Kira: Processing Astronomy Imagery Using Big Data Technology

Zhao Zhang Kyle Barbary Frank Austin Nothaft Evan R. Sparks
Oliver Zahn Michael J. Franklin David A. Patterson Saul Perlmutter

Abstract—Scientific analyses commonly compose multiple single-process programs into a dataflow. An end-to-end dataflow of single-process programs is known as a many-task application. Typically, HPC tools are used to parallelize these analyses. In this work, we investigate an alternate approach that uses Apache Spark—a modern platform for data intensive computing—to parallelize many-task applications. We implement Kira, a flexible and distributed astronomy image processing toolkit, and its Source Extractor (Kira SE) application. Using Kira SE as a case study, we examine the programming flexibility, dataflow richness, scheduling capacity and performance of Apache Spark running on the Amazon EC2 cloud. By exploiting data locality, Kira SE achieves a 4.1× speedup over an equivalent C program when analyzing a 1TB dataset using 512 cores on the Amazon EC2 cloud. Furthermore, Kira SE on the Amazon EC2 cloud achieves a 1.8× speedup over the C program on the NERSC Edison supercomputer. A 128-core Amazon EC2 cloud deployment of Kira SE using Spark Streaming can achieve a second-scale latency with a sustained throughput of ∼800 MB/s. Our experience with Kira demonstrates that data intensive computing platforms like Apache Spark are a performant alternative for many-task scientific applications.

Index Terms—Distributed Computing, Data Processing, Astronomy, Many-Task Applications

1 Introduction

Dramatic increases in dataset sizes have made data processing a major bottleneck for scientific research in many disciplines, such as astronomy, genomics, social science, and neuroscience. Researchers frequently start with a C or Fortran program that is optimized for processing a small amount of data on a single-node workstation and then use distributed processing frameworks to improve processing capacity. Examples include the Montage astronomy image mosaic application [1], the sequence alignment tool BLAST [2], and high energy physics histogram analysis [3]. These applications are known as many-task applications because they comprise many small single-process tasks that are connected by dataflow patterns [4].

Scientists have used dedicated workflow systems (e.g., HTCondor [5]), parallel frameworks (e.g., the Message Passing Interface, MPI [6]), and more recently the data processing system Hadoop [7] to build these applications [3], [8]. Each approach has its own advantages such as provenance tracking, high scalability, and automated parallelism. However, these approaches also have shortcomings such as limited programming flexibility, lack of fault-tolerance, or a rigid programming model.

Apache Spark [9] was designed to support fast iterative data analyses on very large datasets by relying on an in-memory data model that allows for the caching of intermediate results. By using a directed acyclic graph (DAG) to describe parallel tasks, Apache Spark provides resilience against transient failures by replaying computational lineage and can optimize for data locality when scheduling tasks. These features have made Apache Spark a widely used distributed computing platform for machine learning [10], [11] and computational science [12], [13], [14], and make Apache Spark a natural platform for executing many-task applications.

In addition to traditional batch many-task processing, some more recent scientific applications also have latency requirements that could easily be met by a stream processing approach. For example, in the Large Synoptic Survey Telescope (LSST, [15]) survey, the image processing pipeline needs to send out alerts to the community based on transient events such as supernovae detections. It also assesses data quality at near real-time to provide feedback to telescope operations. The latency in these cases should not exceed 60 seconds [16]. Apache Spark's built-in streaming processing module [17] provides a convenient way to deploy applications enabled by Apache Spark in a streaming manner.

Previous research using Apache Spark for computational science reimplemented the underlying domain algorithms [12], [13], [14]. In contrast, our research investigates how to leverage Apache Spark by reusing existing code bases for many-task applications. This approach avoids rewriting existing code which takes unnecessary effort and can introduce errors. We study this question in the context of Kira, an astronomy image processing toolkit [18]. In particular, we focus on the *source extractor* component of Kira (Kira SE) to evaluate the programing flexibility, dataflow richness

- *Zhao Zhang is with the AMPLab and Berkeley Institute for Data Science, UC Berkeley.*
- *Frank Austin Nothaft and David A. Patterson are with the AMPLab and ASPIRE Labs, UC Berkeley.*
- *Evan R. Sparks is with the AMPLab, UC Berkeley.*
- *Kyle Barbary, Oliver Zahn, and Saul Perlmutter are with the Berkeley Institute for Data Science and Berkeley Center for Cosmological Physics, UC Berkeley.*
- *Michael J. Franklin is with the Department of Computer Science, University of Chicago.*

and scheduling capacity of Apache Spark. Source extraction, is a common function in astronomy pipelines, that identifies point sources of light in an image. We evaluate Kira SE's performance by comparing against an equivalent C implementation that is parallelized using HPC tools. We also use Kira SE to examine the use of Spark Streaming to address the near real-time requirements of astronomy image processing, such as in the LSST. Leveraging a multi-language analytics platform like Apache Spark provides several advantages for many-task applications:

1) Apache Spark can use existing astronomy libraries written in Python and C. This allows astronomers to reuse existing libraries to build new analysis functionality.
2) Apache Spark supports a broad range of dataflow patterns such as pipeline, broadcast, scatter, gather, reduce, all-gather, and all-to-all (shuffle). This broad dataflow pattern support can be used to optimize the data transfer between computation stages.
3) Apache Spark's broad support for underlying file systems allows Kira to process data stored in a distributed file system such as HDFS [19], as well as data stored in HPC-style shared file systems such as GlusterFS [20] or Lustre [21], or cloud blob stores like Amazon S3 and Microsoft DataLake.
4) Apache Spark also provides fault tolerance, a feature missing from MPI [6].
5) Kira can leverage other components of the Berkeley Data Analytics Stack (BDAS), e.g., Spark Streaming [17].

Our experiments indicate that in addition to these benefits of flexibility and ease of development, Apache Spark provides performance benefits that can be leveraged by a scientific computing application such as Kira. For example, our results show that Apache Spark is capable of managing $O(10^6)$ tasks and that Kira SE runs 4.1× faster than an equivalent C program when using a shared file system on the Amazon EC2 cloud with the 1TB from the Sloan Digital Sky Survey [22] Data Release 7. We also show that running Kira SE in the Amazon EC2 cloud can achieve performance that is 1.8× faster than that of the equivalent C program running on the NERSC Edison supercomputer. Enabled by Spark Streaming, a deployment of Kira SE on 16 nodes on Amazon EC2 cloud achieves second-scale latency and a sustained throughput of ∼800 MB/s.

Our experience with Kira indicates that Big Data platforms such as Apache Spark are a competitive alternative for many-task scientific applications. We believe this is important, because leveraging such platforms would enable scientists to benefit from the rapid pace of innovation and large range of systems and technologies that are being driven by wide-spread interest in Big Data analytics. Kira is open source software released under an MIT license and is available from https://github.com/BIDS/Kira.

This paper is an extension of a previous conference paper [23]. Compared to that earlier work, in this paper, we optimize Kira SE's data layout and reduce data copying, resulting in significant performance improvements over the previous implementation. Further, we study how solid state disks can improve Kira SE's performance compared to spinning disks. We also investigate the feasibility of Kira SE when deployed as a steam application with Spark Streaming. In summary, this paper makes stronger claims about the applicability and the benefits of using Big Data technology (i.e., Apache Spark) for this astronomy imagery processing application and other many-task scientific applications.

2 Background

This section reviews the science behind sky surveys, introduces the source extraction kernel, explores engineering requirements, and discusses the origin and usage of Apache Spark.

2.1 Sky Surveys

Modern astronomical research is increasingly centered around large-scale sky surveys. Rather than selecting specific targets to study, such a survey will uniformly observe large swaths of the sky. Example surveys include the Sloan Digital Sky Survey (SDSS) [22], the Dark Energy Survey (DES) [24], and the Large Synoptic Survey Telescope (LSST, [15]). Enabled by new telescopes and cameras with wide fields of view, these surveys deliver huge datasets that can be used for many different scientific studies simultaneously.

In addition to studying the astrophysical properties of many different individual galaxies, the large scale of these surveys allows scientists to use the distribution of galaxies to study the biggest contemporary mysteries in astrophysics: dark matter, dark energy, and the properties of gravity. These surveys normally include a time component: each patch of the sky is imaged many times, with observations spread over hours, days, weeks or months. With this repeated imaging, transient events can be detected via "difference imaging". Transients such as supernovae can be detected in large numbers to better measure dark energy, and the large survey area often results in the discovery of new, extremely rare, transient phenomena.

2.2 Source Extraction

Source extraction is a key step in astronomical image processing pipelines. SExtractor [25] is a widely used C application for source extraction. The source extraction kernel identifies and extracts point sources of light against the dark background of a standard telescope image. Although SExtractor is currently implemented as a monolithic C program, the application's logic can be divided into background estimation, background removal, object detection, and astrometric and photometric estimation.

Astronomers can improve extraction accuracy by running multiple iterations of source extraction. Detected objects are removed after each iteration. While the original C program contains the required functionality for building this iterative source extractor, it does not expose the interfaces through the command line. To resolve this issue, SEP [26] reorganizes the code base of SExtractor to expose the core estimation, removal, and detection functions as a library. SEP provides both C and Python interfaces. Users can then build the iterative source extractor using SEP primitives. Our system, Kira SE is implemented by calling into the SEP library.

