Exploiting Spark for HPC Simulation Data: Taming the Ephemeral Data Explosion

Ming Jiang
Lawrence Livermore National
Laboratory
jiang4@llnl.gov

Brian Gallagher
Lawrence Livermore National
Laboratory
gallagher23@llnl.gov

Albert Chu
Lawrence Livermore National
Laboratory
chu11@llnl.gov

Ghaleb Abdulla
Lawrence Livermore National
Laboratory
abdulla1@llnl.gov

Timothy Bender
Lawrence Livermore National
Laboratory
bender10@llnl.gov

ABSTRACT

In this paper, we address the challenge of analyzing simulation data on HPC systems by using Apache Spark, which is a Big Data framework. One of the main problems we encountered with using Spark on HPC systems is the *ephemeral data explosion*, which is brought about by the *curse of persistence* in the Spark framework. Data persistence is essential in reducing I/O, but it comes at the cost of storage space. We show that in some cases, Spark scratch data can consume an order of magnitude more space than the input data being analyzed, leading to fatal out-of-disk errors. We investigate the real-world application of scaling machine learning algorithms to predict and analyze failures in multi-physics simulations on 76TB of data (over one trillion training examples). This problem is 2–3 orders of magnitude larger than prior work. Based on extensive experiments at scale, we provide several concrete recommendations as state-of-the-practice, and demonstrate a 7x reduction in disk utilization with negligible increases or even decreases in runtime.

KEYWORDS

Apache Spark, machine learning, ALE simulation, HPC system

ACM Reference Format:

Ming Jiang, Brian Gallagher, Albert Chu, Ghaleb Abdulla, and Timothy Bender. 2020. Exploiting Spark for HPC Simulation Data: Taming the Ephemeral Data Explosion. In *International Conference on High Performance Computing in Asia-Pacific Region (HPCAsia2020)*, January 15–17, 2020, Fukuoka, Japan. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3368474.3368482>

1 INTRODUCTION

Managing and analyzing simulation data has always been challenging. Large-scale simulations have the capacity to produce massive amounts of data that HPC architectures are not designed to analyze effectively. Furthermore, simulation data must be analyzed in

place since movement between systems is prohibitively expensive. The introduction of MapReduce and Big Data analytics spurred interest in adapting these tools to HPC environments [22]. Interest in combining Big Data and HPC [10, 32, 38] increased with the development of Apache Spark’s feature-rich library for data analysis and machine learning (ML) [7]. With the rise in popularity of using ML to analyze simulation data and automate scientific workflows [14, 15, 20, 21, 23, 24], there is growing interest in exploiting Spark for HPC simulation data. Recent work on using Spark varies from implementing traditional analysis pipelines [26, 33, 34] to developing APIs to speed up data access [17, 25, 37].

However, several challenges need to be addressed to bring Big Data approaches to the HPC environment. Big Data clusters typically have a large amount of node-local storage. HPC environments often have little or no node-local storage, instead they rely on off-loading data to large distributed network storage. Using network storage can degrade the performance of traditional Big Data workflows [12, 38]. In addition, Big Data clusters have tended towards larger memory per node, a system design optimization that is not necessarily beneficial to HPC workflows.

While an optimal solution would be to create a dedicated Big Data cluster to analyze simulation data in place, HPC users are constrained by real-world HPC environments that do not typically provide such resources. Therefore, in this paper, we focus on how to effectively exploit Spark for simulation data residing on HPC systems, which has not been adequately addressed in the current literature. Specifically, we present *best practices* based on lessons learned from our experience of using Spark for performing distributed ML on massive simulation data (2–3 orders of magnitude larger than previous work). In order to make our *best practice* recommendations as broadly applicable as possible, we focus on user-configurable options rather than customizations to Spark itself. The advantage of this approach is that our recommendations are general for HPC systems and applicable across different Spark versions and configurations.

The main problem we encountered when using Spark on HPC systems is what we refer to as the *ephemeral data explosion*, brought about by the *curse of persistence*. For multi-pass Spark computations, data persistence is essential in reducing I/O, but it comes at a cost in terms of storage space. We show that in some cases (e.g., the popular random forest (RF) algorithm), Spark scratch data can consume an order of magnitude more space than the original input data being

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

HPCAsia2020, January 15–17, 2020, Fukuoka, Japan

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-7236-7/20/01...\$15.00

<https://doi.org/10.1145/3368474.3368482>

Figure 1: Bubble Shock simulation ensemble for training Random Forest regression model for failure prediction.

analyzed. Thus, there is a fundamental trade off between space and time: it takes time to read data from storage, and it takes space to cache data locally.

The main contribution of this paper is a set of *best practice* recommendations for using Spark to perform ML on HPC simulation data. What makes this work novel is that our *best practices* are in direct contrast to prevailing recommendations in both the official Spark documentation [7] and the current literature (see Section 7). In order to justify these contrarian recommendations, we present results from an extensive set of experiments on massive simulation data. Our experiments employ a real-world application of training ML algorithms to predict and analyze failures in multi-physics simulations, and they culminate with the demonstration of training a RF regression model on 76TB of data, with over one trillion training examples. Although the primary goal of this paper is to provide concrete recommendations for users who are interested in using Spark for their own applications, we also present novel ML results for our specific application to demonstrate the kinds of experiments enabled by the recommendations described in this paper.

2 BACKGROUND

2.1 Machine Learning for ALE Simulations

The Arbitrary Lagrangian-Eulerian (ALE) method is used in a variety of engineering and scientific applications for enabling multi-physics simulations. Unfortunately, the ALE method can suffer from simulation failures that require users to adjust parameters iteratively in order to complete a simulation. For a given problem, identifying which parameters to adjust and by how much is as much art as science. Often, finding the right ALE strategy requires many iterations (simulation fails, adjust parameters, and rollback) that can be disruptive and time consuming.

For ALE simulations, there are two general ways that failures can occur: 1) mesh geometry becomes distorted or 2) physical quantities become distorted. Recently, Jiang et al. [20] developed a supervised machine learning framework for predicting conditions leading to mesh tangling failures using a RF regression algorithm. For testing and evaluation, [20] used the Bubble Shock simulation, which models a planar shock traveling through a spherical helium bubble suspended in air from the right side of the geometry. The Bubble Shock simulation is important for testing the development of ALE

codes that are vital to Stockpile Stewardship Science. Figure 1a shows a pseudocolor visualization of the materials at the start of the simulation using a blue-orange colormap.

