example, NERSC makes Spark available on their Edison and Cori systems [6].

In this paper, we investigate the performance of Cray XC40™systems running Spark. We focus on two workloads: the Spark benchmarks from the Intel HiBench 4.0 suite [7], [8], and the CX matrix decomposition algorithm Spark implementation described by Gittens et al [9]. The HiBench Suite is a representative suite of big data analytics benchmarks. On the other hand, the CX algorithm represents an interesting application of analytics frameworks to a scientific computing problem. We study performance from both ends of the spectrum: a bottom-up view of system metrics, IO, and network performance; and a top-down application level log-based look at where the analytics applications are spending their time. Based on the results of our study, we suggest methods to improve the performance of analytics workloads on these systems.

A. Outline

The remainder of this paper is organized as follows:

- Section II provides background information on the Cray systems and analytics frameworks we used in our performance analysis.
- Section III describes the tools we used for our analysis.
- Section IV contains our analysis of the HiBench Suite.
- Section V details our analysis of the CX algorithm.
- Section VI details our analysis of TCP traffic patterns and performance on the XC40.[1]
- Finally, Section VII suggests modifications to both the analytics software stack and the underlying system to improve the performance and scaling of analytics workloads on Cray systems.

II. Background

In this section, we first describe the Apache Spark analytics framework in more detail (Section II-A). We then provide details on the Cray XC40 that we used for our benchmarking and performance analysis (Section II-B).

A. Apache Spark

Apache Spark provides a framework for parallel, distributed, fault-tolerant data analytics [3], [4]. A running

[1]This is relevant due to the reliance of many analytics frameworks on TCP sockets for communication.

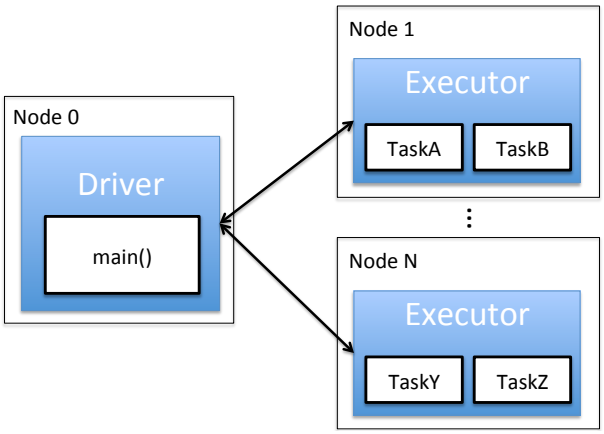

Figure 1. An illustration of a job running in the Spark framework. A driver process runs on the master node, and executes the user's `main` routine. The application partitions the data into RDDs or DataFrames that reside on the worker nodes. Executors on each worker node run tasks that compute operations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

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, as 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, developers use Java, Scala, Python, or R APIs to create *Resilient Distributed Datasets* (RDDs) or *DataFrames* 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 these commands 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
```

Due to Spark's lazy execution model, no processing occurs until the actions which return data to the driver (`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 computation required to compute the results of the action. The computation—referred to by Spark as a *job*—is broken into a series of *stages*, where a stage consists of a series of computations that can be performed locally on each partition of the data. At the end of each non-final stage, the Spark executors initiate an all-to-all data exchange, referred to as a *shuffle*. At the end of the final stage of the job, the executors return results to the driver. Each stage is then split into *tasks*, where each task represents the work of the stage on a single partition of the data. The scheduler executes tasks on executors as executor resources become available. Better load balancing is typically achieved when the Spark application's cores are oversubscribed—in other words, when there are roughly 2-3x as many partitions (and thus tasks) as allocated cores.

When a job's DAG indicates that communication is required between two stages, a barrier is inserted between the stages and a shuffle is initiated. Shuffles are divided into two phases: the shuffle write (send), and the shuffle read (receive). Borrowing Hadoop terminology, 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 local intermediate files via the block manager thread on the task's local node.[2] Once all of the map tasks have written out their shuffle data, the shuffle read commences. During the read, each reduce task requests its incoming data from the block manager threads on the remote nodes. The block managers in turn read the data from the intermediate files and send it over the network to the requesting reduce task. 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 reduce task into a single intermediate file.

[2]These intermediate files are often cached by the OS.

Spark also comes packaged with specialized modules for graph analytics (GraphX [10]), machine learning (MLLib [11], [12]), processing streaming data (Spark Streaming [13]), and running SQL queries (Spark SQL [14]). All of these modules build on Spark's RDD and DataFrame abstractions, and provide specialized abstractions and operations.

B. Cray XC40 and Spark Configuration

We performed our performance analysis on a two cabinet internal XC40 development system. The system was composed of 351 dual socket Haswell nodes, each node containing 256 GB DDR-4 memory and 16 cores per socket (32 cores per node). We installed Spark 1.5 on this system, and ran it using Cray Cluster Compatibility Mode (CCM) and Spark's standalone cluster manager (for details, see our paper from last year's CUG conference [15]). The Cray XC40 system uses the 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.