2.3 Engineering Requirements

In some experiments—such as ones that search for supernovae explosions—it is important to process the images as rapidly as possible. A rapid processing pipeline enables astronomers to trigger follow-up observations with more sensitive instrumentation before the peak of the supernovae occurs. High throughput is also needed in large scale sky survey pipelines that perform real time data analysis, such as the LSST [15]. The LSST uses a 2.4m-wide optical telescope that captures 3.2 billion pixels per image. This telescope produces approximately 12.8 GB in 39 seconds for a sustained rate of ∼330 MB per second, and a typical night produces 13 TB of data [27]. Over the planned 10-year project, the survey is expected produce 60 PB of raw data, which will be consolidated into a 15 PB catalog. LSST also requires a latency that should not exceed 60 seconds [16] to notice the community with transient events and quality assessment alerts. The latency requirements of this pipeline couple with the massive amount of data captured to create a challenging throughput requirement for the processing pipeline.

2.4 Apache Spark

Apache Spark is a dataflow-based execution system that provides a functional, collection oriented API [9]. Apache Spark's development was motivated by a need for a system that could rapidly execute iterative workloads on very large datasets, as is common in large scale machine learning [28]. Apache Spark has become widely adopted in industry, and academic research groups have used Apache Spark for the analysis of scientific datasets in areas such as neuroscience [12] and genomics [14].

Apache Spark is centered around the Resilient Distributed Dataset (RDD) abstraction [9]. To a programmer, RDDs appear as an immutable collection of independent items that are distributed across the cluster. RDDs are immutable and are transformed using a functional API. Operations on RDDs are evaluated lazily, enabling the system to schedule execution and data movement with better knowledge of the operations to be performed than systems that immediately execute each stage. Apache Spark provides Scala, Java, and Python programming interfaces. By default, Apache Spark uses HDFS [19] for persistent storage, but it can process data stored in Amazon S3 or on a shared file system such as GlusterFS [20] or Lustre [21]. Apache Spark provides fault tolerance via lineage-based recomputation. If a partition of data is lost, Apache Spark can recover the data by re-executing the section of the DAG that computed the lost partition.

Apache Spark also provides a streaming interface using a discretized stream (D-Stream) architecture, which leverages the RDD abstraction [17]. D-Stream partitions data streams into mini-batches, then applies a sequence of transformations on the mini-batches. Spark streaming can recover from faults and stragglers quickly by using the lineage-based fault tolerance techniques of RDDs. D-Streams can provide subsecond latency when processing streaming data.

3 Applying Apache Spark to Many-Task Applications

Scientific analysis pipelines are frequently assembled by building a dataflow out of many single-process programs. Many-task applications arise in scientific research domains including astronomy, biochemistry, bioinformatics, psychology, economics, climate science, and neuroscience. In these applications, tasks are typically grouped into stages that are connected by producer-consumer data sharing relationships. A previous survey [29] identified seven common dataflow patterns among a group of many-task applications. The patterns include pipeline, broadcast, scatter, gather, reduce, all-gather, and all-to-all. Most many-task applications can be viewed as stages of independent tasks that are linked by these dataflow patterns.

The map-reduce [30] model uses a similar pattern to schedule jobs. Traditional map-reduce systems such as Google's MapReduce [30] and Apache Hadoop MapReduce [7] abstract producer-consumer relationships into a map stage and a reduce stage. These two stages are then linked by a data shuffle. Although these systems have proved very powerful for processing very large datasets, the map-reduce API has been criticized as inflexible [31]. Additionally, since jobs are restricted to a single map and reduce phase, tools such as FlumeJava [32] are necessary for assembling pipelines of map-reduce jobs. Since data is spilled to disk at the end of each map and reduce phase, traditional map-reduce platforms perform poorly on iterative and pipelined workflows [9].

To resolve these problems, second-generation map-reduce execution systems such as DryadLINQ [33] and Apache Spark [9] allow for applications to be decomposed into DAGs. In these DAGs, nodes represent computation, and the nodes are linked by dataflows. In Apache Spark, this abstraction is provided by RDDs [9]. Table 1 demonstrates how seven common dataflow patterns can be mapped to Apache Spark.

TABLE 1
Dataflow Pattern Primitives in Apache Spark

Pattern	Spark primitive
Pipeline	RDD.map()
Broadcast	sparkContext.broadcast()
Scatter	sparkContext.parallelize()
Gather	RDD.collect()
Reduce	RDD.reduce()
All-gather	RDD.collect().broadcast()
All-to-all	RDD.reduceByKey() or RDD.repartition()

Apache Spark improves upon Hadoop MapReduce by adding an in-memory processing model that natively supports iterative computation. As compared to other DAG based methods such as DryadLINQ, this enables the efficient execution of chained pipeline stages. In a chained pipeline, disk I/O and inter-process communication are only performed before the first stage of the chain, and after the last stage of the chain. Apache Spark uses communication barriers to synchronize the execution of each stage [9].

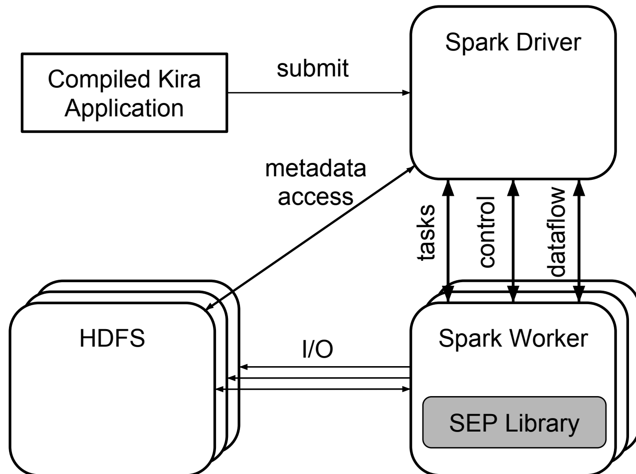

Fig. 1. Overview of Kira's Architecture and Inter-Component Interactions

Another observation from the survey [29] is that many of the documented applications are data-intensive. When these applications are executed in distributed or parallel platforms, reading and writing data between disk and memory dominates the execution time. The data analysis applications supported by Apache Spark and Hadoop are also data intensive. Thus, the data locality optimization in Apache Spark/Hadoop should also be effective to improve the performance of the many-task applications.

Given its rich dataflow pattern support and locality optimization, Apache Spark and related systems are strong candidates for many-task applications.

4 Kira Design and Implementation

When designing the Kira astronomy image processing toolkit, we focused on improving computational performance and I/O cost while providing a flexible programming interface that enables code reuse.

4.1 Architecture Overview

Kira's overall architecture is shown in Figure 1. Each outer box with rounded corners is a process. A process can be a Spark Driver, a Spark Worker, or a HDFS daemon (NameNode or DataNode). Kira runs on top of Spark, which supports a single driver and multiple workers. The SEP library (shaded inner box) is deployed to all worker nodes. The input files are stored in the underlying file system.

To run Kira, we submit the compiled program, the parameters, and library dependencies to the Spark Driver. The Spark Driver manages control flow, dataflow, and task scheduling by coordinating the Spark Workers. The Spark Driver accesses distributed/parallel file systems for metadata and the I/O operations are distributed across the Spark Worker nodes in parallel.

When running a task, workers perform computation by calling out to the SEP library. For Apache Spark's native Scala/Java interface, Kira calls the C library through the Java Native Interface (JNI). With Apache Spark's Python bindings (PySpark), Kira calls the C library through the precompiled Python interface provided by the SEP library.

4.2 Computation

We considered three approaches when implementing the Source Extractor algorithm in Kira:

1) **Reimplement** the Source Extractor algorithm from scratch.
2) Connect existing programs as **monolithic** pieces without changing them.
3) Reorganize the C-based SExtractor implementation to expose a programmable **library** that we call.

While **reimplementing** the functionality of the C SExtractor code using Apache Spark's Scala API would allow us to execute SExtractor in parallel, it would lower the efficiency of the computation and would require a significant reimplementation effort. The **monolithic** approach would not involve a modification to the original executable. While we could integrate with the original codebase at this level, this would lock us in to the hardcoded program logic of the original program. For example, astronomers can improve extraction accuracy by running multiple iterations of source extraction with detected sources being removed from the input image after each iteration. The original SExtractor contains the required functionality for the inner loop of this iterative process, however, the hardcoded logic only allows users to run the source extraction once. In order to not be locked in to the rigid control flow limitations of the *monolithic* model, we instead opt for a **library**-based model. This approach allows us to reuse the legacy code base without sacrificing control-flow flexibility.

4.3 I/O

The Flexible Image Transport System (FITS) [34] format is a widely adopted file format for astronomy images. Each FITS file contains ASCII metadata and binary image data. The FITS format is commonly used by sky surveys, thus Kira must be able to process and export FITS files. In Kira, one of our goals is to leverage the locality information provided by HDFS. When a file is loaded into HDFS, the file is split into blocks that are replicated across machines. When a system such as Apache Spark loads the file, HDFS then provides information about which machines have copies of each block of the file. This allows the scheduler to optimize task placement for data locality.