To generate the necessary training data, they developed a failure generation procedure and applied it to an ensemble of Bubble Shock simulations by varying the initial conditions of piston speed, which is a measure for the shock strength, and bubble density, which can produce varying degrees of deformation in the bubble mesh geometry. Figure 1b provides visualizations of the deformed bubble mesh based on the initial conditions from the ensemble.

The dataset generated from this ensemble of 64 simulation variations consists of 12,068 simulation runs with a grand total of 90,328,241 simulation cycles. Since the learning instance used to train the RF model is a mesh element at a given timestep, and there are 11,560 elements in the Bubble Shock mesh, this results in 1,044,194,465,960 learning instances, which constitutes a 76TB training dataset! Figure 1c provides a heat map of the data size distribution amongst the simulation variations. Training ML models on 76TB of data is extremely challenging. In fact, 76TB is on the same scale as the current sort benchmark challenge of 100TB [6]. In the published literature, ML scaling experiments on HPC systems were conducted in the range of hundreds of gigabytes [18, 27, 35, 36].

2.2 Big Data Analytics via Spark

Apache Spark provides an attractive solution to this challenge since it is inherently distributed and includes a built-in RF implementation. Spark is a popular distributed computing framework used predominantly for data analytics. It includes rich APIs for data manipulation, ML, and querying of structured data. The foundation of Spark is built on the resilient distributed dataset (RDD) representation for parallel operations and fault tolerance. Spark’s MLlib includes a robust set of common ML algorithms using the RDD API. Structured data processing is provided by the Spark SQL API and the DataFrame representation. Spark ML is the package of ML algorithms designed to work with the DataFrame API.

Spark provides a rich API to perform *transformations* and *actions* on RDDs, such as *map*, *filter*, and *reduce*. The *transformations* and *actions* in a user job are processed by Spark into an execution plan with *stages*. Each *stage* is a unit of work within a Spark job, each separated by a shuffle operation on the data. Every *stage* contains

Figure 2: The Ephemeral Data Explosion: training RF on 2TB of data prematurely terminates due to exceeding available storage on a 32-node allocation (over 23TB total).

tasks that will be executed in parallel across worker nodes. The number of *tasks* is dependent on the number of *partitions* of data that exist, which can be configured using *block size*.

One of the most important features of RDDs is that they can be cached (or “persisted”) in memory, disk, or memory & disk, the latter overflowing to disk if necessary. This allows developers to cache data for repeated access, providing the ability to optimize their code for the common case. When caching to disk or memory & disk, care must be made to not overflow the available disk space.

Processing of RDDs creates *intermediate data* that may need to be shuffled between nodes. This is costly when *intermediate data* exceeds memory limits and needs to be stored on disk. Several papers [38] [12] have discussed how storing this data on local storage instead of networked storage can improve performance. The disk storage location of cached RDDs and intermediate data is specified by Spark’s *spark.local.dir*. This is often referred to as Spark’s “scratch space.”

3 THE EPHEMERAL DATA EXPLOSION

An important conclusion from [20] was that RF model performance improves by training on data from as many simulations as possible. However, since the prior work relied on a single-node shared-memory algorithm from the scikit-learn library [30], there is a hard limit on the amount of data that can be used to train an RF model. In this paper, we are interested in training ML models on the full 76TB data by using Spark. Specifically, we chose to implement and evaluate two standard ML analyses: 1) RF for supervised learning of failure prediction and 2) k-means clustering for unsupervised analysis of failure modes.

In addition to providing insights into the problem of analyzing and predicting failures in ALE simulations, these two ML analyses also illustrate two typical usages of Spark: 1) calling a single, complex function to get direct results, and 2) composing a set of lightweight functions to perform high-level analysis. For training the RF regression models, we simply called:

```
pyspark.mllib.tree.RandomForest.trainRegressor (1)
```

with parameters: numTrees=100, maxDepth=4, and maxBins=32.

For failure mode analysis, we developed a Spark processing pipeline as follows. First, the RDD of learning features is converted

Figure 3: Training RF on 32 nodes for 1TB of input data, Spark produces over 12TB of scratch files, dominated by cached RDDs.

into a DataFrame by adding a schema. Learning features are then assembled into a vector and standardized using:

```
pyspark.ml.feature.StandardScaler (2)
```

Clustering is performed using the Spark ML implementation of k-means:

```
pyspark.ml.clustering.KMeans (3)
```

For failure mode analysis, we chose k=2 to distinguish failures from non-failures. Finally, the cluster quality is evaluated by using:

```
pyspark.ml.evaluation.ClusteringEvaluator (4)
```

The evaluation produces a silhouette score to indicate the cohesiveness of the cluster members and separation from other clusters.

Our immediate observations regarding training RF in Spark are: a) Spark does, in fact, scale to much larger data sizes than a single-node, shared-memory strategy; however, b) scaling is limited by unexpectedly high disk utilization. In particular, we were surprised to observe that Spark’s scratch data for RF runs were an order of magnitude larger than the input data size. We refer to this blowup as the *ephemeral data explosion*.

After some investigation, we identified several sources of this explosion: 1) our original data is stored as single precision floats, but Spark RF represents everything as double precision, 2) the data RDD is persisted once in the PySpark RF implementation¹ and then a transformed data RDD is persisted in the underlying Scala implementation², and 3) we observed a lot of data serialization overhead in the persisted RDD data.

This highlights a fundamental issue with iterative computations in Spark, which we refer to as “the curse of persistence.” For iterative computations (or any computation requiring multiple passes over the data), data persistence is essential. Without it, data needs to be read from storage over and over, which can bog down the entire computation. The “blessing of persistence” is that it eliminates redundant I/O. The curse of persistence is the *ephemeral data explosion*. There is a fundamental trade off here between space and time: it takes time to read data from storage, and it takes space to cache data locally. In the following sections, we investigate this data explosion empirically and explore ways to mitigate it.

¹src/main/scala/org/apache/spark/mllib/api/python/PythonMLLibAPL.scala

²mllib/src/main/scala/org/apache/spark/ml/tree/impl/RandomForest.scala

3.1 The X-Factor

Both the official Spark documentation [7] and recent studies [12, 35, 38] strongly recommend the use of node-local storage (such as SSD) for storing intermediate data. In addition, disabling compression on RDDs is recommended to improve performance [35]. These state-of-the-practice guidelines are motivated by runtime considerations and our experiments confirm that the combination of uncompressed data and node-local storage can produce the best runtime performance when all the data fits.