In a typical Spark installation, each cluster node has local storage space available for Spark's local scratch directory. This directory is utilized by Spark for storing data to be sent to other nodes during a shuffle (see Section II-A), and for spilling data partitions when there is insufficient memory. On the XC40, however, there is no local persistent storage. Possible options to replace this include utilizing a directory on the shared Lustre file system, or utilizing a directory in the DRAM-based tmpfs file system in /tmp. We found that exclusively using Lustre for scratch space led to very high metadata overheads and caused the Lustre Metadata Server (MDS) to become the bottleneck in shuffle performance, a finding corroborated by the work of Chaimov et al [16]. For most of the applications we ran, there was sufficient DRAM on the nodes to host the scratch directories entirely in /tmp. When there was not, we used a combination of /tmp and Lustre, biased towards /tmp.[3] If using Lustre is necessary, the impacts can be partially mitigated by leaving sufficient memory for the OS to cache some of the remote writes to Lustre. In addition to increased bandwidth and reduced latency to locally-cached data, caching allows some of the writes to Lustre to be aggregated; reducing the metadata load on the Lustre MDS.

III. Analysis Tools

In this section, we describe the analysis tools used to study the performance of our benchmark workloads. We performed three types of analyses:

[3]Spark allows you to specify multiple directories, and it will alternate between them in a round robin fashion. So, for example, if we list two directories under /tmp and one on Lustre, 2/3 of the scratch blocks will be written to the tmpfs.

Workload	KBReadAll	Reads/sec	ReadKB/sec
Wordcount Run 1	1608457044	3487	47
Wordcount Run 2	1610252818	3528	47
Wordcount Run 3	1609668696	3526	57
PageRank Run 1	20011109	224	5
PageRank Run 2	19964128	164	6
PageRank Run 3	20083990	167	6

Table I
AGGREGATED LUSTRE READ METRICS FOR THREE RUNS OF WORDCOUNT AND PAGERANK

1) `Collectl`-based analysis of system metrics on the compute nodes
2) Log-based Spark event profiling of application stages
3) Analysis of network traffic with `tcpdump` and `iperf3`

The remainder of this section discusses our `collectl`-based profiling and the Spark event logs. We describe the network traffic analysis later, in Section VI.

A. Analytics Workload Analysis with Collectl

`Collectl` is a well-known tool for collecting system-level performance data on HPC systems. We used `collectl` to profile the performance characteristics of data analytics workloads on the XC40. To analyze the large amount of data collected, we developed a set of scripts and an R interface built on top of pbdMPI R that allow us to quantify and plot system metrics including:

- Memory usage by the analytics application and the operating system
- Load imbalance
- Processor cores used by the application and OS, and their distribution
- Interconnect and file system usage

The performance information can be extracted at different levels of detail, including individual executors/nodes, groups of executors, or all executors in a job. We have found that the overhead associated with the use of `collectl` for analytics workloads is negligible.

Due to Spark's dynamic execution model and the granularity of the sampling used by `collectl` for collecting the system metrics data, we expect some variability in measured performance run to run. In order to quantify this variability, and validate the accuracy of our tool, we compared profiles of the I/O activities in three repeated runs of six benchmarks. Since the size of the initial input from Lustre should be consistent from run to run, we used the aggregated Lustre metrics associated with reading input data to validate our data collection. For this validation, we ran the WordCount and PageRank benchmarks three times using 41 compute nodes. One compute node ran the Spark master and

Benchmark	Classification	Description
Sort	Microbenchmark	Sorts input data
WordCount	Microbenchmark	Counts occurance of each word in input data
TeraSort	Microbenchmark	Standard TeraSort Benchmark by Jim Gray
Sleep	Microbenchmark	Tests framework scheduler
PageRank	Web Search	Tests Spark MLlib implementation of PageRank
Bayes	Machine Learning	Naive Bayesian Classification, Spark MLlib implementation
Kmeans	Machine Learning	K-means clustering, Spark MLlib example code

Table II
THE HIBENCH WORKLOADS SELECTED FOR OUR SPARK PERFORMANCE ANALYSIS.

the remaining 40 compute nodes ran Spark executors. We launched each executor with 30 cores. For each benchmark, we ran the Java, Scala, and Python versions. Table I shows the aggregated Lustre metrics for the six Scala runs (the results for the other APIs were similar). KBReadAll is an aggregated sum over all executors for the entire run. The other metrics are averaged values per node per second. These tests showed only a small variability in the total data read metrics for the sets of WordCount and PageRank benchmarks, and closely matched the known input data sizes.

B. Spark Event Log Analysis

Spark application executions can optionally produce *event logs* which track application configuration data, and the start and end times of all jobs, stages, and tasks. This data can be viewed, sorted, and analyzed in the Spark history server. The history server also allows users to view a number of metrics about each task, including garbage collection, scheduler delay, shuffle read and write, network fetch, deserialization, and compute times. It displays the distribution of these metrics between the tasks of each stage, and makes it easy to sort and identify outliers.

The event log can be parsed by other tools. For example, the trace analysis tool from Kay Ousterhout [17], [18] 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. Some of this functionality has been integrated into the Spark History server. However, it will only display a portion of the tasks, rather than the entire waterfall. We produced scripts to parse the event logs of the CX application runs, and used them to generate the data in Section V.