Kira uses the `SparkContext.binaryFiles()` API. This API loads all files within a directory as a sequence of tuples. Each tuple contains the file object and a byte stream containing the contents of the file. With Apache Spark's Scala interface, we then use the jFITS [35] library to convert these byte streams into the FITS objects that users can transform and compute upon. In PySpark, we use the Astropy [36] Python library.

5 Programming Kira

The Kira API is described in Table 2. Background methods are used to estimate and remove the image background. The Extractor API is used for extracting objects and estimating astrometric and photometric parameters. The Ellipse API offers helper functions for converting between ellipse representations, and for generating masks that are

based on an object's elliptical shape. The `sum_circle()`, `sum_ellipse()`, and `kron_radius()` methods in the extractor category and all methods in the ellipse category perform batch processing, where the input coordinates are passed as a three dimensional array. With Apache Spark's Scala API, we are able to amortize the cost of each Java Native Interface (JNI) call by processing objects in batches.

This API allows us to build a source extractor in Kira that is equivalent to the SEP extractor [26]. Listing 1 contains pseudocode describing how to implement a source extractor using Kira's API. This code uses Apache Spark's `binaryFiles()` method to load input files from persistent storage. We then map over each file to convert the FITS data into a matrix with associated metadata. In the final map stage, we estimates and remove the background from the matrix. Once the background is removed, we then extract the objects from the matrix.

Listing 1. Objects Extraction Logic

```
1 val input_rdd = sparkContext.binaryFiles(src)
2 val mtx_rdd = input_rdd.map(f => load(f))
3 val objects_rdd = mtx_rdd.map(m => {
4   /* mask is a 2-d array with
5    * the same dimensions as m
6    */
7   val mask = null
8   val bkg = new Background(m, mask)
9   val matrix = bkg.subfrom(m)
10  val ex = new Extractor
11  val objects = ex.extract(matrix))
12 })
```

In Listing 2, we demonstrate how the Kira API can be used to perform iterative image refinement. Although the original SExtractor [25] contains all necessary functionality, it is not feasible for users to implement this feature due to the hardcoded program logic. However, since Kira provides library level bindings, it is easy to implement a multi-stage refinement pipeline.

Listing 2. Iterative Objects Extraction Logic

```
1 val input_rdd = sparkContext.binaryFiles(src)
2 val mtx_rdd = input_rdd.map(f=>load(f))
3 val objects_rdd = mtx_rdd.(m => {
4   /*mask is a 2-d array with
5    *the same size of m
6    */
7   var mask = null
8   var ex = new Extractor
9   for(i <- 0 until 5) {
10    var bkg = new Background(m, mask)
11    var matrix = bkg.subfrom(m)
12    var objects = ex.extract(matrix)
13    mask = mask_ellipse(objects)
14  }
15  objects
16 })
```

Listing 2 wraps the source extraction phase from Listing 1 in a loop. This allows us to update the mask used for extraction, which is used to further refine the extraction in subsequent iterations.

6 Tuning Apache Spark

This section discusses how we configure Apache Spark in terms of parallelism and scheduling to make Kira SE more efficiently use EC2 computing resources.

6.1 Parallelism

Apache Spark allows for both thread and process parallelism. By default, Apache Spark makes use of thread-level parallelism by launching a single Java Virtual Machine (JVM) per worker machine. Users then specify the number of threads to launch per worker (typically, one thread per core). However, with Apache Spark's Scala API in Kira SE, neither the jFITS library nor the JNI are thread safe. To work around this, we configured Apache Spark to support process level parallelism by launching a worker instance for each core. This configuration may reduce scalability, as it increases the number of workers the driver manages and can reduce the performance of broadcast operations, as broadcast objects are replicated across workers. However, our experiments with 512 workers in §8 show that Kira's scalability is not severely impacted by worker management or broadcast overhead. The Kira/PySpark implementation is thread safe, and we launch a single Spark Worker per machine when running under PySpark. Each Spark Worker then manages the multiple cores on the machine.

6.2 Scheduling

Apache Spark's task-scheduling policy aims to achieve fairness while maximizing data locality by using delay scheduling [37]. In the context of Apache Spark, a task is an instance of code that runs on a worker and a job consists of many tasks running on many workers. In this scheduling paradigm, if node n has the data needed to run task j, task j will execute on node n if task j would wait less than a threshold time t to start. The policy is tunable through three parameters:

- spark.locality.wait.process
- spark.locality.wait.node
- spark.locality.wait.rack

These parameters allow users to specify how much time a task will wait before being sent to another process, node, or rack. For Kira SE, we have found that data balancing can impact task distribution, leading to node starvation and a reduction in overall performance. Loading a 65 GB (11,150 files) dataset from SDSS Data Release 2 to a 16-node HDFS deployment ideally should result in 699 files on each node. In reality, the number of files on each node varies between 617 and 715. Enforcing locality with longer *spark.locality.wait* time (3000 ms) leads to task distribution imbalance, which makes Kira SE 4.5% slower than running with *spark.locality.wait* set to 0 ms. In practice, we set all *spark.locality.wait* parameters to zero, so that tasks do not wait for locality. This setting effectively avoids starvation and improves the overall time-to-solution.

The root cause of the ineffectiveness of delay scheduling is the size of the input files for the Kira SE tasks. Each input file is ~6 MB, compared to a typical block size of 64/128 MB [19]. Delay scheduling's parameters let users specify how long a task should wait for locality before getting executed elsewhere. This waiting time can be viewed as the expected job completion time difference between executing the task with data locality and without locality. In the Kira SE case, the need for scheduling a task to a node without locality only occurs when there is a possible

TABLE 2
Kira Primitives and Explanation

Group	API	Explanation
Background	makeback()	Builds background from an input image
	backarray()	Returns the background as a 2D array
	subbackarray()	Subtracts a given background from image
Extractor	extract()	Returns objects extracted from the input image
	sum_circle()	Sums data in circular apertures
	sum_ellipse()	Sums data in elliptical apertures
	kron_radius()	Calculate iron radius within an ellipse
Ellipse	ellipse_coeffs()	Converts from ellipse axes and angle to coefficient representations
	ellipse_axes()	Converts from coefficient representations to ellipse axes and angles
	mask_ellipse()	Masks out certain pixels that fall in a given ellipse

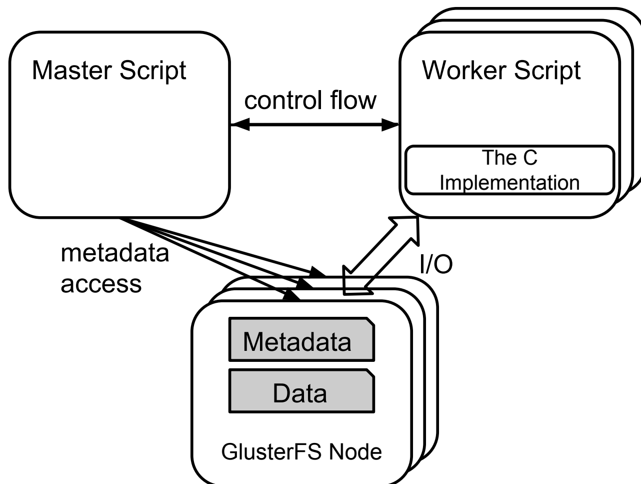

Fig. 2. Overview of a Parallel Version of Source Extractor using HPC Tools

starvation if we continue to enforce the locality. Since our input files are small, the cost of doing a remote fetch is low and thus we should only wait for a short period of time [38]. By experimenting with different waiting time settings, we found that not waiting (i.e., delay=0) delivered the best performance. [1]

7 An HPC Solution

A typical way to parallelize a many-task application such as source extractor is to use a scripting language or MPI to launch multiple tasks concurrently. Figure 2 shows the architecture of such a solution.

All the input files are stored in the shared file system, e.g., GlusterFS or Luster. We use a master script first to read all the input file names, then partition the file names into partitions. After that, the master script informs the worker scripts on each node to process an independent partition of files in parallel in a batch manner.

For GlusterFS, metadata is distributed across the cluster and the metadata query from the master script communicates with all nodes. The file I/O is done by the worker nodes. Note that unlike HDFS, POSIX I/O libraries cannot take advantage of data placement locality during file read/write. While Lustre file I/O is similar to GlusterFS, file system metadata is consolidated onto a small set of metadata servers.

8 Performance

We migrated an earlier version of Kira that used Apache Spark's Scala interface [18] to PySpark for increased uptake in the astronomy community. In the course of this migration, we optimized data layout and took efforts to make zero-copy calls to external libraries, which led to significant performance improvements. To differentiate the two implementations, we refer to them as Kira-SE-v1 and Kira-SE-v2, respectively.

We compare both of the Kira implementations against the HPC solution that runs the C code in the SEP library (referred to as the C version in the following text). Because all implementations use the same SEP library, Kira SE (both versions) perform an identical amount of computation and read and write the exact same files.