However, in HPC environments, users are constrained in their ability to add compute nodes or node-local storage to a system to accommodate larger problems. Inevitably, larger jobs will run out of local disk space, which results in a fatal out-of-disk error and premature termination. So, while uncompressed local RDDs can produce optimal performance in an optimal environment, in practice, we need to carefully consider the relevant time/space tradeoffs in HPC environments.

While experimenting with Spark for our application, we observed that disk space, rather than compute time, is often the most limiting factor. For example, while attempting to train a RF model on a modest input data size of 2TB, we exceeded available local storage (23TB total) on a 32-node allocation. Figure 2 shows the average SSD utilization (error bars indicate min and max) across 32 nodes for three training data sizes: 0.5TB, 1TB and 2TB. For the 2TB run, at least one node used up all available local disk (745GB) and triggered a fatal out-of-disk error.

The obvious question is: what is all this disk space being used for? Figure 3 provides an answer. Using the successful 1TB run described above, we counted the number of persisted RDD files and intermediate shuffle files and computed their total file sizes. From the figure, we can see that although the number of RDD and shuffle files are within the same range, their total files sizes differ by three orders of magnitude. In this case, the total files sizes are consistent with the sum of the disk utilization we observed, which is how we compute the *x-factor*.

$$x_factor = \frac{Spark_disk_utilization}{original_data_size} \quad (5)$$

For the 1TB run, the total disk utilization is 12.6TB; hence, the *x-factor* is 12.6TB/1TB=12.6x!

The 12.6TB of scratch data is dominated by RDDs, which are created and persisted in the internals of the MLlib library that the user has little control over. The obvious user-level options to reduce disk utilization are: 1) shrink RDD size through compression and/or 2) persist RDDs to the much larger network storage available in an HPC environment. Both of these options go against what is recommended in the state-of-the-practice. Figure 4 shows the results of applying these two options for training RF on 32 nodes. For runtime, Figure 4a confirms that using compression can incur cost up to 10%. The orange bar is slightly higher than the blue bar, except for the 2048 run, which suffered the fatal out-of-disk error that caused it to terminate prematurely. For *x-factor*, Figure 4b shows a huge benefit to using both compression and network storage in terms of space requirements (up to 70% reduction). The question we address in this paper is how to achieve the best of both worlds, by minimizing both storage requirements and runtime.

Figure 4: Training RF on 32 nodes on 1) node-local SSD with no RDD compression, 2) node-local SSD with RDD compression, and 3) Lustre network storage with no RDD compression. Spark RDD compression is an effective way to reduce disk utilization (i.e., the *x-factor*).

To this end, we propose the following set of specific configurations based on an extensive evaluation: 1) enable RDD compression, but use ZStd (level-1) rather than the default LZ4, 2) utilize node-local storage as much as possible, but fall back on network storage as necessary to handle overflow, and 3) increase Spark default block size to improve network storage I/O performance. In this paper, we demonstrate that these configurations allow us to scale to arbitrarily large data sizes, while optimizing runtime for any given problem size. For convenience, we refer to these configurations collectively as TEDX: Taming the Ephemeral Data eXplosion.

4 EXPERIMENTAL SETUP

4.1 Test Environment

We conducted our experiments on the Catalyst cluster at Lawrence Livermore National Laboratory (LLNL). Catalyst is a 324 node production cluster deployed in 2013. Each Catalyst node has two 12-core Intel Xeon E5-2694v2 processors, 128GB RAM, and 745GB SSD storage. The cluster scheduler/resource manager is Slurm, the interconnect is QDR Infiniband, and Lustre version 2.8 is the parallel file system. The Lustre file system is backed by 36 ZFS based OSTs and enables LZ4 transparent compression. Testing on Catalyst allows us to evaluate and compare Spark’s performance using both node-local and network storage.

Lustre is commonly used in HPC environments to store simulation data. An important part of our experiments is that all input data are stored on Lustre and read in from Lustre. Other papers

have discussed results testing on HDFS [18] or even RAMDisk [38]. A significant number of HPC systems do not have node-local storage suitable for Big Data analytics jobs. Similarly, RAMDisk may be fast, but severely limits the size of data and further exacerbates issues with Spark cache overflowing to disk. For the Bubble Shock simulations, the average data file size is ~400MB. Likewise, the average RDD file size is ~150MB. Based on these considerations, we used Lustre with the default stripe count of 1 and stripe size of 1M.

4.2 Magpie: Enabling Spark on HPC

Magpie [5] contains a number of scripts for running Big Data analytics on HPC systems. It supports many Big Data frameworks, including Hadoop [3] and Spark [7]. Several schedulers and resource managers are supported, including Slurm, Moab, Torque, and LSF. Magpie functions without significant modification to any of the projects it supports, making it easy to experiment with different project versions. Although there are other frameworks for running Big Data on HPC systems [4, 8, 10, 22], we chose Magpie because it has been tested on over 50 versions of Spark and it is actively maintained in our test environment. Magpie has been open sourced onto GitHub where it is actively supported and has contributions from multiple organizations.

Magpie provides a set of job submission scripts that may be customized with user configurations. Users must configure basics, such as the number of nodes to run on and paths to scripts they wish to run. More advanced configurations allow for performance tuning. Magpie configures a reasonable set of defaults for the hardware the user is running on, such as the number of CPUs or memory available in the cluster allocation. It allows users to store data to Hadoop Distributed File System (HDFS), an HDFS setup over network storage (such as Lustre), or directly to network storage. It also allows users to take advantage of any node-local storage (such as SSD) that may exist. Magpie generally runs master daemons on a single node of the allocation and workers on all remaining nodes. Anytime this paper mentions the use of N nodes, the master node is not included in this count and the reader can assume $N+1$ nodes were actually allocated.

4.3 Spark Setup & Configuration

Our experiments were run with Spark 2.3.0. Magpie supports running Spark with either the Hadoop Yarn scheduler or the Spark Standalone scheduler. For our experiments, we used the Standalone scheduler, which is a simple FIFO scheduler with limited capabilities. Since a Slurm allocation is dedicated to a single user and our experiments run only a single experiment at a time, the Standalone scheduler is more than sufficient for these experiments.