IV. HIBENCH SPARK WORKLOAD ANALYSIS

This section describes our performance analysis of a subset of the HiBench 4.0 suite workloads. We have selected the workloads shown in Table II for our performance evaluation. The selected set includes the micro-benchmarks which have Spark implementations, the Spark-MLLib implementation

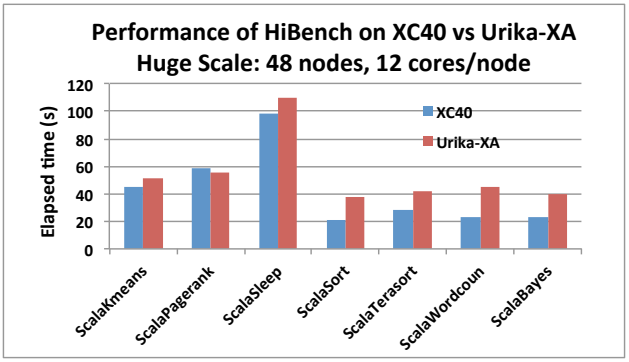

Figure 2. A perfomance comparison using the HiBench workloads on 48 nodes of XC40 versus a 48 node Infiniband FDR cluster (Urika-XA).

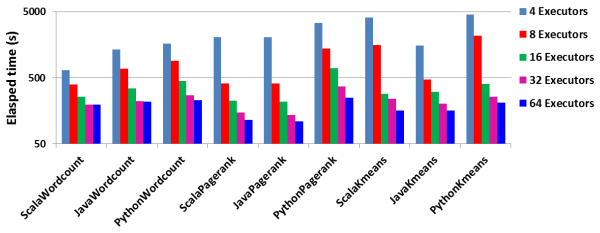

Figure 3. Scaling results from varying the node/executor count from 4 to 64 for the Scala, Java, and Python versions of the Wordcount, PageRank, and KMeans workloads.

of PageRank as a representative for Search Indexing, and two machine learning algorithms: Bayesian Classification and K-Means Clustering. The set does not include the SQL Analytics Queries (Scan, Join, and Aggregate), as these require Hive to be installed.

A. Performance and Scaling

We first looked at the raw performance of our HiBench workloads, and compared them to a Cray Urika-XA™. The Urika-XA is a 48 node analytics platform running Hadoop and Spark with an Infiniband FDR interconnect. It uses fast local SSDs for Spark and Hadoop local temporary files. For a fair comparison, we also used a 48 node allocation

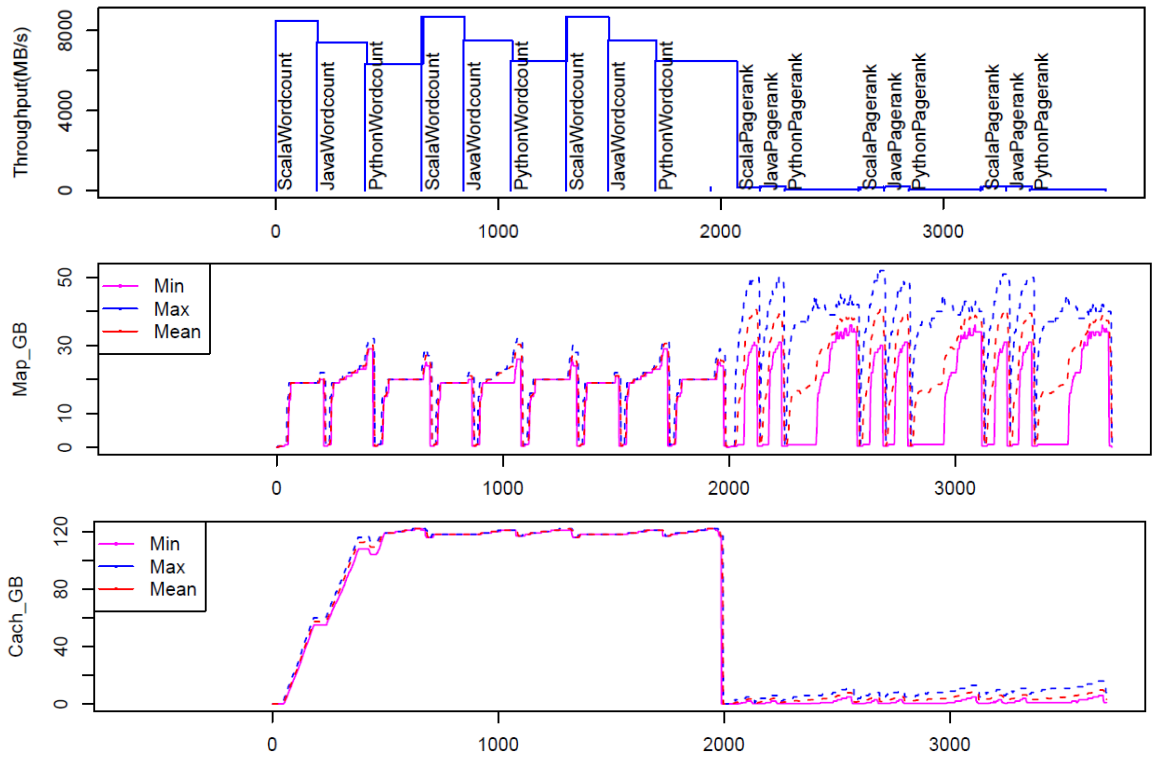

Figure 4. The minimum, maximum, and mean memory utilization by executor nodes during the execution of the Wordcount and PageRank workloads.

on our XC40. Performance on the XA appeared to peak at around 12 cores per node, so we used that setting on both platforms. Our results are shown in Figure 2. Sort, TeraSort, Wordcount, and Bayes all showed a clear advantage on the XC40. The other three workloads (Sleep, KMeans, and PageRank) had similar results on both platforms.