To understand Apache Spark's overhead, we first compare Kira SE's performance against the C version on a single machine. Then we fix the problem size and scale Kira SE and the C implementation across a varying number of machines on EC2 to understand the relative scalability of each approach. With a 1 TB dataset from Sloan Digital Sky Survey Data Release 7, we study the difference in performance between Kira SE and the C version for large dataset processing. Finally, we show some interesting results when running the C version on the Edison supercomputer that is deployed at the National Energy Research Scientific Computing Center (NERSC).

In all experiments using Amazon's EC2 service, we use m2.4xlarge instances for spinning disks and r3.2xlarge instance for solid state disks. The m2.4xlarge instance type has eight cores (each running at 2.4GHz), 68 GB RAM, two hard disk drives (HDD), and Gigabit Ethernet. We chose this HDD based configuration to give a fair comparison against the Edison supercomputer, which is backed by HDDs. We also report results from a performance study using nodes with solid state drives (SSDs).

In addition to the batch implementations, we also study a streaming deployment of Kira SE on a 16 m2.4xlarge

1. While this result seems to contradict our result in §8 that states that Apache Spark outperforms the HPC solution due to data locality, it is not contradictory because even with no delay scheduling, 98% of the tasks scheduled have local data. Thus the delay scheduling penalty is not needed in this case. We have traded a 2% decrease in locality for a 4.5% improvement in overall performance.

instance cluster. This study will show Spark Streaming's capability in supporting near real-time processing as required by surveys such as LSST.

Software configurations are described in the following experiments. If not stated otherwise, we run each experiment three times, and present the average. Error bars in the figures are the standard deviation of the measurements.

8.1 Single Machine Performance

We begin by describing a set of scale-up experiments on a single machine. The purpose of these experiments is to understand Kira SE's relative overhead compared to that of simply running the SEP C code on a single node. Note that, the two Kira versions and the C implementation run the same SEP C code. Kira-SE-v1 and Kira-SE-v2 call the C code through Java Native Interface and the Cython interface, respectively. We also want to identify the resource requirements (computation or disk I/O) that dominate Kira SE's performance. For both Kira SE and the C implementation, we store data locally in an ext4 file system [39].

For this experiment, we use a 12GB dataset from the SDSS DR2 survey. The dataset contains 2310 image files where each image is ∼6MB.

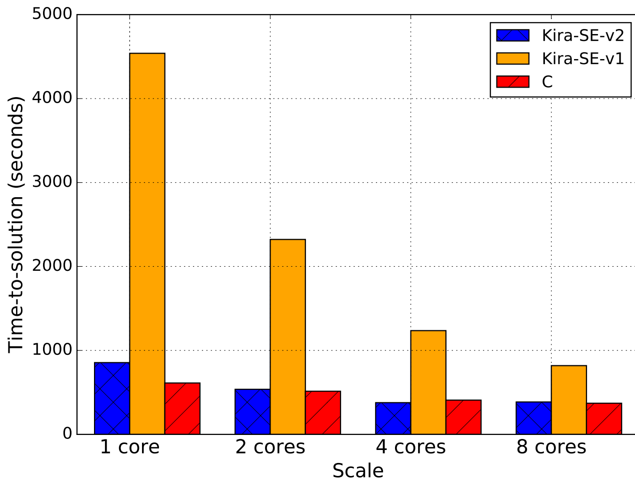

Fig. 3. Single-Node Scale-Up Performance Comparison between Kira SE and the C Version (Lower is Better)

Figure 3 shows the running time of 3 different deployments of the Source Extractor on a single m2.4xlarge machine as it is scaled from 1 to 8 cores. The figure shows that C and Kira-SE-v2 have similar performance while Kira-SE-v1 is significantly slower, although the difference decreases as more cores are used. The runtime of the C implementation is dominated by disk I/O and the runtime improves by 65% when scaling from a single core to eight cores. Although Kira-SE-v1 is 7.4× slower than the C implementation on a single core, Kira-SE-v1 is only 2.2× slower than the performance of the C version when using all eight cores on the node. The performance slowdown of Kira-SE-v1 compared to the C implementation is mainly caused by Java Virtual Machine and JNI overhead when calling to the C libraries. With 8 cores on the node, the performance of Kira-SE-v1 is dominated by CPU. Kira-SE-v2 is 39.7% slower than the C implementation with a single core, while this performance gap shrinks as the core count increases. The performance of Kira-SE-v2 converges to the C performance beyond four cores. The performance improvement of Kira-SE-v2 over Kira-SE-v1 is attributable to using the SEP-native data layout in our PySpark application as well as the reduction in data copying between SEP library and the core C library. A further performance breakdown of Kira-SE-v2 reveals that the the library calling and data copying overhead of SEP averages 108 msec. That is, by allowing SEP to operate in-place on the images without an expensive pre-process and copy step, we are able to improve Kira's performance dramatically.

We also profiled the C implementation with both warm and cold file system caches. When running with a cold cache, the job completed in 371 seconds while the job completed in 83 seconds when running with a warm cache. This indicates that 78% of job execution time is consumed by I/O (reading and writing data between local disk and memory). Since the C implementation of SExtractor is dominated by disk I/O, we believe that it is representative of a data intensive application.

8.2 Scale-Out Performance

Next, we wanted to understand the strong scaling performance of both Kira SE and the C implementation. Although Kira-SE-v1 has 2.2× worse performance than the C implementation when running on a single machine, we expect that Kira-SE-v1 will achieve better performance at scale due to disk locality. We expect that this will allow Kira-SE-v1 to outperform the C implementation on large clusters. Kira-SE-v2 combines improved locality with computational requirements that are similar to that of the C implementation, so we expect Kira-SE-v2 to be faster than the C implementation at all scales.

We use a 65GB dataset from the SDSS DR7 that comprises 11,150 image files. Kira SE was configured to use HDFS as a storage system, while the C version used GlusterFS. Both HDFS and GlusterFS are configured with a replication factor of two.

Figure 4 compares the performance of Kira-SE-v1, Kira-SE-v2, and the C version across multiple compute nodes(shown with log scale). Kira-SE-v1 is 2.7× slower than the C version on eight cores. However, the gap between the two implementations decreases as we scale up. On 256 cores and 512 cores, Kira-SE-v1 is respectively 5.6% and 22.4% faster than the C version. The Kira-SE-v2 shows a more significant improvement over the C implementation with a speedup of 2.2×–3.1× across scales. Both Kira-SE-v1 and Kira-SE-v2 achieve near linear scalability.

The fundamental driver of Kira SE's linear scalability is its consistent local disk hit ratio, which is the ratio between the number of tasks that access the input file on the local disk (rather than having to go across the network) and total number of tasks. Taking Kira-SE-v2 as an example, Apache Spark and HDFS optimize for data locality during scheduling and achieve a hit ratio around 98% with a small standard deviation (around 0.2%), as shown in Figure 5. In contrast, the C implementation's estimated locality hit ratio decreases in half as the cluster size doubles.

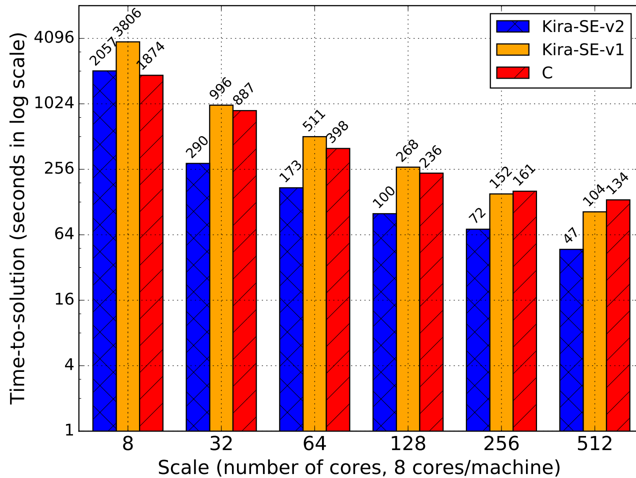

Fig. 4. Scale-Out Performance Comparison between Kira SE and the C Version in Logarithmic Scale (Lower is Better)

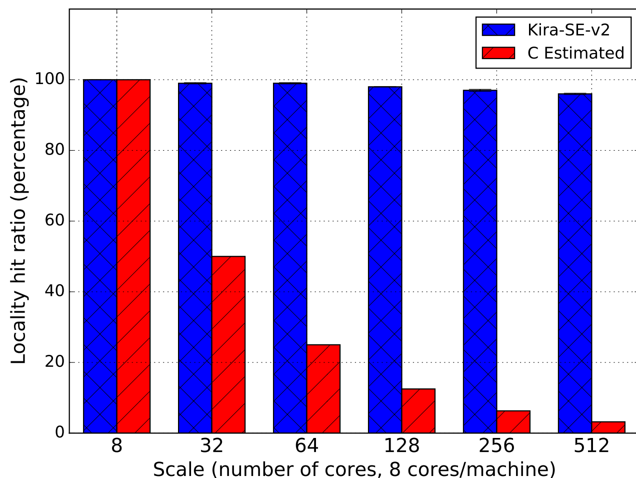

Fig. 5. Locality Hit Ratio (Higher is Better)

In general, a shared file system can be configured in many ways to achieve better availability, performance, and resilience. To understand the impact of the shared file system configuration, we compare the performance of Kira SE (time-to-solution) against four configurations of GlusterFS. The four configurations are *distributed*, *replicated*, *striped*, and *striped replicated*. Table 3 explains the data layout of each configuration. When possible, we set the replication and striping factors to two. GlusterFS manages metadata in a distributed manner by spreading metadata across all available nodes with a hash function. This allows the clients to deterministically know the location of the metadata of a given file name in the cluster.

