Performance Evaluation of Apache Spark on Cray XC Systems

Nicholas Chaimov
Allen Malony
University of Oregon
Eugene, Oregon, USA
{*nchaimov,malony*}*@cs.uoregon.edu*

Shane Canon
Costin Iancu
Khaled Z. Ibrahim
Jay Srinivasan
Lawrence Berkeley National Laboratory
Berkeley, California, USA
{*scanon,cciancu,kzibrahim,jsrinivasan*}*@lbl.gov*

***Abstract*—We report our experiences in porting and tuning the Apache Spark data analytics framework on the Cray XC30 (Edison) and XC40 (Cori) systems, installed at NERSC. Spark has been designed for cloud environments where local disk I/O is cheap and performance is constrained by the network latency. In large HPC systems diskless nodes are connected by fast networks: without careful tuning Spark execution is dominated by I/O performance. In default mode the centralized storage system, such as Lustre, results in metadata access latency being a major bottleneck that severely constrains scalability. We show how to mitigate this by using per-node loopback filesystems for temporary storage. With this technique, we reduce the communication (data shuffle) time by multiple orders of magnitude and improve the application scalability from $O(100)$ to $O(10,000)$ cores on Cori. With this configuration Spark's execution becomes again network dominated. This reflects in the performance comparison with a cluster with fast local SSDs, specifically designed for data intensive workloads. Due to slightly faster processor and better network, Cori provides performance better by an average of 13.7% for the machine learning benchmark suite. This is the first such result where HPC systems outperform systems designed for data intensive workloads. Overall, we believe this paper demonstrates that local disks are not necessary for good performance on data analytics workloads.**

***Keywords*-Spark; Berkeley Data Analytics Stack; Cray XC; Lustre; Shifter**

I. Introduction

Apache Spark [1] is a data analytics framework which provides high-level constructs for expressing computations over datasets larger than the system physical memory. The runtime provides elastic parallelism, i.e. resources could grow or shrink without requiring any change to application code, and provides resilient execution, which allows automatic recovering from resource failures.

Spark is part of the Berkeley Data Analytics Stack [2], which includes storage, resource management and scheduling infrastructure, such as the Hadoop Distributed File System (HDFS) [3] and the Hadoop YARN resouce scheduler [4]. High-level application-domain libraries are built on top of spark, such as GraphX for graph analytics [5], Spark SQL for database queries [6], MLLib for machine learning [7], and Spark Streaming for online data processing [8].

Spark targets directly cloud or commodity clusters compute environments, which have latency-optimized local disk storage and bandwidth-optimized network, relatively few cores per node, and possibly little memory per node. HPC systems such as the Cray XC series, in contrast, feature diskless compute nodes with access to a high bandwidth global filesystem, large core counts, large memory sizes per compute node, and latency-optimized networks designed for use with HPC tightly coupled applications. The question remains if design decisions made for cloud environments translate well when running Spark on HPC systems and whether the latter can bring any value to analytics workloads due to their superior and tightly integrated hardware. In this paper, we present a comparative performance analysis of Spark running on Cray XC HPC systems and on a system (Comet) designed for data intensive workloads with large local SSDs.

We discuss the design of the Spark runtime, showing where file I/O occurs and what file access patterns are commonly used. On Cori, a Cray XC40 system installed at NERSC [9], we show that the use of the Lustre filesystem to store intermediate data can lead to substantial performance degradation as a result of expensive metadata operations. Initial Spark scalability is limited to $O(100)$ cores. To reduce I/O impact we extend Shifter, a lightweight container infrastructure for Cray systems, to mount a per-node loopback filesystem backed by Lustre files. This reduces the impact of the metadata operations by many orders of magnitude. With loopback, single node Spark performance on Cray XC improves by $6\times$ and it becomes comparable to that of single Comet node with SSDs. Even more exciting, loopback allows us to scale out to $O(10,000)$ cores and we observe orders of magnitude improvements at scale.

After calibrating and obtaining equivalent node performance on the Cray and Comet, we can compare the performance across the two system architectures. While the CPUs are roughly equivalent, Comet's nodes are connected with InfiniBand and the Cray uses Aries [10]. When comparing InfiniBand and Aries, the former exhibits higher latency and lower bandwidth. We use the `spark-perf` benchmark suite, consisting of a set of core RDD benchmarks and a set

of machine learning algorithm benchmarks using `MLLib`.

On both systems, the default Spark configuration uses TCP/IP over the native transport. In this configuration, the Cray XC is 55% faster than Comet at 16 nodes, which we attribute to the better network and natively optimized TCP/IP stack. TCP/IP overhead has been recognized as unnecessary overhead in HPC systems and optimized Spark implementations use RDMA communication.

Overall, these results are very encouraging. Simple configuration choices make HPC systems outperform architecures specifically designed for data analytics workloads with local SSDs: a global file system that provides a global name space can provide good performance. This indicates current system HPC designs are good to execute both scientific and data intensive workloads. The performance differences between the Cray XC and Comet may provide incentive for the aquisition of HPC systems in the "commercial" domain.

II. Spark Architecture

Spark implements the Map/Reduce model of computation. From the application developer's perspective, Spark programs manipulate *resilient distributed datasets* [11] (RDD), which are distributed lists of key-value pairs. The developer constructs RDDs from input data by reading files or parallelizing existing Scala or Python lists, and subsequently produces derived RDDs by applying *transformations* and *actions*. Transformations, such as `map` and `filter`, declare the kind of computation that could occur, but does not actually trigger computation; rather, a *lineage* is constructed, showing how the data represented by an RDD can be computed, when the data is actually required. *Actions* actually retrieve values from an RDD, and trigger the deferred computation to occur. Figure 1 shows a simple example of such a chain of transformations terminated by an action triggering computation.