Next, in Figure 3, we show the scaling perfomance of our HiBench workloads. We varied the executor (node) count from 4 to 64, and used 30 cores per node. Most workloads tested continued to scale up to 64 nodes. Beyond that, scaling was limited due to the problem size and framework overheads.

B. Memory Usage

To better characterize the memory requirements for each of these benchmarks, and to show the benefits of `collectl`-based profiling, we also looked at the system memory metrics for the Wordcount and PageRank workloads. Figure 4 shows the memory usage for the Wordcount and PageRank workloads. The three graphs are aligned temporally. The top graph displays the workload being executed at each time, as well as its average throughput (input data size divided by execution time). The middle graph shows the mapped memory used by Spark while executing the workload, and the bottom graph shows the memory used by the OS file cache (which is used to buffer file I/O of input and output data, as well as temporary files used for spills and shuffles). For the mapped and cache memory graphs, we collect data from every executor node, and calculate and display the minimum, maximum, and mean node value. Our scripts are also able to provide the alternate view of mapped memory shown in Figure 5. This plot shows the memory usage of each executor (along the vertical axis) over time (the horizontal axis). As in the previous plot, the left half represent Wordcount and the right half represents PageRank.

We see in these plots that the PageRank benchmark has much more variability in mapped memory usage. This variability is indicative of an imbalance in the sizes of the partitions, due to the uneven distribution of links in the input data. We also see that Wordcount has significantly higher OS cache usage, which is indicative of more file system activity. That activity can in turn be tracked down by looking at the TCP activity shown in Figure 6. Here we see that Wordcount has significantly larger internode communication spikes. Recall that in Spark, all communication between executors occurs during shuffle stages, and the shuffle data is first written to disk by the sending nodes (thus explaining the high file cache utilization).

This ability to track the per-node memory usage over time, as well as non-application memory usage, shows the benefit of tools like `collectl`. The storage tab of the Spark History Server, in contrast, only provides a view of

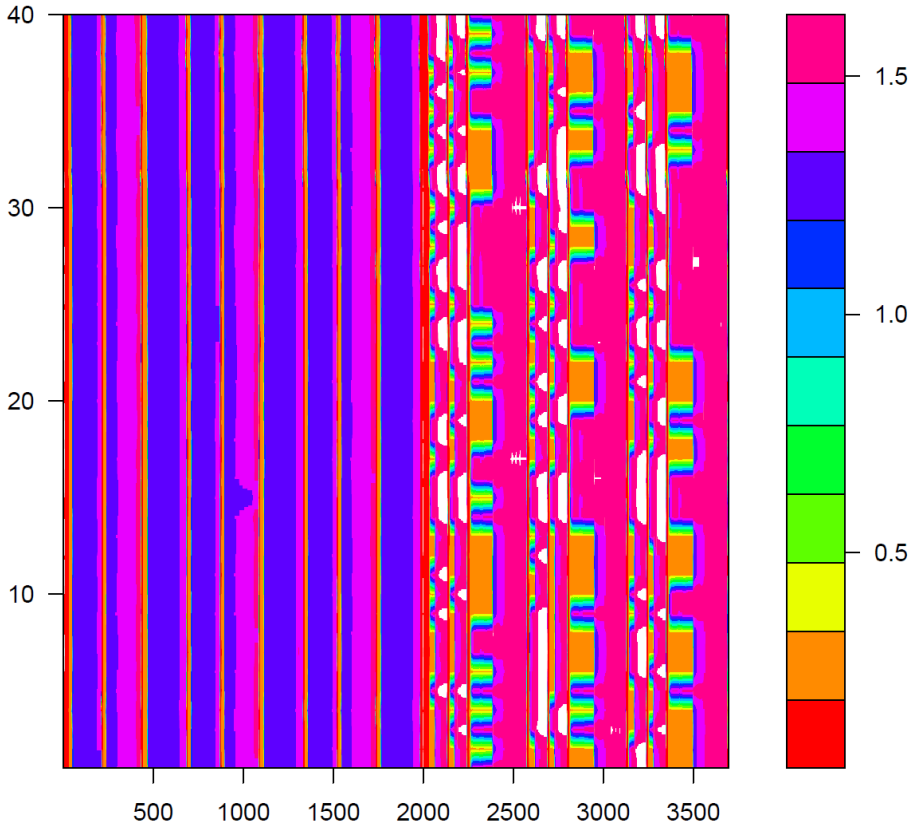

Figure 5. Executor memory usage over time for Wordcount (left) and PageRank (right). The vertical axis represents executor nodes, and the horizontal axis represents time. The color indicates application memory usage in gigabytes on a log_{10} scale.

the size of persisted RDDs and DataFrames. The Spark event logs and history server do not track usage variations over time, buffer cache usage, or the sizes of non-persisted data partitions.

V. CX for Spark algorithm

CX is a matrix factorization technique used to find a low rank (rank k) approximation of a matrix A using a subset of columns of A as opposed to a linear combination of A's columns, like in PCA. CX at a high level approximates the top k right singular vectors of A using a randomized SVD method and then uses those k vectors to define a probability distribution to sample the columns of A from. The Spark implementation of of CX we used is described in detail in Gittens et al [9]. For the purposes of our analysis, it can be divided into four phases:

- **Load Matrix Metadata**: The dimensions of the matrix are read from the filesystem to the driver.
- **Load Matrix**: A distributed read is performed to load the matrix entries into an in-memory cached RDD containing one entry per row of the matrix.