Unless otherwise noted, we use the default Magpie configuration. Magpie configures each Spark Worker to use all available CPUs. On Catalyst, hyperthreading is enabled, so we elected to fix Spark Workers to only use the 24 physical cores available on each node. Magpie assigns 80% of node memory to Spark Workers and increases `spark.shuffle.file.buffer` from 32K to 1024K. We set all Spark timeout values to 12000 seconds since Lustre is slow to respond when loaded and we had no issues with node failures in our HPC environment. The remaining configuration is identical to the defaults provided by Spark.

5 EXPERIMENTAL RESULTS

In this section, we present a set of experiments designed to illustrate our proposed *best practices* for using Spark to perform ML on simulation data residing on HPC systems. In order to avoid the fatal out-of-disk errors, our aim is to minimize storage requirements and overall runtime through user-configurable options.

5.1 User-Configurable Options

In order to make our *best practice* recommendations as broadly applicable as possible, we focus on user-configurable options rather than customizations to Spark itself. The advantage of this approach is that our recommendations are general and applicable across different Spark versions and configurations. Taken together, we refer to this set of recommendations as TEDX.

5.1.1 Compression. As we saw in Figure 4, enabling RDD compression in Spark is an effective way to reduce disk utilization (i.e., the x-factor) with minimal computational overhead. What is surprising in Figure 4b is that Lustre, even with RDD compression disabled, was able to reduce the x-factor more than compressed SSD. As it turns out, the reason behind this is that at LLNL, Lustre uses ZFS for its back-end storage, which enables transparent compression so all data is automatically compressed. The fact that Lustre/ZFS appears to compress Spark scratch data beyond Spark’s default compression suggests that further x-factor reductions may be possible by tuning Spark’s RDD compression.

To further explore the compression question, we experimented with ZStd, one of the alternative RDD lossless compression algorithms that Spark provides. The Spark compression algorithm is specified using `spark.io.compression.codec` option. Table 1 presents results from a set of experiments comparing the Spark default LZ4 with ZStd over a range of data sizes (1TB, 2TB and 4TB) and node counts (32, 64 and 128). For each 3x3 sub-table in Table 1, columns capture strong scaling (i.e., fixed data size, varying node count) and the diagonals (indicated in bold) capture weak scaling (i.e., fixed data size per node).

For SSD scratch, we observe a runtime overhead of 11%-31% using ZStd rather than Spark’s default LZ4. However, we also observe a substantial reduction in x-factor (>58%) due to using ZStd instead of LZ4. For large-scale problems, this space/time trade-off will be well worth it, especially when the choice is between getting an answer 15% slower and not getting an answer at all due to out-of-disk errors. One ZStd configuration detail: we use the default ZStd level-1. After experimenting with a range of compression levels, we found quickly diminishing returns for anything above level-1.

5.1.2 Hybrid Strategy. To address the runtime performance gap between SSD and Lustre that we saw in Figure 4, we propose a “hybrid” strategy, utilizing SSD as much as possible and Lustre only as necessary to prevent out-of-disk failures. To implement the hybrid strategy, we take advantage of a little-known option in Spark that allows for multiple user-specified scratch directories. Spark will store data to each of these scratch directories in a round-robin fashion. This allows us to control the proportion of data written to SSD vs. Lustre by specifying an appropriate number of scratch directories in each location. For example, `spark.local.dir` could be

	Data Size	SSD: LZ4–32MB			SSD: ZStd–32MB			Lustre: LZ4–32MB			Lustre: ZStd–128MB		
		1TB	2TB	4TB	1TB	2TB	4TB	1TB	2TB	4TB	1TB	2TB	4TB
Runtime	32 Nodes	1.58h	2.89h	5.48h	1.75h	3.34h	6.01h	2.15h	4.11h	8.32h	1.85h	3.51h	6.51h
	64 Nodes	0.99h	1.51h	2.81h	1.04h	1.75h	3.41h	1.44h	2.97h	5.62h	1.20h	2.06h	3.76h
	128 Nodes	0.49h	0.94h	1.61h	0.55h	1.08h	2.11h	1.26h	2.60h	4.99h	1.02h	1.87h	3.18h
X-factor	32 Nodes	4.14x	4.25x	4.32x	1.69x	1.73x	1.77x	2.80x	2.85x	2.92x	1.60x	1.70x	1.73x
	64 Nodes	3.87x	4.15x	4.27x	1.61x	1.71x	1.76x	2.60x	2.80x	2.88x	1.60x	1.65x	1.70x
	128 Nodes	2.98x	3.91x	4.17x	1.29x	1.62x	1.73x	2.10x	2.65x	2.80x	1.20x	1.50x	1.60x

Table 1: Comparison of Spark defaults (LZ4 RDD compression and 32MB block size) with TEDX (ZStd RDD compression and 128MB block size). For SSD scratch, TEDX reduces space requirements by >58% while incurring a 11%-31% runtime overhead. For Lustre scratch, TEDX simultaneously reduces space by >40% and runtime by up to 36%.

Figure 5: The hybrid strategy enables larger data sizes than SSD-only and faster runtimes than Lustre-only. It can be adjusted (here 50/50) to balance problem size and runtime.

set to “/ssd/username,/lustre/username” to inform Spark to store half of the ephemeral data to SSD and half to Lustre.

Figure 5 shows weak scaling results (input data size per node fixed at 8GB) for SSD-only, Lustre-only, and our hybrid strategy (here 50% SSD/50% Lustre). SSD-only scales nearly perfectly, remaining flat regardless of cluster/data size. In contrast, the runtime of Lustre-only increases dramatically with cluster/data size. The hybrid approach achieves the desired effect, running significantly faster than Lustre-only, while enabling scaling to data sizes larger than possible with SSD-only.

5.1.3 Block Size. As we saw from Figure 5, judicious use of network storage (i.e., the “hybrid” strategy) provides an effective means of reducing node-local storage requirements. However, Figure 5 also shows that there is a runtime cost to using Lustre vs. SSD. To address the relatively poor performance of Lustre, we consider the effects of block size on runtime. In general, Lustre is optimized for large parallel writes and is known to slow down when asked to read and write lots of small files. Therefore, we hypothesize that the performance on Lustre will improve when reading and writing fewer large blocks than many small blocks. To test this hypothesis, we ran a set of experiments with both SSD-only and Lustre-only scratch, varying the Spark block size configuration, which is specified using `fs.local.block.size` option. The results of these experiments are shown in Figure 6. We see that as the block size increases, the number of Spark intermediate files decreases (with the x-factor unchanged), and Lustre performance improves

Figure 6: Training RF on 32 nodes, 4TB of data with varying block size. As block size grows, the number of temporary Spark files decreases and Lustre performance improves.

to the point where it is on par with SSD performance. Note that SSD performance is largely unaffected by block size.