Internally, RDDs are divided into *partitions*, which in turn are divided into *blocks*. All blocks within a partition are coresident on a particular node, and computation occurs at the partition level. When a partition is to be computed, the runtime produces and schedules a *task* which is submitted to the scheduling infrastructure. A task is assigned to a core and the number of partitions during execution is usually a small multiple of the number of cores.

The system consists of a *driver* running the application code, which constructs RDDs and submits tasks to compute RDD partitions, and one or more *executors*, where tasks run. The runtime automatically manages storage for blocks, through a per-executor BlockManager. The BlockManagers are responsible for servicing requests for blocks.

When a block is requested which is part of a partition owned by the current executor, the BlockManager checks to see if the block has already been computed and is cached in memory; if so, it is immediately returned. If not, the BlockManager retrieves a persisted block from disk, if available. If the block is neither resident in memory nor on disk, the BlockManager triggers computation of the block, which may recursively trigger additional computations on blocks earlier in the lineage chain. When a block finishes computation, it is optionally cached in memory. The BlockManager has a fixed size memory buffer allocated to it. If a block is to be cached in memory but not enough memory is available, the least-recently-used block is evicted from memory and either stored to the disk or dropped entirely.

During *map* tasks, all data dependencies are intra-partition. During a *reduce* task, inter-partition dependencies can occur, and it is only during reduce tasks that inter-node communication occurs. This occurs through a *shuffle*. During a shuffle, the ShuffleManager sorts data within each partition by key, and the key-value pairs within each partition are written to per-partition shuffle data files on disk. Each executor then submits requests for blocks, which are either local or remote. Each node then requests blocks, both locally and from other executors. When a block is requested which is owned by a remote executor, the local BlockManager makes a remote request to the owning BlockManager, which maintains a queue of requests which are serviced once the shuffle data is written to the corresponding shuffle file. The runtime storage infrastructure is shown in Figure 2.

III. Disk I/O Patterns in Spark

Disk I/O can occur in almost every stage of a Spark application. Figure 3 shows a lineage in which each RDD is annotated with the type of Disk I/O that can occur during its computation.

During the construction of input stages, disk I/O occurs to read the input data. In a traditional Spark installation, the input data would be stored in an HDFS overlay built on top of local disks, while on the Cray XC input data is stored directly on the Lustre filesystem available to all compute nodes. Output data is similarly stored either in HDFS or in Lustre, depending on the installation. When data is stored outside of HDFS, there is one file per partition, so as a minimum, there must be at least as many file opens and file reads as there are partitions. Additionally, each file is accompanied by a checksum file used for verifying integrity, and an index file indicating file names and offsets for specific blocks. Simple file readers such as the text file reader perform two file opens per partition: one for the checksum file and one for the partition data file. More complex file format readers perform more file operations; for example, the Parquet compressed columnar storage format performs four file opens per partition: first the checksum is opened, the partition data file is opened and the checksum computed, and both closed. Each partition data file is then opened, the footer is read, and then the file is closed. Finally, the file is opened again, the remainder of the file is read before closing the file again.

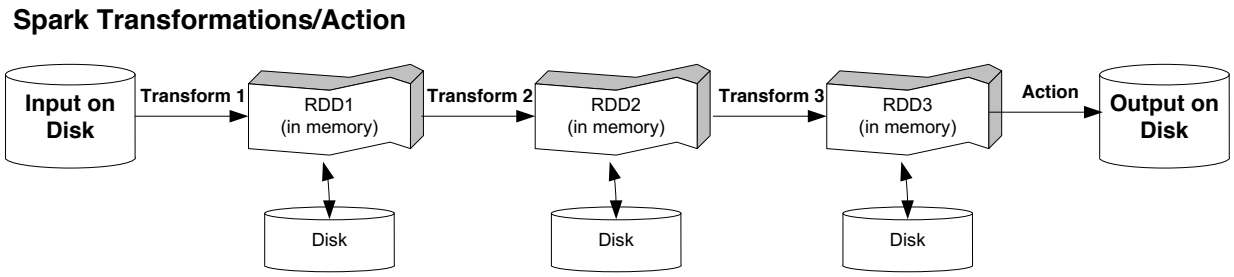

Figure 1. A chain of RDDs produced by applying transformations to an input dataset or predecessor RDDs. An action at the end of the chain triggers deferred computation.