We evaluate these configurations using the same dataset as the scale-out experiment. We select 128 cores and 256 cores as the target scale since it is the transition point in Figure 4 where Kira-SE-v1 begins to run faster than the C version. As stated previously in §7, the C version performs a two-step process. The first step collects and partitions all file paths. We refer to this step as metadata overhead. The processing step occurs next, and is where each node processes its own partition. Figure 6 compares the performance of Kira-SE-v1, Kira-SE-v2, and the C version with profiled metadata overhead.

TABLE 3
GlusterFS Configuration Modes and Data Layout

Conf Mode	Data Layout
distributed	files are distributed to all nodes without replication
replicated	files are distributed to all nodes with a number of replicas specified by the user
striped	files are partitioned into a pre-defined number of stripes then distributed to all nodes without replication
striped replicated	files are partitioned into a pre-defined number of stripes and the stripes are distributed to all nodes with a number of replicas specified by the user

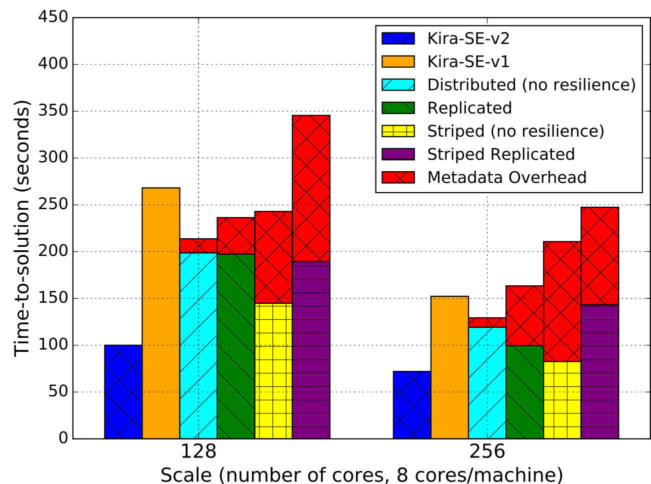

Fig. 6. Kira SE Performance Compared to All GlusterFS Configurations (Lower is Better)

The C version running in the *distributed* mode outperforms Kira-SE-v1 at both scales. However, the *distributed* mode is not practical in a cloud environment since it has no replication or any other resilience mechanism. Replicating or striping files introduces extra metadata overhead when we compare the *replicated* mode to the *distributed* mode.

Another observation is that striping will further slow down metadata processing, whereas the processing part takes less time than the *distributed* mode for both scales due to the doubled probability of accessing a file stripe (with the striping factor of two) in local disk. Since the input files are ∼6MB each, and are always processed by a single task, the *replicated* mode should be preferred to the *striped replicated* mode.

When running on 256 cores, Kira-SE-v1 outperforms all GlusterFS configurations except for the (impractical) *distributed* mode. When compared to the *distributed* mode, Kira-SE-v1 delivers comparable performance, as it is 18% slower. In our experiments with the 1TB dataset in Section 8.3.1, Kira-SE-v1 outperforms the *distributed* mode.

As expected, Kira-SE-v2 outperforms the *distributed*

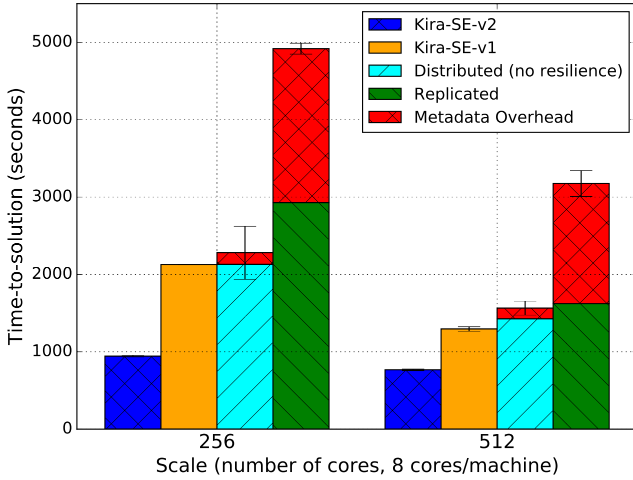

Fig. 7. Kira SE Performance with 1TB Input Compared to the C Version Running on GlusterFS on EC2 (Lower is Better)

mode, in this case by 2.1× and 1.8× on 128 cores and 256 cores, respectively.

8.3 1TB Dataset Performance

Having examined the performance of the different approaches using a relatively small (65GB) data set, we now investigate their performance when there is significantly more data to process. We select a 1TB dataset from the SDSS DR7 survey, which is comprised of 176,938 image files. With this experiment, we seek to answer the following questions:

- Can Kira scale to process a 1TB dataset?
- What is the relative performance of Kira compared to the HPC version on the Amazon EC2 cloud?
- How does Kira SE performance compare to the HPC version on a supercomputer?

8.3.1 Cloud

We configure GlusterFS in the *replicated* and *distributed* modes and compare Kira-SE-v1's and Kira-SE-v2's performance against the C implementation. A detailed breakdown of the performance is shown in Figure 7. In this experiment, Kira-SE-v1 runs $1.1\times$ and $1.3\times$ faster than the C version running on top of GlusterFS configured in *distributed* mode on 256 cores and 512 cores respectively. Compared to the more practical *replicated* configuration of GlusterFS, Kira-SE-v1 is $2.3\times$ and $2.5\times$ faster. On the other hand, Kira-SE-v2 runs $2.5\times$ and $2.0\times$ faster than the impractical *distributed* mode on 256 cores and 512 cores, respectively. Kira-SE-v2 is also $4.1\times \sim 5.2\times$ faster than the *replicated* mode. The C version in *distributed* mode is slower than both Kira SE implementations due to the lack of the locality notion in the HPC solution presented in §7. The C version in *replicated* mode slows down $2.2\times$ than that in *distributed* mode because the directory metadata query is dramatically slower ($13.4\times$), and the additional replica for each output file and associated metadata update introduces a slowdown of $1.4\times$.

Compared to the experiment with the 65GB dataset in Section 8.2, Kira-SE-v2 processes $15.9\times$ more data in $16.3\times$ more time. If we discount the Apache Spark startup time, we can see that Kira-SE-v2 scales linearly in relation to the data size.

The overall throughput of Kira-SE-v2 is 1,335 MB/second, which is $4.0\times$ greater than necessary to support the upcoming Large Synoptic Survey Telescope (LSST), as discussed in Section 2.3. This high throughput enables real-time image processing.

8.3.2 Supercomputer Performance

Many astronomers have access to supercomputers and believe that supercomputers outperform commodity clusters for data-intensive applications. To examine this belief, we compare Kira-SE-v1 and Kira-SE-v2 on the Amazon cloud versus the performance of the C version running on the NERSC Edison supercomputer, a Cray XC 30 System. On the supercomputer we use the Lustre file system, which provides a peak throughput of 48GB/s. Each compute node of Edison is equipped with a 24-core Ivy Bridge processor, with a 2.4GHz clock rate. This is comparable to the CPU speed of the Amazon EC2 m2.4xlarge instance (eight vCPUs of Intel Xeon E5-2665, each running at 2.4GHz). The experiments on Edison run on 21 nodes (a total of 504 cores) while Kira SE uses 64 nodes (512 cores) on EC2. Figure 8 shows the measurements.

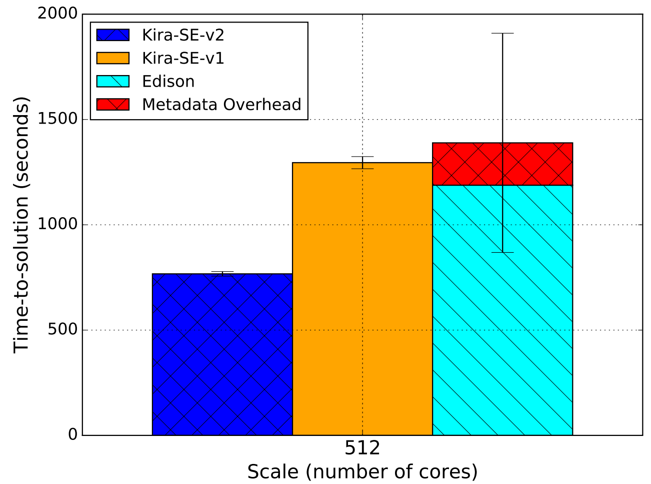

Fig. 8. Kira SE Performance with 1TB Input Compared to the C Version Running on NERSC Edison Supercomputer (Lower is Better)

Kira-SE-v1 delivers performance comparable to that of the C version on Edison. While Kira-SE-v2 achieves a $1.8\times$ speedup compared to that of Edison performance. During the experiments, we observed that the C version performance varies significantly with an average time-to-solution of 1,388.9 seconds and a standard deviation of 520.5 seconds. These results clearly fall into two classes. The first class has an average time-to-solution of 937.8 seconds with a standard deviation of 69.5 seconds. The second class has an average time-to-solution of 1840.1 seconds with a standard deviation of 248.7 seconds. A further analysis shows that we are only using 0.4% of the computing resources of the Edison machine. In the first class, the sustained I/O bandwidth is 1.0 GB/s, which is 2.1% of the I/O bandwidth available on the file system. While in the second class, the sustained

I/O bandwidth is down to 0.5GB/s. Since the Edison cluster scheduler only schedules a single job per compute node, computing resources are completely isolated. Thus, we can reason that it is the I/O network resource or the file system that causes the performance variance.