- **Power Iterations**: A series of five matrix multiplication iterations.
- **Finalization (Post-Processing)**: Collecting the data and performing a final SVD computation.

A. CX Scaling Results

Figure 7 shows how the distributed Spark portion of our code scales. We considered 240, 480, and 960 cores. An additional doubling (to 1920 cores) would be ineffective as there are only 1654 partitions in the dataset, so many cores would remain unused. When we go from 240 to 480 cores, we achieve a speedup of 1.6x: 233 seconds versus 146 seconds. However, as the number of partitions per core drops below two, and the amount of computation-per-core relative to communication overhead drops, the scaling slows down (as expected). This results in a lower speedup of 1.4x (146 seconds versus 102 seconds) from 480 to 960 cores.

B. Multi-Platform CX Comparison

Table III shows the total runtime of CX for a 1 TB bioimaging dataset on the three platforms described in Table IV. Table III also provides selected timings obtained by parsing the Spark event logs. We give two sets of results for the XC40. The first set describes our results with a configuration where we direct Spark's shuffle and spill temporary files to a mixture of tmpfs (in DRAM) and Lustre.

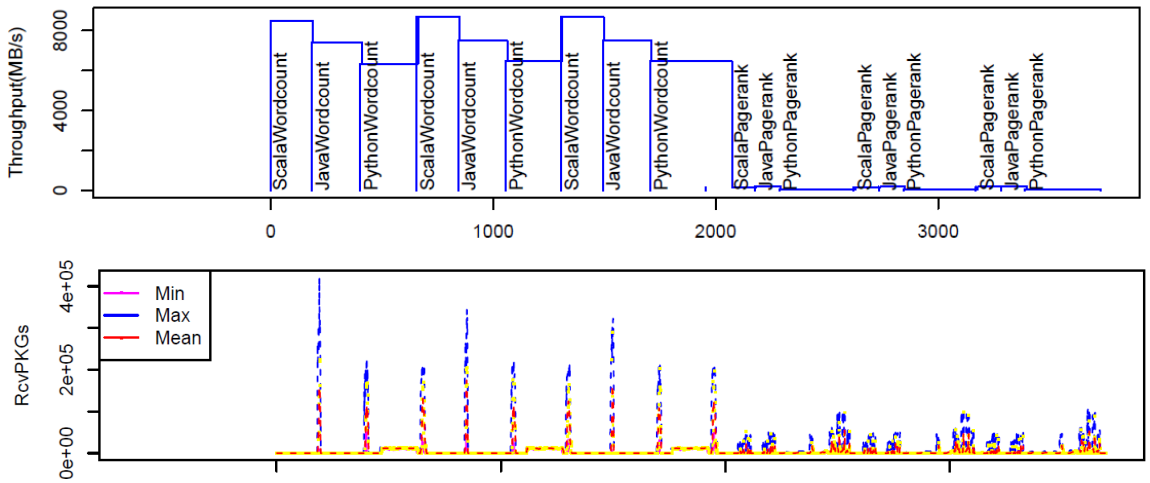

Figure 6. The minimum, maximum, and mean received packets by executor nodes during the execution of the Wordcount and PageRank workloads.

Platform	Total Runtime	Load Time	Time Per Iteration (5 total)	Average Local Task	Average Aggregation Task	Average Network Wait
Amazon EC2 r3.8xlarge w/ 10GbE	24.0 min	1.53 min	2.69 min	4.4 sec	27.1 sec	21.7 sec
Cray XC40 w/ Lustre and tmpfs	23.1 min	2.32 min	2.09 min	3.5 sec	6.8 sec	1.1 sec
Cray XC40 w/ tmpfs	18.2 min	2.28 min	1.56 min	3.0 sec	7.3 sec	1.5 sec
Aries-based cluster	15.2 min	0.88 min	1.54 min	2.8 sec	9.9 sec	2.7 sec

Table III
CX RANK 16 PERFORMANCE BREAKDOWN ON A 1 TB DATASET.

Platform	Total Cores Used	Core Frequency	Interconnect	DRAM	SSDs
Amazon EC2 r3.8xlarge w/ 10GbE	960	2.5 GHz	10 GbE	244 GB	2 x 320 GB
Cray XC40	960	2.3 GHz	Aries	252 GB	None
Aries-based cluster	960	2.5 GHz	Aries	126 GB	1 x 800 GB

Table IV
PLATFORMS USED IN THE CX PERFORMANCE COMPARISON.

This type of configuration may be necessary if the size of the shuffled data is too large to fit into the node's local memory. The second set of XC40 results show the improvement when we are able to fit all of the data into tmpfs. All platforms were able to successfully process the 1 TB dataset in under 25 minutes. Most of the variation between the platforms occurred during the power iterations and, to a lesser extent, the data loads. The data load variation can be explained by the presence of node-local SSDs on the compute nodes of the EC2 instance and the Aries-based cluster. In both of those cases, we loaded the input data set from a file system backed by the SSDs.