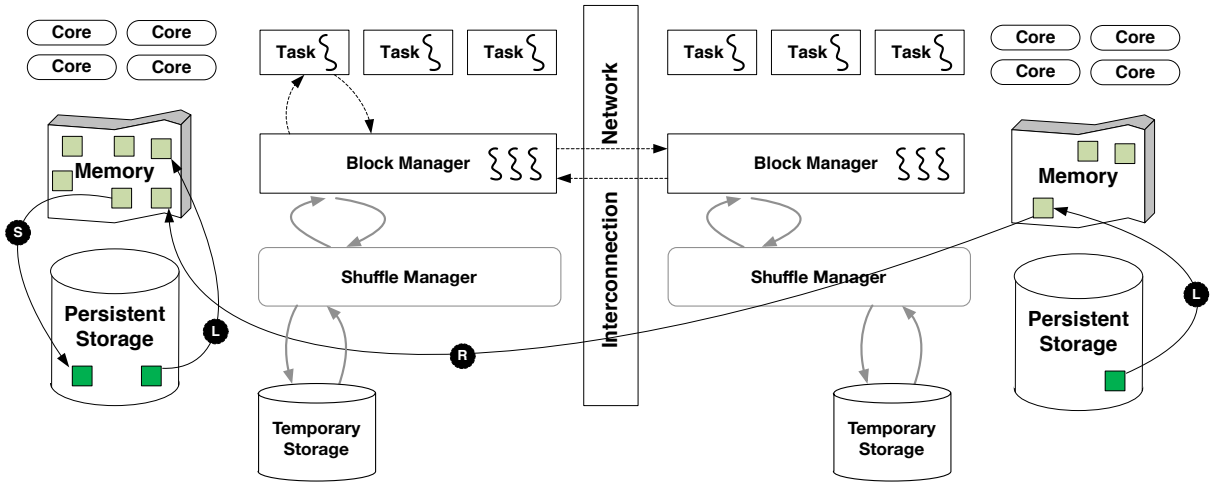

Figure 2. The storage infrastructure of Spark. Tasks request blocks from and store blocks to the BlockManager, which manages both memory and disk. The BlockManager communicates with the ShuffleManager to sort shuffle data and write shuffle data to temporary storage. Inter-node communication occurs through messages sent between BlockManagers.

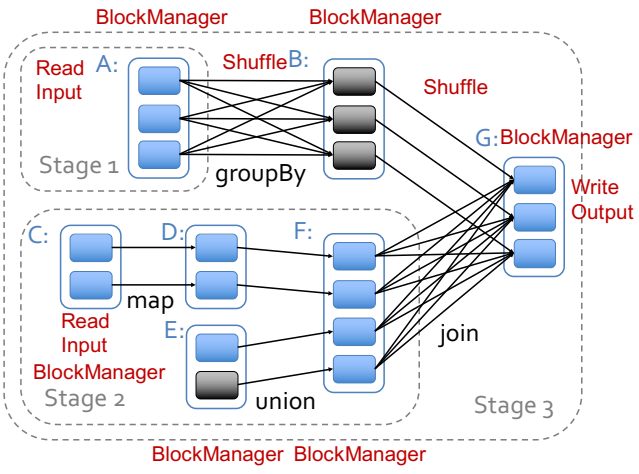

Figure 3. An example lineage showing the types of Disk I/O that can occur during each phase of the computation.

During every phase, BlockManager I/O can occur. If a block is requested while being stored on disk, the corresponding temporary file is opened and the data read and stored in memory, potentially triggering an eviction. When an eviction occurs, the corresponding temporary file is created, if necessary, and is written to. If sufficient memory is available that problem data fits in RAM, BlockManager disk I/O does not occur.

During shuffles, files are created storing sorted shuffle data. As originally designed, each shuffle map task would write an intermediate file for each shuffle reduce task, resulting in $O(tasks^2)$ files being written. The very large number of files produced tended to degrade performance by overwhelming the `inode` cache [12], so this was replaced with a single file per shuffle reduce task. However, as tasks are not supposed to affect the global state of the runtime except through the BlockManager, every map task writing to a per-reduce-task file opens the file, writes to it, and closes it. Similarly, every shuffle reduce task opens the file, reads from it, and closes it. Thus, although the total number of files has been reduced to $O(tasks)$, the number of *metadata*

operations remains $O(tasks^2)$. Shuffle intermediate files are *always* written regardless of the amount of memory available.

IV. Evaluation

We evaluate the `spark-perf` Core and `MLLib` benchmark suites on two systems, Cori and Comet. Cori [13], installed at NERSC, is a Cray XC40 consisting of 1,630 compute nodes, each with two 2.3 GHz 16-core Intel "Haswell" processors (Intel Xeon E5-2698 v3). Each node of Cori is equipped with 128 GB DDR4 2133Mhz MHz memory, and nodes are connected using a Cray Aries interconnect [10] based on the Dragonfly topology. A Lustre file system is used to provide storage to compute nodes on Cori. Comet [14], installed at SDSC, is Dell cluster consisting of 1,944 compute nodes, each with two 2.5 GHz 12-core Intel "Haswell" processors (Intel Xeon E5-2680 v3). Each node of Comet is equipped with 128 GB DDR4 DRAM. Nodes are connected using InfiniBand [15] FDR. A Lustre filesystem is provided, and additionally each node is equipped with a 320 GB SSD for fast scratch storage.

All of our evaluations are performed using Spark 1.5.0. On Cori, we have ported Spark to run under Extreme Scalability Mode (ESM) as described in [16]. On Comet, we use the preinstalled Spark 1.5.0 package launched using myHadoop [17].

On Cori we evaluate three Spark configurations: 1) using Lustre as backend storage; 2) using a `ramdisk` as backend storage; and 3) using as backend a mounted file system backed by a single Lustre file. On Comet we evaluate two configurations: 1) using Lustre as the backend storage; and 2) using the local SSDs.

V. Spark Performance on Lustre

Previous work on porting Spark to the Cray platform [18] running under Cluster Compatibility Mode revealed that performance of TeraSort and PageRank was up to four times worse on a 43 nodes of a Cray XC system compared to an experimental 43-node Cray Aries-based system with local SSDs, even though the experimental system had fewer cores than the Cray XC (1,032 vs 1,376). To mitigate this problem, the authors redirected shuffle intermediate files to an in-memory filesystem, but noted that this limited the size of problem that could be solved, and that the entire Spark job fails if the in-memory filesystem becomes full. Multiple shuffle storage directories can be specified, one using the in-memory filesystem and one using the Lustre scratch filesystem, but the Spark runtime then uses them in a round-robin manner, so performance is still degraded.

On Cori we compare directly Lustre with in-memory execution performance. On Comet we compare Lustre with SSD storage. To illustrate the main differences we use the GroupBy benchmark which is a worst-case shuffle. GroupBy generates key-value pairs with a limited number of keys across many partitions, and then groups all values associated with a particular key into one partition. This requires all-to-all communication, and thus maximizes the number of shuffle file operations required, as described in Section III, above.