On the other hand, Kira-SE-v2 is able to speedup the performance by a factor of 1.8 compared to the C performance on the Edison Supercomputer with an average time-to-solution of 767.4 seconds and a standard deviation of 10.9 seconds. The stable performance of both versions of Kira SE can be attributed to exploiting data locality: both the Kira implementations move the I/O from network to local disk access, which gives a higher I/O bandwidth as well as better task-to-task isolation.

8.4 Solid State Disk Performance

In §8.1, we stated that the Kira-SE-v2 performance is dominated by disk I/O. By running with high bandwidth SSDs, we can quantitatively evaluate the potential performance improvement that increased I/O bandwidth can provide.

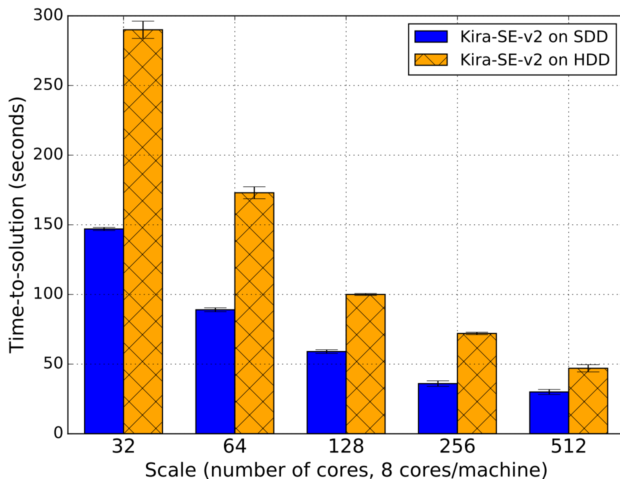

Fig. 9. Kira-SE-v2 Performance with 65GB Input Using Solid State Disk (SSD) Compared to Spinning Disks (HDD) (Lower is Better)

Figure 9 shows that using SSDs allows Kira-SE-v2 to run $1.6\times \sim 2.0\times$ faster than spinning disks across various cluster scales. The measurement with the 1TB dataset shows a $1.9\times$ and $2.0\times$ speedup on 256 cores and 512 cores, respectively. Using SSDs boosts the sustained processing rate on 512 cores from 1,335 MB/second to 2,723 MB/second, which is about $8.3\times$ higher than the LSST real time processing requirement.

8.5 Spark Streaming

In §2.3, we stated that the LSST sky survey requires near real-time processing with a maximum latency of 60 seconds [16], and that the sustained data generation rate of LSST is ∼330 MB/s [27]. Though executing Kira in the batch mode periodically can meet the latency and throughput requirements of such an application, doing so in a continuous fashion would require external tools to manage the Kira invocation and handle fault-tolerance across executions. For example, results from finished batch executions are lost due to a failure, Apache Spark can not recover using its lineage since the lost results are out of scope once the spark execution finishes. In contrast, deploying Kira with Spark Streaming will make continuous processing fully automatic, and also Spark Streaming has the ability to efficiently recover lost data from failures. Thus, in this section, we examine the deployment of Kira using Spark Streaming. Also, we would like to find out more generally the latency and throughput bounds of Kira with Spark Streaming to understand the applicability of Spark Streaming for providing low latency processing in a scientific environment.

8.5.1 Spark Streaming Overview

Spark Streaming's D-Stream abstraction [17] partitions data streams into mini-batches, then applies a sequence of transformations on the mini-batches. Users can specify the "batch interval", which is the time period that defines the size of each mini-batch. At the end of an interval, processing is begun on the data that arrived during that interval and a new interval is started. This process is depicted in Figure 10.

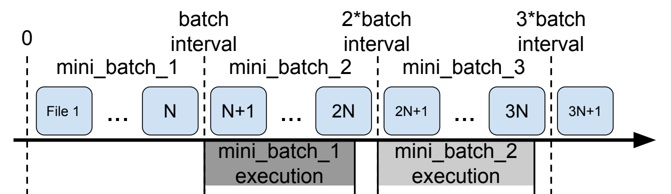

Fig. 10. Spark Streaming Operations along Time Line with Batch Interval Longer than per Batch Execution Time

Technically, Spark streaming can monitor an HDFS directory for new files and process the new files periodically. The Python interface of the Spark Streaming 1.6.0 release supports both text files and fixed-length binary records. We implement a new streaming interface called `SparkContext.binaryFileStream()` to read the FITS files as binary streams. Then, the source extraction procedure can be applied to the data by calling Kira APIs.

We evaluate the streaming deployment based on its processing latency. We define the processing latency as the time from when an interval starts until the final file of that interval is processed. As shown in Figure 10, the processing latency is the sum of two parts.

$$proc_latency = batch_interval + exec_time \qquad (1)$$

The first part is its waiting time inside the batch interval. The second part is the execution time for the mini-batch. If the the batch interval is shorter than the average execution time, then it is an infeasible deployment. This is because the file processing latency would increase as time proceeds.

8.5.2 Experiment Results

In this experiment, we use a fixed cluster of 16 m2.4xlarge instances (128 cores in total) and stream the 65 GB dataset to an HDFS directory at the rate of 780 MB/s (∼128 files/s). We vary the batch interval parameter in the range of {1, 2, 4, 8} seconds and measure the execution time of each mini-batch. Then we examine the feasibility of these batch interval values by comparing to average execution time.

Figure 11 shows the measured execution time with a varying batch interval of {1, 2, 4, 8} seconds. With all four batch interval settings, we see that the execution time of the first mini-batch is higher than the other mini-batches due to startup cost of loading libraries into memory. After the first batch, the execution time is fairly stable. As would be expected, decreasing the batch interval by 2 approximately halves the execution time, until the step from the 2 second batch interval to 1 second.

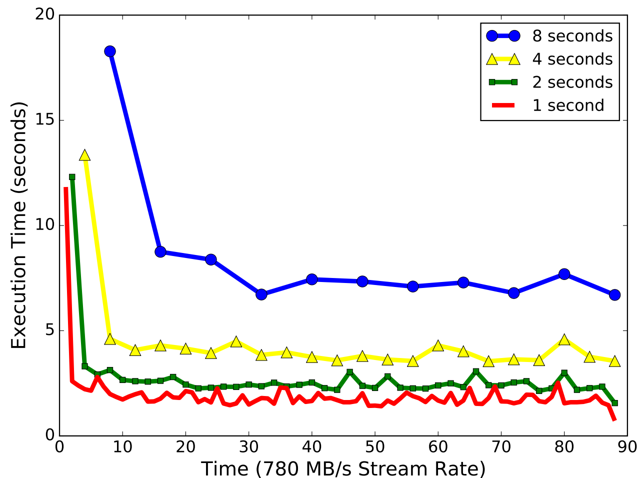

Fig. 11. Per Mini-batch Execution Time of Kira-SE-v2 Enabled by Spark Streaming with Varying Batch Intervals

Figure 12 shows the scatter plot of the batch interval and the average execution time. We classify the space into "feasible" and "infeasible" based on the difference of batch interval and average execution time. In the lightly shaded area, the execution time of a mini-batch is less than the batch interval. So points that fall in this area represent feasible Spark Streaming deployments. In contrast, the points that fall in the dark shaded area indicate infeasible deployments, as the execution time is on average longer than the batch interval.

Among the four batch interval values test, the batch intervals of 4 and 8 seconds result in feasible deployments. The processing latencies, calculated using Equation 1, for them are 7.9 seconds and 15.4 seconds, respectively. Both deployments are able to keep up with the 780 MB/s data streaming rate.

This performance on 16 m2.4xlarge instances easily meets LSST's requirements of near real-time processing with a maximum 60 seconds latency and the sustained data generation rate at $\sim$330 MB/s. Our results show that Spark Streaming has significant performance headroom in the case so that it could support even more stringent requirements.

9 Related Work

Many systems have tackled the problem of executing single process programs in parallel across large compute clusters. This includes workflow systems such as HTCondor, and ad hoc Hadoop and MPI based approaches.