5.1.4 Summary of User-Configurable Options. Table 1 provides a summary of the performance of Spark defaults (LZ4 RDD compression and 32MB block size) with TEDX, our *best practice* recommendations for user-configurable options (ZStd RDD compression and 128MB block size). For SSD, we stick with the default block size of 32MB since SSD performance is insensitive to block size (blue bars in Figure 6). Enabling ZStd compression reduces space requirements by >58% while incurring a runtime increase of 11%-31%. For Lustre, the combination of ZStd compression and a larger block size simultaneously reduces space requirements by >40% and runtime by up to 36%.

5.2 Application Scenarios

In the previous section, we performed an in-depth study of scaling performance for a limited application scenario: running a single isolated Spark job to train a RF regressor on ALE simulation failure data. In this section, we consider the implications of generalizing the application scenario in two ways: 1) running multiple concurrent Spark jobs and 2) running a different Spark data analysis workflow with differing performance characteristics.

5.2.1 Concurrent Usage. With the block size optimizations described in Section 5.1.3, Lustre is almost as fast as SSD under ideal

Figure 7: Training RF on 32 nodes, 1TB of data, and varying the number of concurrent runs. The performance gap between SSD and Lustre grows as concurrency increases. The hybrid strategy helps mitigate the effects of concurrency.

conditions. However, network storage is a shared resource and is, therefore, subject to slowdowns due to activities of other users. Obviously, node-local storage is a private resource and should not be subject to such external slowdowns. To simulate Big Data load on network storage, we ran a set of experiments where we varied the number of identical Spark jobs that we run simultaneously. We repeated these experiments using both SSD-only and Lustre-only as scratch space. Figure 7 shows the results of these experiments. We see that, even with a sufficiently large block size (128MB), the performance gap between SSD and Lustre re-emerges as concurrency (in the form of multiple simultaneous jobs) increases. We also see that the proposed hybrid strategy helps mitigate the negative effects of concurrency on Lustre, achieving runtimes closer to SSD-only performance than Lustre-only performance.

5.2.2 Failure Mode Analysis. Motivated by the need to understand simulation failure modes (as described in Section 3), the second ML analysis we evaluated is a user-defined Spark pipeline involving k-means clustering. As in our RF evaluation, we ran a series of scaling experiments by varying node count and data size. Figure 8 compares both the runtime performance and x-factor for the failure mode analysis pipeline using the current state-of-the-practice (no compression) vs. TEDX (RDD compression with ZStd) across different data sizes using 32 nodes. Although this analysis pipeline has very different performance characteristics than RF (in particular, a much smaller x-factor), we still see a benefit to TEDX, which reduces space requirements up to 30% (blue bar vs. orange bar) with negligible runtime overhead (green dotted line vs green solid line). Table 2 presents weak scaling results for both runtime and x-factor for TEDX with SSD scratch. We varied the data size from 1TB to 40TB, and the node counts from 32 to 160. We see that TEDX scales well for this analysis pipeline, achieving a relatively flat performance even for large data sizes and node counts.

5.3 Large-Scale Runs

Throughout the course of our experiments, we performed a number of large-scale runs with different Spark configurations. Here, we highlight two such runs, one for RF regression and one for failure mode analysis. We visualize these runs in Figure 9, showing both memory (orange) and disk (blue) utilization across time and the

Figure 8: Failure mode analysis on 32 nodes using SSD Spark scratch, comparing current practice (no compression) with TEDX (ZStd RDD compression). TEDX reduces space requirements up to 30% with negligible runtime cost.

corresponding stages of each Spark job. This plot shows the average utilization plus the min/max of the allocated nodes.

Figure 9a shows a large-scale run using TEDX, in which we trained a single RF regression model on the full 76TB data (over one trillion training examples) using the entire Catalyst system, which had 293 non-drained/down nodes. We observe the following flow of computation: 1) memory fills as data is read from Lustre and persisted (stage 2), 2) once memory fills up, SSD begins to fill as well (stage 5), 3) once all data has been persisted, the main computation begins (stage 6 on). By training RF models with varying *maxDepth*, we deduced that the Spark RF implementation runs 2 stages for every level of the learned trees. So in this case (*maxDepth*=4), the 8 stages (6–13) represent the main computation of the RF training algorithm. (In Figure 3, the sharp drops in the number of shuffle files correlate directly to the Spark stages of the RF algorithm.) Overall, training an RF regression model on 76TB of data only took 12.5 hours on 293 Catalyst nodes!

Figure 9b shows a large-scale run in which we performed failure mode analysis on 40TB of data on 160 nodes of Catalyst. Here, we see a similar progression as with the RF computation, with a few notable exceptions; there are additional stages for data standardization (stages 3–7) and cluster quality evaluation (stages 50–52). Once the data is persisted to memory and disk, the heart of the computation is 20 iterations of the k-means algorithm, labeled in the figure as “K-Means Clustering,” which spans stages 8–49. Overall, performing the failure mode analysis on 40TB of data took 23.5 hours on 160 Catalyst nodes.

5.4 Machine Learning Results

The *best practices* described in this paper for exploiting Spark on HPC systems enable investigation of questions on ML for simulation data that could not be answered previously. Here, we present preliminary ML results to demonstrate the kinds of experiments enabled by this work. Future work will include a complete investigation of questions regarding generalization across simulations and the effectiveness of various data sampling and labeling strategies.

5.4.1 Data Sampling Investigation. The first experiment is to investigate the effects of sub-sampling the training data. In order to

Figure 9: Memory and disk utilization across time for large-scale RF and failure mode analysis runs.

Experiment	Timing	X-factor
32 nodes for 1TB data	2.89h	1.18x
64 nodes for 2TB data	2.80h	1.19x
128 nodes for 4TB data	2.84h	1.22x
40 nodes for 10TB data	20.2h	1.33x
80 nodes for 20TB data	21.7h	1.34x
160 nodes for 40TB data	23.5h	1.35x

Table 2: Weak scaling experiments for failure mode analysis using TEDX (ZStd RDD compression). TEDX exhibits good (flat) weak scaling performance for a wide range of node counts and data sizes.

evaluate ML models trained using different sampling strategies, we first need to establish a performance baseline using the full 76TB data. Although for a scaling evaluation, it makes sense to train on the full 76TB data, a meaningful ML evaluation requires held-out test data. Therefore, in order to get an accurate measure of generalization performance across simulations, we partitioned the full data into separate training (40TB) and testing (36TB) simulation

variations. Overall test error on the full data set is extremely low: 0.002 root mean squared error (RMSE). However, digging a level deeper reveals highly imbalanced errors. Specifically, the model achieves low error (RMSE=0.0001) on non-failed zones, but much larger error (RMSE=0.575) on failed zones. This result is not surprising due to an imbalanced class distribution in the training data for this problem (i.e., the majority of label values are 0).