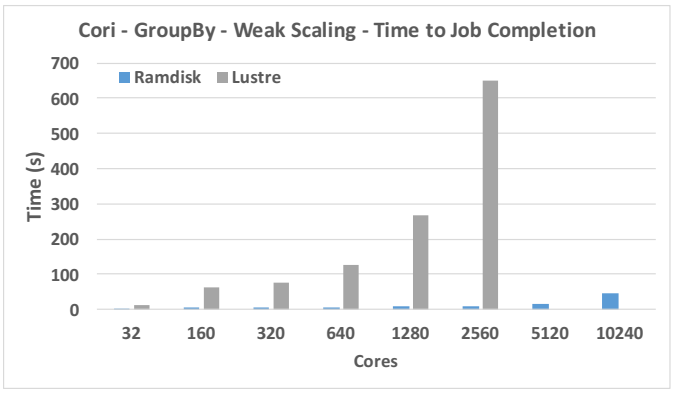

Figure 4. GroupBy benchmark performance (worst-case shuffle) on NERSC Cori, with shuffle intermediate files stored on Lustre or RAMdisk. Number of partitions in each case is $4 \times cores$

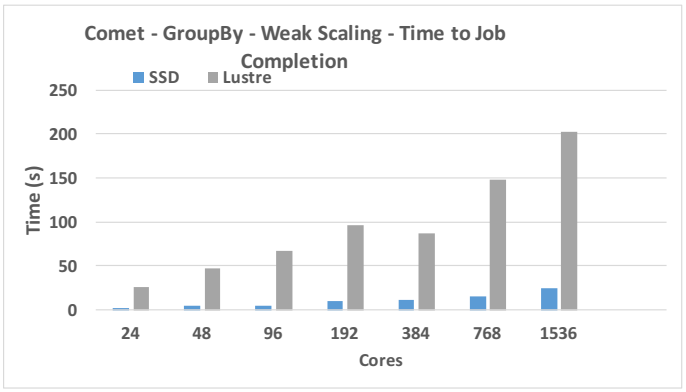

Figure 5. GroupBy benchmark performance (worst-case shuffle) on SDSC Comet, with shuffle intermediate files stored on Lustre or local SSD. Number of partitions in each case is $4 \times cores$

Figure 4 shows the results on Cori. On a single node (32 cores), when shuffle intermediate files are stored on Lustre, time to job completion is 6 times longer than when shuffle intermediate files are stored on an in-memory filesystem. The performance degradation increases as nodes are added: at 80 nodes, performance is 61 times worse on Lustre than the in-memory filesystem. Runs larger than 80 nodes using Lustre fail.

Results on Comet are shown in Figure 5. On one node, shuffle performance is 11 times slower on Lustre than on the SSD; however, the performance penalty does not become worse as we add nodes. Because Comet compute nodes feature local SSDs, there is less contention for the Lustre metadata server, as other jobs running on the system tend to make use of the SSD for intermediate file storage.

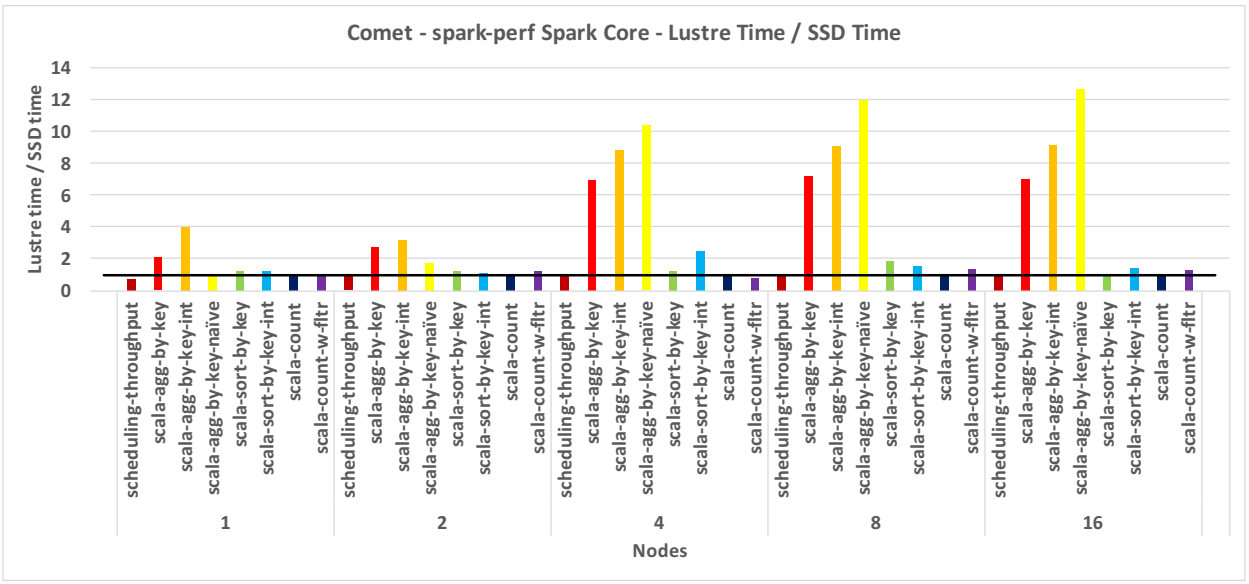

Figure 6. Slowdown of spark-perf Spark Core benchmarks on Comet with shuffle intermediate data stored on the Lustre filesystem instead of local SSDs.