To better understand the remainder of the performance variations, we now consider how the Spark framework implements the power iterations. Spark divides each iteration into two stages. The first local stage computes each row's contribution, sums the local results (the rows computed by the same worker node), and records these results. The second aggregation stage combines all of the workers locally-aggregated results using a tree-structured reduction. Most of the variation between platforms occurs during this aggregation phase, where data from remote worker nodes is fetched and combined. In Spark, all inter-node data exchange occurs via shuffle operations. In a shuffle, workers with data to send write the data to their local scratch space. Once all data has been written, workers with data to retrieve from remote nodes request that data from the senders block manager, which in turns retrieves if from the senders local scratch

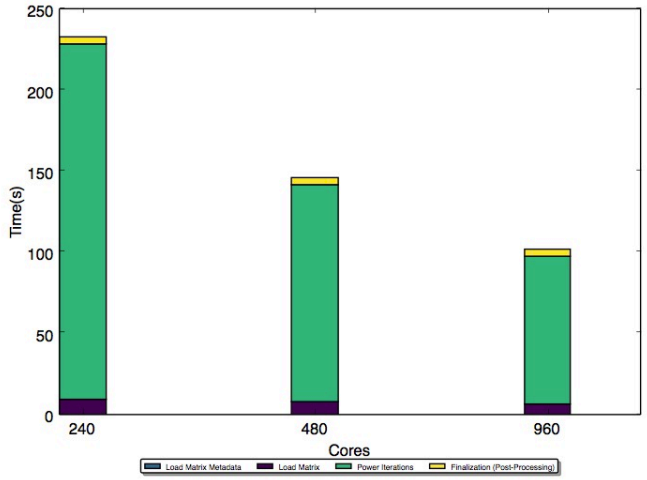

Figure 7. Scaling results on XC40 for the CX application on a 100GB dataset.

space, and sends it over the interconnect to the receiving node. Examining our three platforms (Table IV), we notice two key hardware differences that impact shuffle operations:

1) First, both the EC2 instance and the Aries-based cluster have fast SSD storage local to the compute nodes that they can use to store Spark's shuffle data. The XC40 system's nodes, on the other hand, have no local persistent storage devices. Thus, we must emulate local storage with a tmpfs in local DRAM and/or a global Lustre file system. The more performant XC40 results use exclusively tmpfs, while the other XC40 results use a combination of tmpfs and Lustre.
2) Second, the Cray XC40 and the experimental Cray cluster both communicate over the HPC-optimized Aries interconnect, while the EC2 nodes use 10 Gigabit Ethernet. We can see the impact of differing interconnect capabilities in the Average Network Wait column in Table III. The XC40 has slightly lower wait times than the Aries-based cluster due to limits on the maximum TCP bandwidth in the cluster configuration. Both are significantly faster than the 10Gb ethernet instance.

VI. Analytics Workload Network Traffic Patterns

Most commonly used analytics frameworks (like Hadoop and Spark) communicate over TCP sockets. This adds overhead, but also allows for portability across many architectures. In order to study the impact of the Aries network on TCP performance, we utilized the `iperf3` [19] tool developed by ESNet (the Energy Sciences Network). `Iperf3` allows us to measure the maximum achievable bandwidth via TCP on various networks. We ran the tool over the Aries network on our XC40 running CLE5.2 (SLES11-based), and over the Infiniband FDR network on a Urika-XA

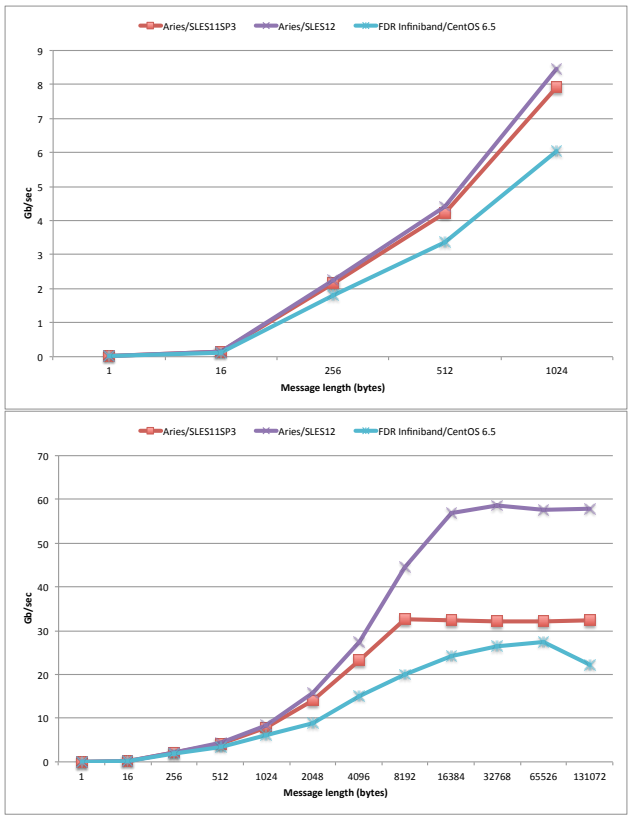

Figure 8. A comparison of maximum TCP bandwidth at various packet sizes for IP over FDR Infiniband on Urika-XA (turquoise/bottom line), IP over Aries on XC40 with the SLES11SP3 kernel (red/middle line), and IP over Aries on the XC40 with the SLES12 kernel (purple/top line). The upper graph is a zoomed in version to show differences at packet sizes under 1 KB.