For these experiments, we use the following procedure. 1) Sub-sample simulation variations to reduce the overall amount of training data. 2) Further sub-sample instances with the majority label value of 0 as necessary to equalize the distribution of 0 and non-0 labels. 3) Train an RF model and evaluate it on a test set from a completely separate simulation variation. We decided to use the simulation variation (piston=100, density=0.17) for the test set, and constructed training data sets, from 1TB to 32TB, using the remaining simulation variations.

A key issue here is: can we improve error imbalance by sub-sampling to better balance the training label distribution? One approach is to sub-sample instances with the majority label value (0 in this case) to balance out the label distribution prior to training. Fortunately, this kind of data pre-processing fits easily and cleanly into our Spark ML pipeline. Table 3 shows the effectiveness of this

Training Data	Pos Class	Neg Class	Overall
1TB	0.266	0.0432	0.192
2TB	0.232	0.0368	0.166
4TB	0.213	0.0320	0.152
8TB	0.211	0.0294	0.152
16TB	0.209	0.0287	0.149
32TB	0.208	0.0288	0.149

Table 3: Training RF on class-balanced data reduces error-imbalance and overall error. Shown is RMSE for positive class (non-0 labels), negative class (0 labels), and overall.

balanced training strategy in reducing error for predicting failed zones and producing a more balanced error distribution. Specifically, the approach reduced RMSE for non-0 labeled instances from 0.575 to as low as 0.208. Overall, we see that prediction error decreases as the amount of training data increases and that the error plateaus around 16TB of training data for this specific simulation.

5.4.2 Failure Mode Investigation. The second experiment is to investigate the different ways in which simulations can fail using the failure mode analysis with k-means clustering. By setting $k=2$, we are interested to see how well k-means is able to distinguish failures from non-failures, especially as we introduce more simulation variations into the training process. For this experiment, we ran the failure mode analysis pipeline on a range of data sizes: 4TB, 10TB, 20TB and 40TB. One of the outputs from the pipeline is the silhouette score, which is a measure of cluster consistency. Silhouette compares intra-cluster similarity to inter-cluster similarity and ranges from -1 to 1, with higher values indicating more consistent clusters. For the above data sizes, we attained scores of: 0.183, 0.239, 0.837 and 0.852, respectively, which clearly shows a trend indicating that cluster quality improves with more data.

Another output from the failure mode analysis are the k-means cluster centroids. Figure 10 shows a parallel coordinates plot of the centroids in their 16-dimensional feature space. Each centroid is represented by a line that crosses each feature axis at its feature value. We colored the centroids based on whether it contains the majority of failures (blue) or not (orange), and we ordered the feature axes based on how well the clusters are distinguished from each other in terms of their feature values. From Figure 10, we make the following observations: 1) failure and non-failure clusters are clearly distinguishable for the majority of features, 2) non-failure clusters seem to share a common mode in feature space, and 3) there appear to be distinct modes of failures as we increase in data size. Further investigation is required along these lines, but preliminary results indicate a common non-failure mode along with several distinct failure modes.

6 DISCUSSION

The Ephemeral Data Explosion — For multi-pass Spark computations, persistence can be both a blessing and a curse. The *blessing of persistence* is that it reduces redundant I/O leading to vastly reduced runtimes. The *curse of persistence* is the *ephemeral data explosion*, discussed in detail throughout this paper. As we have

seen, some computations explode more than others: RF may be as bad as 12x, whereas k-means may be less than 2x. In this paper, we explored both cases, demonstrating that our TEDX approach can mitigate the damage from large data explosions, while incurring a negligible runtime overhead in the case of small explosions.

Compression — The current state-of-the-practice is to persist RDDs uncompressed in order to reduce runtime. However, there are lessons to be learned from the HPC world here: flops are free, data movement is expensive, and latency is often hidden by I/O savings [1, 2]. This paper presents a further lesson from the world of Big Data: the mitigating effects of compression on the *ephemeral data explosion* caused by RDD persistence can outweigh any runtime overhead due to data compression. In some cases, RDD compression may allow all RDDs to be stored on fast node-local storage, obviating the need to spill to network storage, and thus substantially decreasing overall runtime. In cases where it is necessary to persist RDDs to network storage, runtime savings can result from compressing RDDs before sending them across the network. In general, we recommend enabling ZStd compression of Spark RDDs for problems where node-local disk space is an issue.

Network Storage — If compression is insufficient to prevent a data explosion and subsequent out-of-disk errors, the next step is to spill to network storage. The runtime cost of network storage can be substantial if not utilized carefully. Our recommendations: 1) compress first, spill to network storage second, 2) utilize the proposed hybrid strategy to maximize node-local storage and minimize network storage, and 3) increase Spark default block size to improve network storage performance. Increasing block size reduces both the number of Spark tasks and the number intermediate files written to Spark scratch. For large jobs, there is no benefit to Spark to manage millions of tasks and no benefit to Lustre to read and write millions of files.

Machine Learning Results — Putting these *best practices* into practice for failure prediction in ALE simulations enabled us run experiments at a scale that was previously impossible to address some key ML issues. Our preliminary findings are: 1) Training on more simulation data improves model generalization performance up to a point and this point is well beyond what can fit in memory on a single core (e.g., 16TB in Table 3). However, effective training does not require data from every zone of every cycle in every simulation. Future work is required to fully understand the interactions between simulation variations, failure diversity, and specific sampling strategies. 2) Sub-sampling training data to reduce distributional skew of labels can reduce error imbalance and overall error. 3) In general, non-failed zones tend to follow a similar pattern across simulations, whereas failed zones occur in different ways in different simulations. For future work, we will explore how to construct training sets using a full range of failure modes to improve predictive performance.