Figure 6 shows the performance of the spark-perf benchmarks [19] on SDSC Comet. The *scheduling-throughput* benchmark runs a series of empty tasks without any disk I/O; its performance is unaffected by the choice of shuffle data directory. The *scala-agg-by-key*, *scala-agg-by-key-int* and *scala-agg-by-key-naive* benchmarks perform aggregation by key: they generate key-value pairs and then apply functions to all values associated with the same key throughout the RDD; this requires a shuffle to move data between partitions. The version using floating point values (*scala-agg-by-key*) and the integer version (*scala-agg-by-key-int*) are designed to shuffle the same number of bytes of data, so that the number of values in the integer version is larger than for the floating point version, increasing the number of shuffle intermediate file writes. The *scala-agg-by-key-naive* benchmark first performs a groupByKey, grouping all values for each key into one partition, before performing partition-local reductions, so that shuffles move a larger volume of data than for the non-naive versions, giving larger shuffle writes. The three *scala-agg-by-key* benchmarks have degraded performance when intermediate data is stored on Lustre, which continue to degrade as more nodes are added; at 16 nodes, performance for *scala-agg-by-key-naive* is 12 times worse than on SSD. The remaining benchmarks involve little or no shuffling and so are unaffected by shuffle directory placement.

As described in Section III, shuffle intermediate files are opened once for each read or write. When shuffle intermediate files are stored on Lustre, this causes heavy metadata server load which slows the overall process of reading or writing. Figure 7 shows the slowdown that results from opening a file, reading it, closing it, and repeating this process, as compared to opening a file once and performing multiple reads. For read sizes under one megabyte, Lustre filesystems show a penalty increasing with decreasing read size.

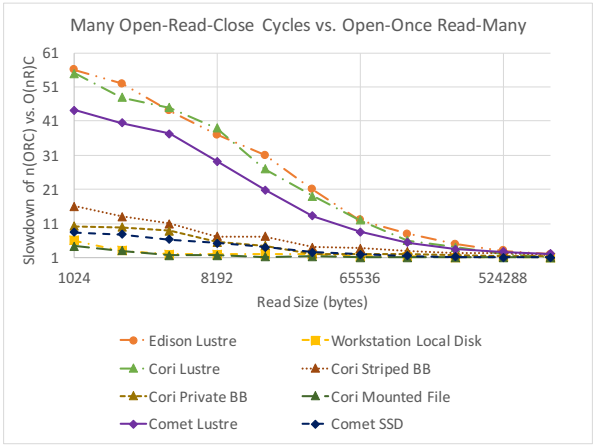

Figure 7. Slowdown from performing open-per-read rather than single-open many-reads for reads of different sizes on various filesystems on Edison, Cori, Comet, and a workstation with local disk. The penalty is highest for the Lustre filesystems.

Spark-perf also provides a set of machine learning benchmarks implemented using MLLib [7]. Figure 8 shows the slowdown of using Lustre storage instead of SSD for these benchmarks. Iterative algorithms – those which perform the same stages multiple times, and therefore have multiple rounds of shuffling – show the worst slowdown. The *lda* (Latent Dirichlet allocation), *pic* (power iteration clustering), summary statistics, *spearman* (Spearman rank correlation) and *prefix-span* (Prefix Span sequential pattern mining)

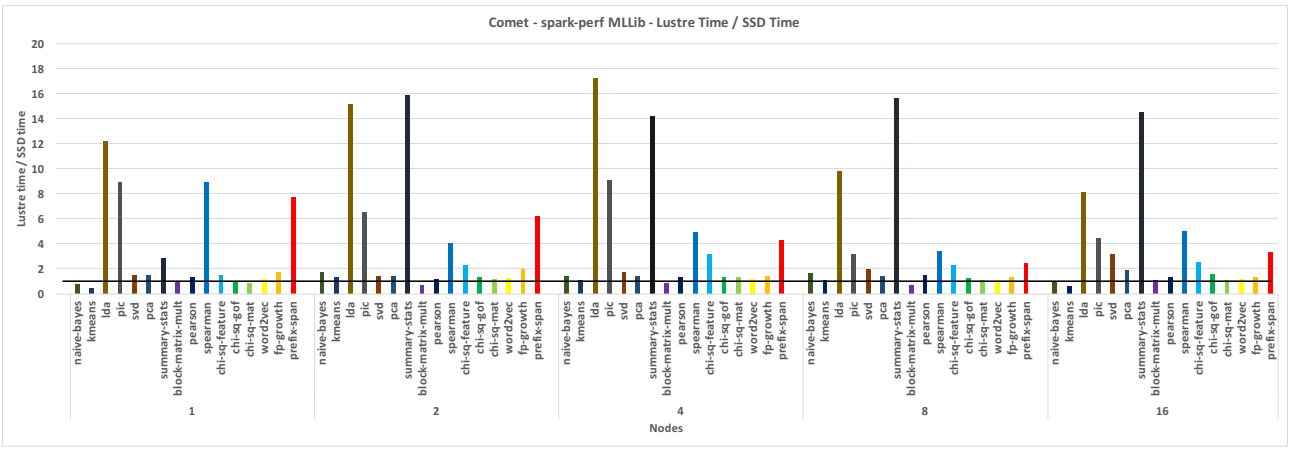

Figure 8. Slowdown of spark-perf MLLib benchmarks on Comet with shuffle intermediate data stored on the Lustre filesystem instead of local SSDs.

benchmarks all show subsantial slowdown when shuffle files are stored on Lustre rather than local SSDs. These are all iterative with the exception of the summary statistics benchmark, which has smaller block sizes than the other benchmarks.

These results demonstrate that shuffle performance is a major cause of performance degradation when local disk is not available or not used for shuffle-heavy applications.