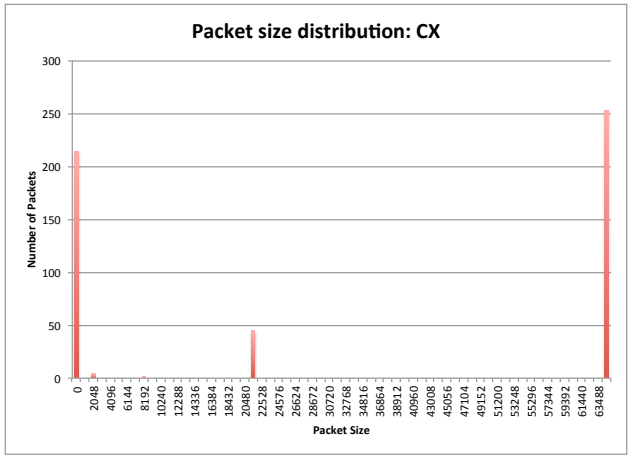

Figure 9. A plot of TCP packet size distribution for the CX benchmark.

(CentOS 6.5-based). We also reran the test on a different XC40 running CLE6.0 (SLES12-based), due to significant improvements in the TCP stack in newer linux kernels. Figure 8 shows these results. For all systems, maximum

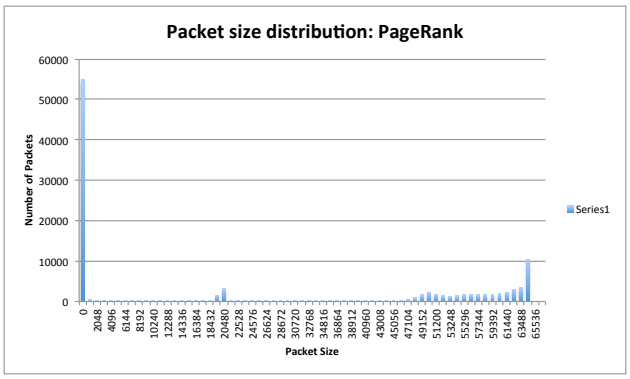

Figure 10. A plot of TCP packet size distribution for the GraphX PageRank benchmark.

bandwidth was highly dependent on packet size. However, we see that Aries significantly outperforms FDR infiniband for most packet sizes. The performance is even more striking with the new kernel. Packet sizes over 16K saw roughly twice the bandwidth with the new kernel.

In order to relate this data to analytics performance, we also did a breakdown of packet sizes (via sampling of `tcpdump` data) during two communication-intensive Spark workloads: the Spark implementation of CX matrix decompostion [9] on a one terabyte data set (Figure 9), and the GraphX PageRank algorithm on the Twitter dataset [10], [20] (Figure 10). The CX algorithm sends uniformly sized large blocks of data (matrix columns), leading to the communication pattern seen in Figure 9, where 58% of the packets sent are 21KB or larger. The large spike at 64KB represents the Aries Maximum Transmission Unit (MTU). The smaller spike at 21KB represents the remainder after dividing the transmitted data blocks into MTU-sized chunks. The smaller packets appear to be a mix of metadata, heartbeats, and other framework-associated communication. The PageRank algorithm, on the other hand, sends numerous small messages, whose distribution is determined by the nature of the graph. These are aggregated on a per-executor basis by the Spark sort-based shuffle, leading to the much less uniform communication pattern seen in Figure 10. However, we still see that 46% of packets are 16KB or larger.

VII. Improving Analytics Performance on the Cray XC40

Because of Spark's pull-based communication model (see Section II-A), where every piece of data sent over the network is first written to and then read back from a local disk or other scratch space, the efficiency of local data movement is critical if we hope to take advantage of the network throughput provided by the high-speed Aries network. This leads us to the following configuration observations and proposals for potential future optimizations:

- **Favor high-bandwidth, low-latency storage when setting Spark's scratch space directories (spark.local.dir or SPARK_LOCAL_DIRS)**: This configuration parameter determines the location where Spark writes and reads its shuffle data. It also determines the location where any spilled data is stored (Spark spills data when there is insufficient heap space to store the entire data set). If insufficient bandwidth is available to this location, we will be unable to saturate the interconnect. If possible, storing this data in a local DRAM-based filesystem (e.g., /tmp on the XC40 compute nodes) provides the best performance. If there is insufficient space available in the DRAM-based filesystem, Spark allows users to configure a list of multiple directories. Blocks will be placed in these directories in a round-robin fashion. Thus it is preferable to bias the data towards faster storage by including multiple directories on the faster devices (e.g., SPARK_LOCAL_DIRS=/tmp/spark1, /tmp/spark2, /tmp/spark3, /lus/scratch/sparkscratch/). We have also found that, if global filesystems must be used for scratch space, we see much better performance if they support client-side caching (as this allows some of the data to remain local, and also provides some aggregation of the data that does have to go to the global file system).
- **Metadata matters: Spark shuffles tend to produce large numbers of small files, and many file opens and closes**: This is a very poor access pattern for Lustre due to the large number of metadata operations. One potential fix for this would be to create a layer underneath Spark that aggregates the many small files into a single large file which remains open until all map tasks have completed their writes. This shim layer would need to maintain a mapping between the abstract small files and offsets into the actual large file.
- **Add more efficient cleanup to Spark**: Spark cleans its local scratch space inefficiently. In particular, shuffle data is not immediately cleaned up after a shuffle completes. This makes fault recovery more efficient, but results in higher storage requirements for scratch space. Another possible improvement for Spark on the XC40 would be to modify Spark to provide a more efficient cleaning strategy. This would make it feasible to fit the scratch data entirely in a local DRAM-based filesystem at larger problem sizes than currently possible.