7 RELATED WORK

In recent years, there has been a considerable amount of research in using Big Data analytics on HPC systems [32]. Our work is unique in that the scale of our data is 2–3 orders of magnitude more than what has been published. The *best practices* we developed allow us to address issues at massive data scales that may not have been

Figure 10: Parallel coordinates plot of k-means cluster centroids from the failure mode analysis reveals a dominant mode for non-failed zones (orange), along with several distinct failure modes (blue).

encountered previously. What makes this work novel is that our *best practices* are in direct contrast to what is recommended in the state-of-the-art, due to data scaling issues.

One area of current research can be described as re-implementing existing simulation or analysis algorithms using Spark. [26] presents an analysis of the targeted application and describe different deep learning technologies that are implemented using Spark. [33] compares the performance of a classification algorithm implemented using Spark versus un-tuned MPI. The problem domain is High Energy Physics and the use case is the classification of an unknown event by comparing it to many known simulated events. Although the Spark implementation scaled well with the data, it did not match the MPI implementation, which uses the Lustre file system. [34] runs a Spark instance and reads the data into a Spark RDD to perform data analytics. Other related work includes parallel implementations of ML algorithms on HPC systems [9].

Another research area is focused on developing APIs to speed up data access for analytics use cases [17, 25, 37]. GLEAN [37] is a framework that provides an infrastructure for accelerating I/O interfacing with running simulations for co-analysis and supports in-situ analysis. Running FLASH, an astrophysics simulation on leadership-class systems, they demonstrated up to 14-fold improvement over post-processing by using in-situ analysis with I/O acceleration. H5Spark [25] is another effort to speed up data access using an efficient parallel I/O API for Spark on HPC systems. Design decisions for the API were made after studying several scientific use cases.

Recently, research on tuning Spark performance has been on the rise [11, 25, 31, 40]. This is due to the increase in the number of applications that use Spark for data intensive tasks [13, 28, 29]. The work described in [29] is an attempt to find the parameters to tune Spark performance using ML. Identifying the best Spark configuration parameters for a specific application is very challenging. Other approaches to dealing with this issue include surveying scientific applications and mapping them to novel architectures [16], and benchmarking a variety of HPC architectures using data science algorithms: singular value decomposition, linear regression, and k-means [39]. The k-means tests were performed using 1, 2 and 5 nodes with up to 50 million training instances.

Finally, there is research on porting and scaling Spark for specific HPC systems, such as [12], which performed its evaluation on the Cray XC system and was able to scale up to 10,000 cores. [19, 36] did comparative studies between Big Data analytics frameworks and specific file systems. [36] compared the performance of Hadoop, Spark and Flink to measure the performance gains when using an in-memory data processing system versus disk-based. [19] studied the performance of Spark using two different file systems: Tachyon and Triple-H and compared their performance to HDFS. Our experiments utilize both node-local and network storage, and we show how to use them in a hybrid strategy to avoid out-of-disk errors and improve runtime.

8 CONCLUSION

In this paper, we addressed the challenge of analyzing simulation data via Spark at scale on HPC systems. We focused primarily on the challenge of the *ephemeral data explosion* brought about by the *curse of persistence* in the Spark framework. For multi-pass Spark computations, data persistence is essential in reducing I/O, but it comes at a cost in terms of node-local storage space. In some cases (e.g., the popular RF algorithm), Spark scratch data can consume an order of magnitude more space than the input data being analyzed, leading to fatal out-of-disk errors.

We investigated the real-world application of training ML models to predict and analyze mesh tangling failures in ALE simulations, and successfully trained an RF regression model on 76TB of data (over one trillion training examples) in 12.5 hours! This problem is 2–3 orders of magnitude larger than prior work. Based on extensive experiments, we provide the following *best practices* with user-configurable options for taming the *ephemeral data explosion* while minimizing runtime:

- **Enable ZStd compression (level-1) of Spark RDDs** for large problems. The mitigating effects of compression often outweigh any runtime overhead by: a) allowing more data to remain on fast node-local storage and b) substantially reducing network communication for network storage.
- **Utilize a hybrid node-local/network storage strategy** via Spark scratch directories to maximize the use of node-local storage and minimize the use of network storage.

- **Increase Spark default block size (up to 128MB)** to improve network storage performance. Increasing block size reduces both the number of Spark tasks and the number files written to Spark scratch.

Overall, these recommendations lead to as much as 7x reduction in disk utilization for training ML models, with negligible increases or even decreases in runtime.

ACKNOWLEDGMENTS

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-772122), with funding provided by LDRD 16-ERD-036 and by Kevin Mueller from the Defense Threat Reduction Agency.