VI. Localizing Metadata Operations with Shifter

To improve the file IO performance, ideally we need to avoid propagating metadata operations to the Lustre filesystem because these files are used solely by individual compute nodes. On Cray XC systems, we do not have access to local disk, and using in-memory filesystems limits the problem sizes. We have previously described a file-pooling technique [16] which maintains a pool of open file handles during shuffling to avoid repeated opens of the same file. However, this requires modifications to the Spark runtime, and affects only operations coming from the Spark runtime. Other sources of redundant opens, such as high-level libraries and third-party file format readers, are not addressed. Furthermore, each file must be opened at least once, still placing load on the Lustre metadata server, even though the files are only needed on one node.

To keep metadata operations local, we have previously experimented with mounting a per-node loopback filesystem, each backed by a file stored on Lustre. This enables storage larger than available through an in-memory filesystem while still keeping file opens of intermediate files local; only a single open operation per node must be sent to the Lustre metadata server, to open the backing file. This approach was not feasible, however, for ordinary use, as mounting a loopback filesystem requires root privileges.

Shifter [20] is a lightweight container infrastructure for the Cray environment that provides Docker-like functionality.

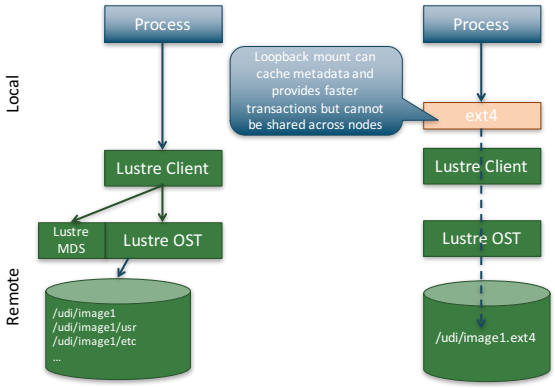

Figure 9. Shifter can mount node-local filesystems, keeping metadata operations local but preventing cross-node access.

With Shifter, the user can, when scheduling an interactive or batch job, specify a Docker image, which will be made available on each of the compute nodes. In order to do this, Shifter provides a mechanism for mounting the image, stored on Lustre, as a read-only loopback filesystem on each compute node within the job. Motivated by our work, Shifter was recently extended to optionally allow a per-compute-node image to be mounted as a read-write loopback filesystem, as shown in Figure 9.

Using mounted files eliminates the penalty for per-read opens, as shown in Figure 7. When we run the GroupBy benchmark on Cori with data stored in a per-node loopback filesystem, we vastly improve scaling behavior, and performance at 10,240 cores is only 1.6× slower than in-memory filesystem, as shown in Figure 10. Unlike with the in-memory filesystem, we can select the size of the per-node filesystem to be larger than the available memory, preventing job failure with large shuffles.

We have run the spark-perf benchmarks used in Section V to compare performance between Lustre and Lustre-backed

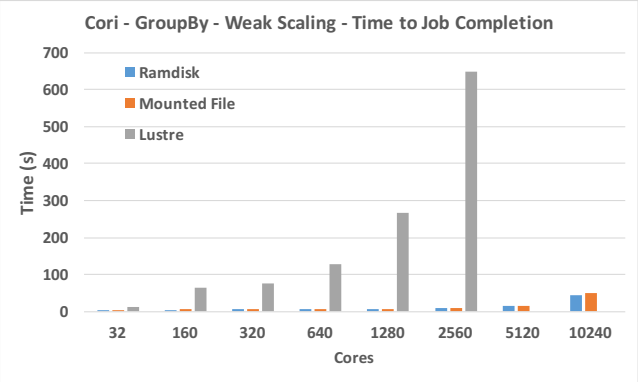

Figure 10. GroupBy benchmark performance (worst-case shuffle) on NERSC Cori, with shuffle intermediate files stored on Lustre, RAMdisk, or per-node loopback filesystems backed by Lustre files. Number of partitions in each case is $4 \times cores$

loopback file systems. Results for the Spark Core benchmarks are shown in Figure 11. Using per-node loopback filesystems improves performance at larger core counts for the *scala-agg-by-key* and *scala-agg-by-key-int* benchmarks, particularly for the latter which performs a larger number of opens. Results for the MLLib benchmarks are shown in Figure 12. The *lda*, *pic*, *spearman*, *chi-sq-feature* and *prefix-span* benchmarks show substantial improvement from the use of per-node loopback filesystems. Furthermore, they exhibit better scaling behavior on Cori than on Comet with local disk. Figure 13 shows weak-scaling performance with those benchmarks on Cori and Comet. Cori nodes provide more cores (32) than Comet nodes (24), although Comet nodes run at a higher clock speed (2.5GHz) than Cori nodes (2.3GHz).

VII. Conclusion

We have evaluated Apache Spark on Cray XC systems using a series of runtime microbenchmarks and machine learning algorithm benchmarks. As compute nodes on these systems are not configured with local disks, files created by the Spark runtime must be created either in an in-memory filesystem, limiting the size of data which can be shuffled, or created on the global scratch filesystem, which we have found to severely degrade performance, particularly as more nodes are used. On other systems, such as SDSC Comet, compute nodes have been equipped with local SSD storage for the purpose of storing temporary data during computation, which provides up to $11\times$ faster performance than using a Lustre filesystem for shuffle-intensive workloads. We have identified that the cause of the performance degradation is not read or write bandwidth but rather file metadata latency, and have used the Shifter container infrastructure installed on the Edison and Cori systems to mount per-node loopback filesystems backed by the Lustre scratch filesystem. This allows for increased performance and scalability, offering performance comparable to the use of local disks for shuffle-intensive workloads, without constraining the maximum problem size as with the in-memory filesystem. This technique is a promising approach for deploying Apache Spark on Cray XC systems.

Acknowledgment

This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1053575.