- **Enable use of hierarchical scratch space directories in Spark**: Spark does not currently allow you to configure primary and backup scratch directories. Instead users must list all scratch directories in a single list, and Spark will distribute the data in a round round fashion between them as long as space is available. Another possible Spark improvement would be to allow users to provide a hierarchical listing of storage locations. For example, this would allow users to specify that

shuffle data should reside entirely in the /tmp ramdisk if possible, and only go out to Lustre if there is insufficient space in the /tmp filesystem.
- **Provide global directory awareness to Spark**: Spark does not allow you to specify that a scratch directory is globally accessible. Thus non-cached data is stored to the global Lustre file system by the sender, and then later retrieved by the sender and sent to the receiver. This wastes a step, since the receiver could easily fetch the data directly from the global file system.

In addition to the data movement issues described above, the pull-based, globally-synchronized stages model of computation and communication in Spark can create scaling issues in the presence of straggler tasks. When a small number of tasks take significantly longer to complete, many executor nodes end up stuck idling at the barrier. At larger node counts, this problem is exacerbated. To remediate this issue, we have tried two strategies, and propose investigating a third:

- **Turning on Spark speculation**: Spark contains an optional task speculation configuration parameter (spark.speculation). When this is enabled, Spark will attempt to re-execute tasks that take more than a configurable multiplier longer than the median task. We found that this helps in cases where we consistently see long stragglers pushing out the stage completion time. However, it does add overhead—and so for applications where stragglers only appeared occasionally our best times were obtained with speculation disabled.
- **Increasing the partitioning of datasets**: Increasing the partitioning of Spark RDDs and DataFrames allows for more effective load balancing, and can thus sometimes mitigate the impact of stragglers. However, increasing the partitioning too much may result in numerous small and inefficient tasks. The Spark scheduler also has limited throughput, and may become a bottleneck if too many partitions (and thus tasks) are created.
- **Enabling a push-based model of computation and communication**: One potential method to address the issue of stragglers in the future would be to enable an alternative push-based model of computation in Spark, where each map task directly sends its data to the reduce tasks. Reduce tasks would then be allowed to start whenever they receive data from any map task. Synchronization would still be required to ensure that no stage completes until it has received and processed all of its inputs—however this would allow additional overlapping of stages and potentially reduce time spent idling at barriers.

VIII. Conclusions

This paper demonstrated several techniques which can be used for analyzing the performance of data analytics frameworks on Cray XC systems. We looked at a bottom-up view of system metrics generated during HiBench runs, and a top-down view of Spark event logs generated during CX runs. We also looked at the TCP performance and the TCP traffic generated during CX and GraphX PageRank executions. We showed that with proper configuration choices, workloads that depend on the communication fabric can achieve performance boosts from the Aries interconnect. We believe that Spark will continue to be an important framework for future analytics applications, and as such suggested some techniques and modifications that could improve scaling on large systems.

References

[1] J. Bertolucci, "10 powerful facts about big data," *Information Week*, Jun. 2014. [Online]. Available: http://www.informationweek.com/big-data/big-data-analytics/10-powerful-facts-about-big-data/d/d-id/1269522

[2] *More Than 30 Billion Devices Will Wirelessly Connect to the Internet of Everything in 2020*, May 2013. [Online]. Available: https://www.abiresearch.com/press/more-than-30-billion-devices-will-wirelessly-conne/

[3] *Apache Spark—Lightning-fast Cluster Computing*, 2016 (accessed March 25, 2016). [Online]. Available: http://spark.apache.org/

[4] 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.

[5] *Welcome to Apache Hadoop!*, 2016 (accessed March 29, 2016). [Online]. Available: http://hadoop.apache.org

[6] *Spark Distributed Analytics Framework*, 2016 (accessed March 29, 2016). [Online]. Available: http://www.nersc.gov/users/data-analytics/data-analytics/spark-distributed-analytic-framework/

[7] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The HiBench benchmark suite: Characterization of the mapreduce-based data analysis," in *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, March 2010, pp. 41–51.

[8] *GitHub - intel-hadoop/HiBench: HiBench is a big data benchmark suite*, 2016 (accessed April 7, 2016). [Online]. Available: https://github.com/intel-hadoop/HiBench

[9] A. Gittens, J. Kottalam, J. Yang, M. Ringenburg, J. Chhugani, E. Racah, M. Singh, Y. Yao, C. Fischer, O. Ruebel, B. Bowen, N. Lewis, M. W. Mahoney, V. Krishnamurthy, and Prabhat, "A multi-platform evaluation of the randomized CX low-rank matrix factorization in Spark," in *5th International Workshop on Parallel and Distributed Computing for Large Scale Machine Learning and Big Data Analytics, at IPDPS*, 2016.

[10] 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.

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

[12] 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.

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

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

[15] K. Maschhoff and M. Ringenburg, "Experiences running and optimizing the Berkeley Data Analytics Stack on Cray platforms," in *Cray User Group Conference (CUG '15)*, Chicago, IL, 2015.

[16] N. Chaimov, A. Malony, S. Canon, K. Ibrahim, C. Iancu, and J. Srinivasan, "Scaling Spark on HPC systems," in *The 25th International ACM Symposium on High-Performance Parallel and Distributed Computing*, 2016, in publication.

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

[18] 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.

[19] *iperf3—iperf3 3.1.2 documentation*, 2016 (accessed April 6, 2016). [Online]. Available: http://software.es.net/iperf/index.html

[20] *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