REFERENCES

- [1] 2010. The Opportunities and Challenges of Exascale Computing, ASCAC Subcommittee Report. http://science.energy.gov/-/media/ascr/ascac/pdf/reports/Exascale_subcommittee_report.pdf.
- [2] 2012. Report out from the Exascale Research Planning Workshop Working Session on Data Management, Visualization, IO and Storage. http://exascale-research.lawworks.org/apr2012planningworkshop/application/layouts/exascale-planning-workshop/public/docs/PRES_WorkingSession-DataIO_120420.pdf.
- [3] 2019. Hadoop. <https://hadoop.apache.org/>
- [4] 2019. hanythingondemand. <https://github.com/hpcugent/hanythingondemand>
- [5] 2019. Magpie. <https://github.com/LLNL/magpie>
- [6] 2019. Sort Benchmark. <http://sortbenchmark.org/>
- [7] 2019. Spark. <https://spark.apache.org/>
- [8] 2019. sparkhpc. <https://github.com/rokrskar/sparkhpc>
- [9] H. Abdulsalam, D. Skillicorn, and P. Martin. 2011. Classification Using Streaming Random Forests. *IEEE Transactions on Knowledge and Data Engineering* 23, 1 (2011), 22–36.
- [10] T. Baer, P. Peltz, J. Yin, and E. Begoli. 2015. Integrating Apache Spark into PBS-Based HPC Environments. In *XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*. Article 34.
- [11] A. Cachuan. 2019. *How does Facebook tune Apache Spark for Large-Scale Workloads?* <https://towardsdatascience.com/how-does-facebook-tune-apache-spark-for-large-scale-workloads-3238ddda0830>
- [12] N. Chaimov, A. Malony, S. Canon, C. Iancu, K. Ibrahim, and J. Srinivasan. 2016. Scaling Spark on HPC Systems. In *ACM International Symposium on High-Performance Parallel and Distributed Computing*. 97–110.
- [13] T. Chiba, T. Yoshimura, M. Horie, and H. Horii. 2018. Towards Selecting Best Combination of SQL-on-Hadoop Systems and JVMs. In *IEEE International Conference on Cloud Computing*. 245–252.
- [14] R. da Silva, R. Filgueira, I. Pietri, M. Jiang, R. Sakellariou, and E. Deelman. 2017. A Characterization of Workflow Management Systems for Extreme-Scale Applications. *Future Generation Computer Systems* 75 (2017), 228–238.
- [15] E. Deelman, A. Mandal, M. Jiang, and R. Sakellariou. 2019. The Role of Machine Learning in Scientific Workflows. *The International Journal of High Performance Computing Applications* 33, 6 (2019), 1128–1139.
- [16] G. Fox, S. Jha, J. Qiu, and A. Luckow. 2014. Towards an Understanding of Facets and Exemplars of Big Data Applications. In *20 Years of Beowulf: Workshop to Honor of Thomas Sterling's 65th Birthday*. 7–16.
- [17] Y. Fu, F. Song, and L. Zhu. 2016. Modeling and Implementation of an Asynchronous Approach to Integrating HPC and Big Data Analysis. *Procedia Computer Science* 80 (2016), 52–62.
- [18] J. Harney, S. Lim, S. Sukumar, D. Stansberry, and P. Xenopoulos. 2016. On-Demand Data Analytics in HPC Environments at Leadership Computing Facilities: Challenges and Experiences. In *IEEE International Conference on Big Data*. 2087–2096.
- [19] N. Islam, M. Wasi-ur-Rahman, X. Lu, D. Shankar, and D. Panda. 2015. Performance Characterization and Acceleration of In-Memory File Systems for Hadoop and Spark Applications on HPC Clusters. In *IEEE International Conference on Big Data*. 243–252.
- [20] M. Jiang, B. Gallagher, J. Kallman, and D. Laney. 2016. A Supervised Learning Framework for Arbitrary Lagrangian-Eulerian Simulations. In *IEEE International Conference on Machine Learning and Applications*. 977–982.
- [21] M. Jiang, B. Gallagher, N. Mandell, A. Maguire, K. Henderson, and G. Weinert. 2019. A Deep Learning Framework for Mesh Relaxation in Arbitrary Lagrangian-Eulerian Simulations. In *SPIE Applications of Machine Learning*, Vol. 11139. 168–182.
- [22] S. Krishnan, M. Tatineni, and C. Baru. 2011. myHadoop—Hadoop-on-Demand on Traditional HPC Resources. <https://pdfs.semanticscholar.org/1357/8e78620db19f8185060f206a95b495435cc8.pdf>
- [23] J. Ling, R. Jones, and J. Templeton. 2016. Machine Learning Strategies for Systems with Invariance Properties. *Journal of Computational Physics* 318 (2016), 22–35.
- [24] J. Ling, W. Kegelmeyer, K. Aditya, H. Kolla, K. Reed, T. Shead, and W. Davis. 2017. Using Feature Importance Metrics to Detect Events of Interest in Scientific Computing Applications. In *IEEE Symposium on Large Data Analysis and Visualization*. 55–63.
- [25] J. Liu, E. Racah, Q. Koziol, and R. Canon. 2016. H5spark: Bridging the I/O Gap Between Spark and Scientific Data Formats on HPC Systems. *Cray User Group*.
- [26] N. Malitsky. 2016. Bringing the HPC Reconstruction Algorithms to Big Data Platforms. In *Scientific Data Summit*. 1–8.
- [27] S. Michael, A. Thota, and R. Henschel. 2014. HPCHadoop: A Framework to Run Hadoop on Cray X-Series Supercomputers. In *Cray User Group*.
- [28] N. Nguyen, M. Khan, Y. Albayram, and K. Wang. 2017. Understanding the Influence of Configuration Settings: An Execution Model-Driven Framework for Apache Spark Platform. In *IEEE International Conference on Cloud Computing*. 802–807.
- [29] N. Nguyen, M. Khan, and K. Wang. 2018. Towards Automatic Tuning of Apache Spark Configuration. In *IEEE International Conference on Cloud Computing*. 417–425.
- [30] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-Learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.
- [31] T. Perez, W. Chen, R. Ji, L. Liu, and X. Zhou. 2018. PETS: Bottleneck-Aware Spark Tuning with Parameter Ensembles. In *International Conference on Computer Communication and Networks*. 1–9.
- [32] D. Reed and J. Dongarra. 2015. Exascale Computing and Big Data. *Communications of ACM* 58, 7 (2015), 56–68.
- [33] S. Sehrish, J. Kowalkowski, and M. Paterno. 2016. Exploring the Performance of Spark for a Scientific Use Case. In *IEEE International Parallel and Distributed Processing Symposium Workshops*. 1653–1659.
- [34] S. Sehrish, J. Kowalkowski, and M. Paterno. 2017. Spark and HPC for High Energy Physics Data Analyses. In *IEEE International Parallel and Distributed Processing Symposium Workshops*. 1048–1057.
- [35] R. Tous, A. Gounaris, C. Tripiiana, J. Torres, S. Girona, E. Ayguadé, J. Labarta, Y. Becerra, D. Carrera, and M. Valero. 2015. Spark Deployment and Performance Evaluation on the MareNostrum Supercomputer. In *IEEE International Conference on Big Data*. 299–306.
- [36] J. Veiga, R. Expósito, X. Pardo, G. Taboada, and J. Tourifio. 2016. Performance Evaluation of Big Data Frameworks for Large-Scale Data Analytics. In *IEEE International Conference on Big Data*. 424–431.
- [37] V. Vishwanath, M. Hereld, M. Papka, R. Hudson, G. Jordan, and C. Daley. 2011. In Situ Data Analysis and I/O Acceleration of FLASH Astrophysics Simulation on Leadership-Class System Using GLEAN. In *SciDAC, Journal of Physics: Conference Series*.
- [38] Y. Wang, R. Goldstone, W. Yu, and T. Wang. 2014. Characterization and Optimization of Memory-Resident MapReduce on HPC Systems. In *IEEE International Parallel and Distributed Processing Symposium*. 799–808.
- [39] P. Xenopoulos, J. Daniel, M. Matheson, and S. Sukumar. 2016. Big Data Analytics on HPC Architectures: Performance and Cost. In *IEEE International Conference on Big Data*. 2286–2295.
- [40] J. Zhang. 2018. *HTuning Spark Performance*. <https://qspace.library.queensu.ca/handle/1974/24439>