References

[1] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, vol. 10, p. 10. [Online]. Available: http://static.usenix.org/legacy/events/hotcloud10/tech/full_papers/Zaharia.pdf

[2] M. Franklin, "Making sense of big data with the berkeley data analytics stack," in *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*, ser. WSDM '15. New York, NY, USA: ACM, 2015, pp. 1–2. [Online]. Available: http://doi.acm.org/10.1145/2684822.2685326

[3] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–10.

[4] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache hadoop YARN: Yet another resource negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. ACM, pp. 5:1–5:16. [Online]. Available: http://doi.acm.org/10.1145/2523616.2523633

[5] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *Proceedings of OSDI*, pp. 599–613. [Online]. Available: https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-gonzalez.pdf

[6] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark SQL: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. ACM, pp. 1383–1394. [Online]. Available: http://doi.acm.org/10.1145/2723372.2742797

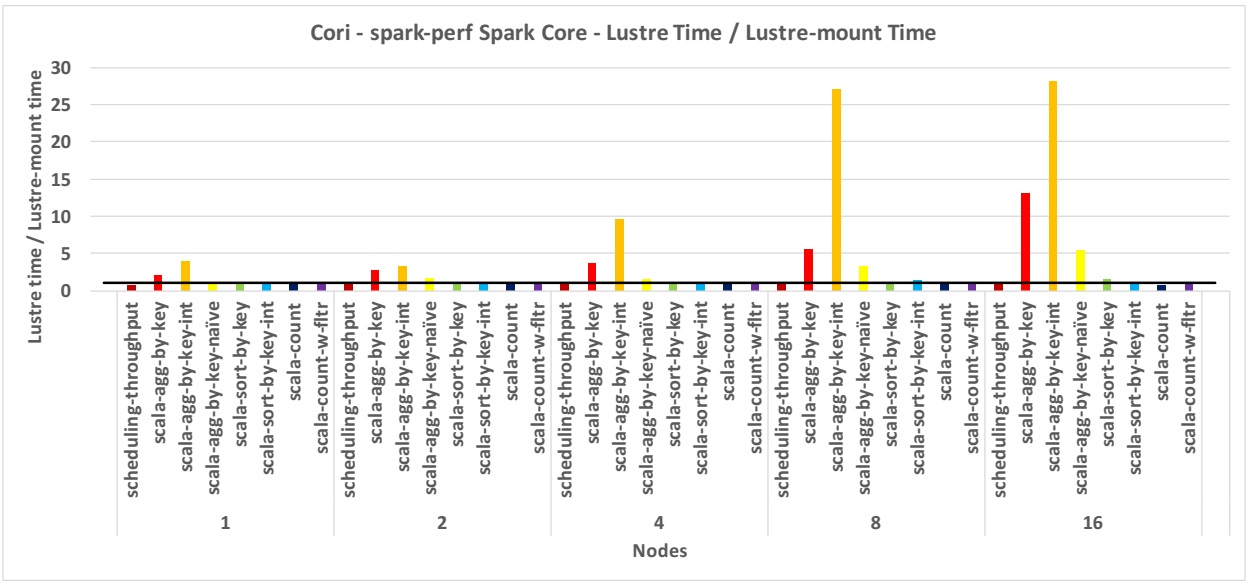

Figure 11. Slowdown of spark-perf Spark Core benchmarks on Cori with shuffle intermediate data stored on the Lustre filesystem instead of Lustre-backed loopback filesystems.

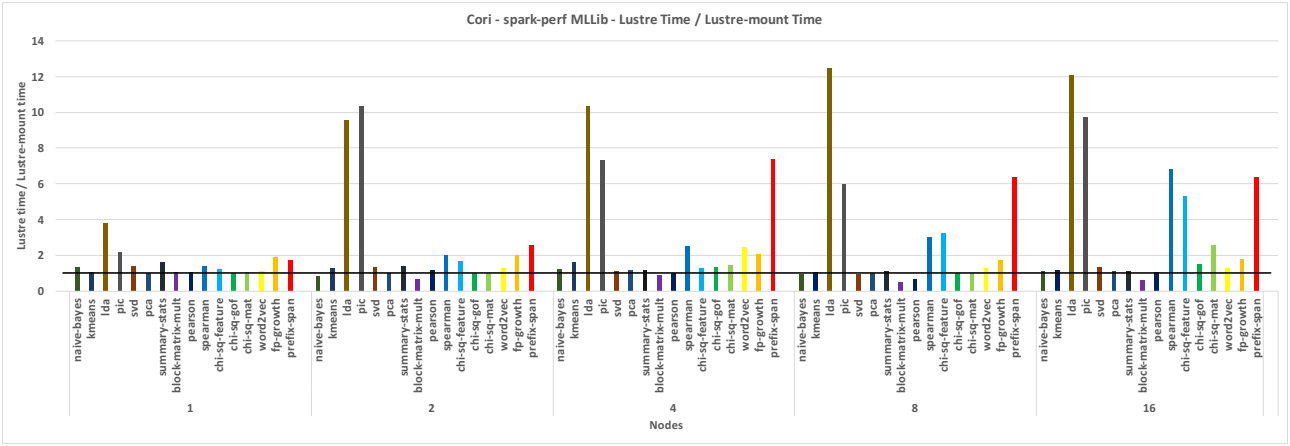

Figure 12. Slowdown of spark-perf MLLib benchmarks on Cori with shuffle intermediate data stored on the Lustre filesystem instead of Lustre-backed loopback filesystems.

[7] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. B. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar, "MLlib: Machine learning in apache spark." [Online]. Available: http://arxiv.org/abs/1505.06807

[8] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters," in *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*, ser. HotCloud'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=2342763.2342773

[9] "National Energy Research Scientific Computing Center," https://www.nersc.gov.