In a workflow system, programmers can easily connect serial/parallel programs by specifying dependencies

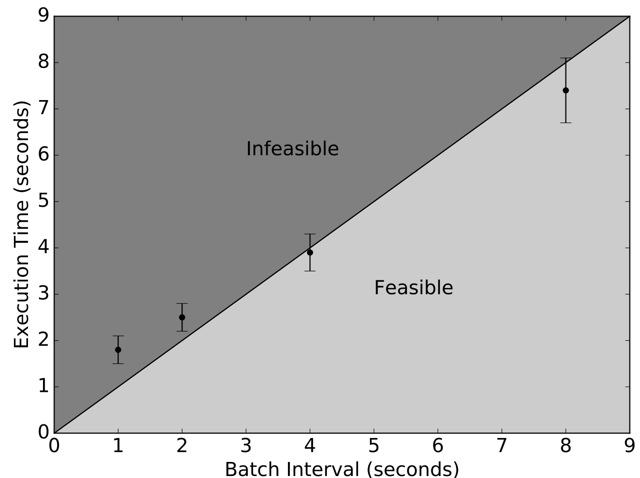

Fig. 12. Batch Interval Setting Feasibility Classification, Error Bars are Standard Deviation of Execution Time.

between tasks and files. These systems do not require any modifications to the original code base. Workflow systems provide a flexible data management and task execution scheme that can be applied to a broad range of applications, but at the cost of programming flexibility.

Researchers have used the Hadoop MapReduce [7] system to parallelize tasks using a map-reduce data model. A variety of scientific applications have been parallelized using Hadoop such as CloudBLAST [8]. Although Hadoop exposes many convenient abstractions, it is difficult to express the application with the restrictive map-reduce API [31] and Hadoop's disk based model makes iterative/pipelined tasks expensive.

MPI has also been used to parallelize a diverse range of workloads. There are MPI-based parallel implementations of astronomy image mosaicing applications (Montage [1]) and sequence alignment and search toolkits (mpiBLAST [40]) applications. As an execution system, MPI has two significant drawbacks. First, to implement a many-task application on top of MPI, a user must develop a custom C wrapper for the application and a custom message-passing approach for communicating between nodes. In practice, the communication stages are critical for performance, which means that the dataflow management scheme must be tailored to the application and hand tuned. Additionally, MPI does not provide fault tolerance, which is problematic when running a long lived application across many (possibly) unreliable nodes.

Traditionally, distributed workflow systems are run on top of a shared file system. Shared file systems (e.g., Lustre [21], and GlusterFS [20]) are commonly used because they are compatible with the POSIX standard and offer a shared namespace across all nodes. However, shared file systems do not expose file locality to workflow systems, thus making suboptimal use of local disks on the compute nodes when possible. Most tools in the Hadoop ecosystem use HDFS [19]). HDFS provides a shared namespace, but is not POSIX compliant. Unlike traditional server-based shared file systems, HDFS uses the disks on the compute nodes which enables data locality on filesystem access.

10 Future Work

Kira is currently available as an alpha release (https://github.com/BIDS/Kira), and we are planning to migrate Kira SE into a larger project for supernovae detection.

By adding processing kernels including image reprojection and image co-addition, Kira will be useful as an end-to-end astronomy image analysis pipeline. We will use this end-to-end pipeline to continue evaluating the use of Apache Spark as a conduit for many-task dataflow pipelines by comparing against the equivalent C implementation. With this system, we will try to determine which data intensive scientific applications execute most efficiently using "big data" software architectures on commodity clusters, rather than using HPC software methods on supercomputers. From this, we hope to obtain insights that can drive the development of novel computing infrastructure for many-task scientific applications.

11 Conclusion

In this paper, we investigated the idea of leveraging the modern big data platform for many-task scientific applications. Specifically, we built Kira (https://github.com/BIDS/Kira), a flexible, scalable, and performant astronomy image processing toolkit using Apache Spark running on Amazon EC2 Cloud. We also presented the real world Kira Source Extractor application, and use this application to study the programming flexibility, dataflow richness, scheduling capacity and performance of the surrounding ecosystem.

The Kira SE implementation demonstrates linear scalability with both increasing cluster and data size. Due to its superior data locality, our Spark-based implementation achieves a speedup of $2.2\times$–$4.1\times$ over the equivalent C implementation running on GlusterFS. Kira SE's performance scales near linearly with the dataset and cluster size. Specifically, Kira SE processes the 1TB SSDS DR7 dataset (176,938 tasks) $4.1\times$ faster than C over GlusterFS when running on a cluster of 64 m2.4xlarge Amazon EC2 instances. Kira SE also achieves a $1.8\times$ speedup compared to the C version running on the NERSC Edison supercomputer. Using SSDs can boost the Kira SE performance by a factor of two compared to the performance with spinning disks. By leveraging the Spark Streaming module, we were able to deploy Kira SE as a streaming application. On a 128 core cluster, Kira SE with Spark Streaming can achieve a second-scale processing latency and a sustained throughput of $\sim$800 MB/s. All these measurements indicate that using Apache Spark can improve the performance of data intensive scientific applications.

We also demonstrated that Apache Spark can integrate with a pre-existing astronomy image processing library. This allows users to reuse existing source code to build new analysis pipelines. We believe that Apache Spark's flexible programming interface, rich dataflow support, task scheduling capacity, locality optimization, and built-in support for fault tolerance make Apache Spark a strong candidate to support many-task scientific applications. Apache Spark is one (popular) example of a Big Data platform. We learned that leveraging such a platform would enable scientists to benefit from the rapid pace of innovation and large range of systems and technologies that are being driven by widespread interest in Big Data analytics.

Acknowledgments

This research is supported in part by NSF CISE Expeditions Award CCF-1139158, DOE Award SN10040 DE-SC0012463, and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, IBM, SAP, The Thomas and Stacey Siebel Foundation, Adatao, Adobe, Apple, Inc., Blue Goji, Bosch, C3Energy, Cisco, Cray, Cloudera, EMC2, Ericsson, Facebook, Guavus, HP, Huawei, Informatica, Intel, Microsoft, NetApp, Pivotal, Samsung, Schlumberger, Splunk, Virdata and VMware. Author Frank Austin Nothaft is supported by a National Science Foundation Graduate Research Fellowship.

This research is also supported in part by the Gordon and Betty Moore Foundation and the Alfred P. Sloan Foundation together through the Moore-Sloan Data Science Environment program.

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.

References

[1] J. C. Jacob, D. S. Katz, G. B. Berriman, J. C. Good, A. C. Laity, E. Deelman, C. Kesselman, G. Singh, M.-H. Su, T. A. Prince, and R. Williams, "Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking," *International Journal of Computational Science and Engineering*, vol. 4, no. 2, pp. 73–87, 2009.

[2] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology*, vol. 215, no. 3, pp. 403–410, 1990.

[3] J. Ekanayake, S. Pallickara, and G. Fox, "MapReduce for data intensive scientific analyses," in *Proceedings of the IEEE International Conference on eScience (eScience '08)*, 2008, pp. 277–284.

[4] I. Raicu, Z. Zhang, M. Wilde, I. Foster, P. Beckman, K. Iskra, and B. Clifford, "Toward loosely coupled programming on petascale systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '08)*. IEEE Press, 2008.

[5] M. J. Litzkow, M. Livny, and M. W. Mutka, "Condor—a hunter of idle workstations," in *Proceedings of the International Conference on Distributed Computing Systems (ICDCS '88)*. IEEE, 1988, pp. 104–111.

[6] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, 1996.

[7] Apache, "Apache Hadoop," http://hadoop.apache.org/.

[8] A. Matsunaga, M. Tsugawa, and J. Fortes, "CloudBLAST: Combining MapReduce and virtualization on distributed resources for bioinformatics applications," in *Proceedings of the IEEE International Conference on eScience (eScience '08)*, 2008, pp. 222 –229.

[9] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI '12)*. USENIX Association, 2012, pp. 2–2.

[10] 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 the USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, 2014.

[11] E. R. Sparks, A. Talwalkar, V. Smith, J. Kottalam, X. Pan, J. Gonzalez, M. J. Franklin, M. I. Jordan, and T. Kraska, "MLI: An API for distributed machine learning," in *Proceedings of the IEEE International Conference on Data Mining (ICDM '13)*. IEEE, 2013, pp. 1187–1192.

[12] J. Freeman, N. Vladimirov, T. Kawashima, Y. Mu, N. J. Sofroniew, D. V. Bennett, J. Rosen, C.-T. Yang, L. L. Looger, and M. B. Ahrens, "Mapping brain activity at scale with cluster computing," *Nature Methods*, vol. 11, no. 9, pp. 941–950, 2014.

[13] M. Massie, F. Nothaft, C. Hartl, C. Kozanitis, A. Schumacher, A. D. Joseph, and D. A. Patterson, "ADAM: Genomics formats and processing patterns for cloud scale computing," UCB/EECS-2013-207, EECS Department, University of California, Berkeley, Tech. Rep., 2013.

[14] F. A. Nothaft, M. Massie, T. Danford, Z. Zhang, U. Laserson, C. Yeksigian, J. Kottalam, A. Ahuja, J. Hammerbacher, M. Linderman, M. Franklin, A. Joseph, and D. Patterson, "Rethinking data-intensive science using scalable analytics systems," in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*, 2015.

[15] Z. Ivezic, J. Tyson, E. Acosta, R. Allsman, S. Anderson, J. Andrew, R. Angel, T. Axelrod, J. Barr, A. Becker *et al.*, "LSST: From science drivers to reference design and anticipated data products," *arXiv preprint arXiv:0805.2366*, 2008.

[16] J. Becla, A. Hanushevsky, S. Nikolaev, G. Abdulla, A. Szalay, M. Nieto-Santisteban, A. Thakar, and J. Gray, "Designing a multi-petabyte database for lsst," in *SPIE Astronomical Telescopes+ Instrumentation*. International Society for Optics and Photonics, 2006, pp. 62 700R–62 700R.

[17] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP '13)*. New York, NY, USA: ACM.

[18] Z. Zhang, K. Barbary, F. A. Nothaft, E. Sparks, O. Zahn, M. J. Franklin, D. A. Patterson, and S. Perlmutter, "Scientific computing meets big data technology: An astronomy use case," in *Proceedings of the IEEE International Conference on Big Data (BigData '15)*. IEEE, 2015, pp. 918–927.

[19] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Proceedings of the IEEE Symposium on Mass Storage Systems and Technologies (MSST '10)*. IEEE, 2010, pp. 1–10.

[20] A. Davies and A. Orsaria, "Scale out with GlusterFS," *Linux Journal*, vol. 2013, no. 235, p. 1, 2013.

[21] S. Donovan, G. Huizenga, A. J. Hutton, C. C. Ross, M. K. Petersen, and P. Schwan, "Lustre: Building a file system for 1000-node clusters," in *Proceedings of the Linux Symposium*, 2003.

[22] D. G. York, J. Adelman, J. E. Anderson Jr, S. F. Anderson, J. Annis, N. A. Bahcall, J. Bakken, R. Barkhouser, S. Bastian, E. Berman *et al.*, "The Sloan Digital Sky Survey: Technical summary," *The Astronomical Journal*, vol. 120, no. 3, p. 1579, 2000.

[23] Z. Zhang, K. Barbary, F. A. Nothaft, E. Sparks, O. Zahn, M. J. Franklin, D. A. Patterson, and S. Perlmutter, "Scientific computing meets big data technology: An astronomy use case," in *Big Data (Big Data), 2015 IEEE International Conference on*. IEEE, 2015, pp. 918–927.

[24] D. E. S. Collaboration *et al.*, "The dark energy survey," *arXiv preprint astro-ph/0510346*, 2005.

[25] E. Bertin and S. Arnouts, "SExtractor: Software for source extraction," *Astronomy and Astrophysics Supplement Series*, vol. 117, no. 2, pp. 393–404, 1996.

[26] K. Barbary, K. Boone, and C. Deil, "sep: v0.3.0," Feb. 2015, https://github.com/kbarbary/sep.

[27] "LSST Software: The Soul of the Machine," http://lsst.org/tour.

[28] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the USENIX Conference on Hot Topics in Cloud Computing (HotCloud '10)*, 2010, p. 10.

[29] D. S. Katz, T. Armstrong, Z. Zhang, M. Wilde, and J. Wozniak, "Many task computing and Blue Waters," Computation Institute, University of Chicago, Tech. Rep. CI-TR-13-0911, November 2011.

[30] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)*, 2004, pp. 137–150.

[31] D. DeWitt and M. Stonebraker, "MapReduce: A major step backwards," *The Database Column*, vol. 1, 2008.

[32] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum, "FlumeJava: Easy, efficient data-parallel pipelines," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*, ser. PLDI '10. New York, NY, USA: ACM, 2010, pp. 363–375. [Online]. Available: http://doi.acm.org/10.1145/1806596.1806638

[33] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey, "DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language." in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, vol. 8, 2008, pp. 1–14.

[34] D. C. Wells, E. Greisen, and R. Harten, "FITS—a flexible image transport system," *Astronomy and Astrophysics Supplement Series*, vol. 44, p. 363, 1981.

[35] "Java Library for access to FITS files," http://www.eso.org/ pgrosbol/fits_java/jfits.html.

[36] Astropy Collaboration, T. P. Robitaille, E. J. Tollerud, P. Greenfield, M. Droettboom, E. Bray, T. Aldcroft, M. Davis, A. Ginsburg, A. M. Price-Whelan, W. E. Kerzendorf, A. Conley, N. Crighton, K. Barbary, D. Muna, H. Ferguson, F. Grollier, M. M. Parikh, P. H. Nair, H. M. Unther, C. Deil, J. Woillez, S. Conseil, R. Kramer, J. E. H. Turner, L. Singer, R. Fox, B. A. Weaver, V. Zabalza, Z. I. Edwards, K. Azalee Bostroem, D. J. Burke, A. R. Casey, S. M. Crawford, N. Dencheva, J. Ely, T. Jenness, K. Labrie, P. L. Lim, F. Pierfederici, A. Pontzen, A. Ptak, B. Refsdal, M. Servillat, and O. Streicher, "Astropy: A community Python package for astronomy," *Astronomy & Astrophysics*, vol. 558, p. A33, Oct. 2013.

[37] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of the European Conference on Computer Systems (EuroSys '10)*. ACM, 2010, pp. 265–278.

[38] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Disk-locality in datacenter computing considered irrelevant," in *Proceedings of the USENIX Workshop on Hot Topics in Operating Systems (HotOS '11)*. HotOS, 2011.

[39] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: current status and future plans," in *Proceedings of the Linux symposium*, vol. 2. Citeseer, 2007, pp. 21–33.

[40] H. Lin, P. Balaji, R. Poole, C. Sosa, X. Ma, and W.-c. Feng, "Massively parallel genomic sequence search on the Blue Gene/P architecture," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '08)*. IEEE, 2008, pp. 1–11.

Zhao Zhang Zhao Zhang is a postdoc researcher in AMPLab and a data science fellow in BIDS(Berkeley Institute for Data Science) at University of California, Berkeley. His research is to enable data intensive scientific applications on distributed and parallel systems. Zhao received his Ph.D from the Department of Computer Science, University of Chicago in June 2014.

Kyle Barbary Kyle Barbary is a Cosmology Data Science Fellow in the Berkeley Center for Cosmological Physics and the Berkeley Institute for Data Science. He studies cosmology using Type Ia supernovae as part of the Nearby Supernova Factory experiment and develops software tools for analyzing astronomical data in Python and Julia.

Frank Austin Nothaft Frank Austin Nothaft is a PhD student in Computer Science at UC Berkeley. Frank holds a Masters of Science in Computer Science from UC Berkeley, and a Bachelors of Science with Honors in Electrical Engineering from Stanford University. Prior to joining UC Berkeley, Frank worked at Broadcom Corporation on design automation techniques for industrial scale wireless communication chips.

Evan R. Sparks Evan R. Sparks is a Ph.D. student in Computer Science at UC Berkeley. He holds a Masters of Science in Computer Science from UC Berkeley and a Bachelor of Arts in Computer Science with High Honors from Dartmouth College. Prior to joining UC Berkeley, Evan worked in Quantitative Asset Management at MDT Advisers and as an Engineer at the Web Intelligence firm Recorded Future.

Oliver Zahn Oliver Zahn is a computational/theoretical astrophysicist and the Executive Director of Berkeley Center for Cosmological Physics. Trained as a multipurpose cosmologist, his research program tries to advance the understanding of the origin and evolution of structure in the Universe by applying a variety of analytical and numerical methods to complementary astrophysical observables.

Michael J. Franklin Michael J. Franklin is the Liew Family Chair of the Department of Computer Science at University of Chicago. He served as the Thomas M. Siebel Professor of Computer Science and the Director of the Algorithms, Machines, and People Laboratory (AMPLab) at UC Berkeley. He is a Fellow of the ACM and a two-time winner of the ACM SIGMOD ”Test of Time” award.

David A. Patterson David Patterson is the E.H. and M.E. Pardee Professor of Computer Science at UC Berkeley and is a past president of ACM, and past chair of the UC Berkeley Computer Science Department. David has been awarded the IEEE von Neumann Medal, the IEEE Johnson Storage Award, the SIGMOD Test of Time award, the ACM-IEEE Eckert-Mauchly Award, and the Katayanagi Prize. He was also elected to both AAAS societies, the National Academy of Engineering, the National Academy of Sciences, the Silicon Valley Engineering Hall of Fame, and to be a Fellow of the Computer History Museum, the IEEE, and the ACM.

Saul Perlmutter Saul Perlmutter is an astrophysicist at the Lawrence Berkeley National Laboratory, a professor of physics at UC Berkeley, and the director of Berkeley Institute for Data Science. He is a member of the American Academy of Arts & Sciences, a Fellow of the American Association for the Advancement of Science, and a member of the National Academy of Sciences. Along with Brian P. Schmidt and Adam Riess, Saul shared the 2006 Shaw Prize in Astronomy, the 2011 Nobel Prize in Physics, and the 2015 Breakthrough Prize in Fundamental Physics for the discovery of the accelerated expansion of the universe.