[10] B. Alverson, "Cray high speed networking," in *Proceedings of the 20th Annual Symposium on High-Performance Interconnects (HOTI)*, 2012.

[11] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12. USENIX Association, pp. 2–2. [Online]. Available: http://dl.acm.org/citation.cfm?id=2228298.2228301

[12] A. Davidson and A. Or, "Optimizing shuffle performance in spark." [Online]. Available: http://www.cs.berkeley.edu/~kubitron/courses/cs262a-F13/projects/reports/project16_report.pdf

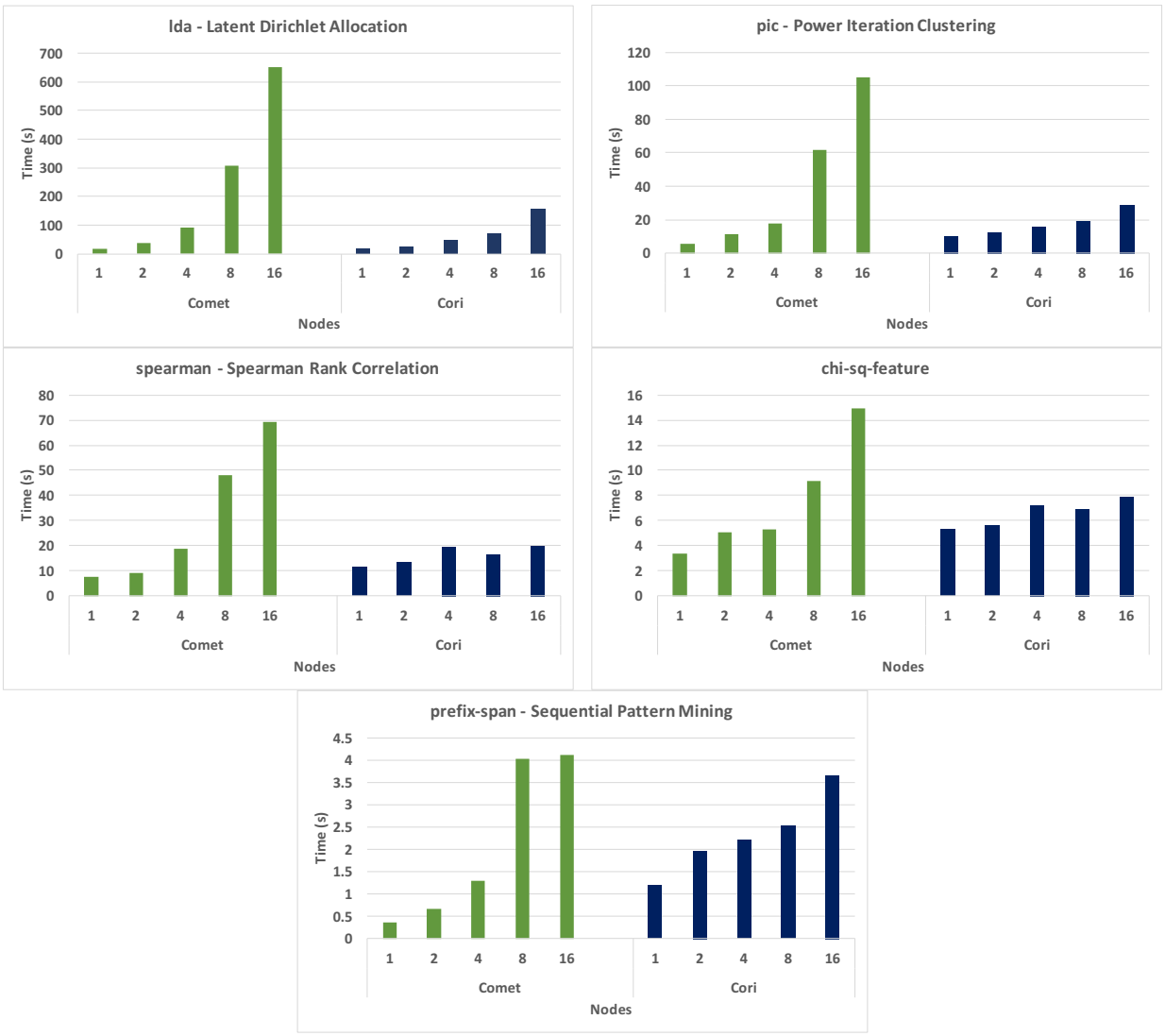

Figure 13. Weak scaling for the MLLib benchmarks most sensitive to shuffle performance on Cori with per-node loopback filesystems and on Comet with local SSDs.

[13] "Cori Phase 1," https://www.nersc.gov/users/computational-systems/cori/.

[14] "SDSC Comet," http://www.sdsc.edu/services/hpc/hpc_systems.html#comet.

[15] I. T. Association *et al.*, *InfiniBand Architecture Specification: Release 1.0*. InfiniBand Trade Association, 2000.

[16] N. Chaimov, A. Malony, S. Canon, C. Iancu, K. Z. Ibrahim, and J. Srinivasan, "Scaling spark on hpc systems," in *Proceedings of the International Conference on High-Performance Parallel and Distributed Computing*, 2015.

[17] S. Krishnan, M. Tatineni, and C. Baru, "myhadoop-hadoop-on-demand on traditional hpc resources," *San Diego Supercomputer Center Technical Report TR-2011-2, University of California, San Diego*, 2011.

[18] K. J. Maschhoff and M. F. Ringenburg, "Experiences running and optimizing the berkeley data analytics stack on cray platforms," in *Cray Users Group*, 2015.

[19] "spark-perf benchmark," https://github.com/databricks/spark-perf.

[20] D. M. Jacobsen and R. S. Canon, "Contain this, unleashing docker for hpc," in *Cray Users Group*, 2015.