Applications of Apache Spark™ for Numerical Simulation

Jordan Koeller[1], **Mark Lewis**[2], **David Pooley**[3]
One Trinity Place, San Antonio, Texas, USA. 78212
[1]Trinity University.
[2]Trinity University Department of Computer Science.
[3]Trinity University Department of Physics and Astronomy.

Abstract— *We analyze the viability of Apache Spark™ for numerical simulation applications. To simulate gravitational lensing, we ray-trace approximately* 10^8 *rays through a galaxy, followed by a spatial query. For optimal performance, we implement custom partitioning schemes in Spark™ and explore Spark™'s viability for applications in spatially relevant data. Two implementations were written of our simulator. One written using Spark™ and the other written using C++/Cython with OpenMP for comparison to more typical approaches to simulators. Comparing performance of Spark™ to the C++ implementation, we find Spark™ is slightly slower, though never by more than an order of magnitude. Lastly, we discuss the advantages and disadvantages of Spark™ for numerical simulation, considering performance, limitations, and ease of use of the framework.*

Keywords: Numerical Simulation, Cluster Computing, High Performance Computing

1. Introduction

1.1 Gravitational Lensing

Gravitational lensing is an astrophysical phenomenon where strong gravitational fields cause light to bend, creating displaced images of distant objects in the universe [1]. To simulate such images we use a reverse ray-tracing algorithm that calculates the deflections of light between the observer and the two-dimensional plane orthogonal to the line of sight that intersects the source object. The source may then be visualized by determining which rays successfully connect the source object to the observer. More importantly, though, we may approximate the magnification coefficient of the lens by counting how many rays successfully connect the observer to source. The magnification coefficent of the lens is something we can observe for physical systems, allowing for comparison of our model to what we see in nature. Such data is typically visualized with a magnification map, a heatmap that displays magnification as a function of source position [2].

Especially of interest is modeling quasars; immensely luminous actively accreting supermassive black holes found at the centers of many galaxies [3]. Because of their small angular size, gravitationally lensed quasars are extraordinarily sensitive to the gravitational field of the lens their light passes through. Thus we cannot model the mass profile of a galaxy as a simple distribution. To adequately model gravitatationally lensed quasars, we must consider minute deflections of light caused by the gravitational field of individual stars within the lensing galaxy around the area where the path of light intersects the lensing galaxy [2].

Performing this calculation is computationally expensive, as it involves summming up the deflection from approximately 10^4 stars per ray, for on the order of 10^9 rays. In summary, the ray-tracing calculation requires approximately 10^{14} floating point operations, consuming on the order of 60 GB of memory to generate magnification maps of high enough quality to be useful to astronomers [2]. Thus, a cluster computing approach is appropriate to perform the calculation.

With the ray-tracing complete, we must then determine which paths succesfully connect the source to the observer. For this calculation a spatial data structure is suitable, such as a kD-Tree or spatial grid. This data structure will be queried many times while producing the magnification map (on the order of 10^6 times. This is the bulk of the calculation). Hence, an efficient querying algorithm is paramount to our simulation.

1.2 Apache Spark™

As specified on the homepage of their documentation, "Apache Spark is a fast and general-purpose cluster computing system," implemented in Scala, with high-level APIs in Java, Scala, R, and Python [4]. Typically, Spark™ is used for applications in big-data analytics, such as SQL or machine learning algorithms, and as such is optimized for these applications [5]. However, at the core of Spark™'s framework is the `RDD`, or Resilient Distributed Dataset. The `RDD` is a data structure that is lazily evaluated, fault-tolerant, and distributed across a cluster. It provides an interface similar to other Scala collections, with methods such as `aggregate`, `fold`, `foreach`, `map`, and `reduce`, amongst others. Especially of note, *the* `RDD` *has methods for specifying how the data should be partitioned across the cluster* [5]. With customizable partitioning schemes, the `RDD` provides an abstraction suitable for our spatial querying

ISBN: 1-60132-487-1, CSREA Press ©

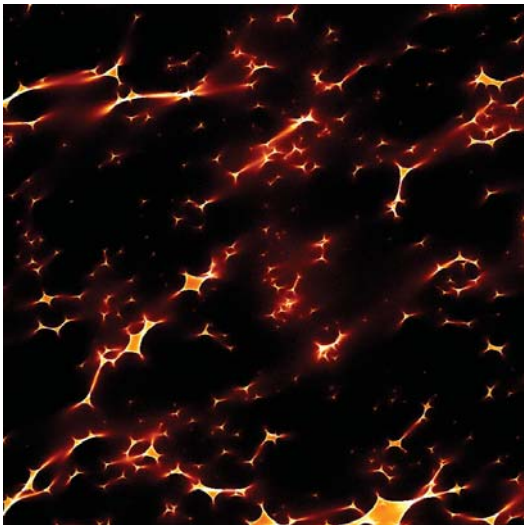

Fig. 1: A sample magnification map. The brighter the pixel, the higher magnification we would see if a source object were placed at that location.

algorithm, as it allows for efficient construction of a kD-Tree across a cluster with minimal network traffic.

2. Implementation

In recent years, Python has seen a huge growth in usage within the astrophysics community. As such, most of our simulator is written in Python, with the computationally expensive parts written in Scala using Spark™. By compiling the Scala code written to a JAR file and adding to the JVM via a command-line argument to Spark™'s launcher script `spark-submit`, we may call functions we wrote in Scala from Python through the `pyspark` interface. We did not use the `pyspark` interface for any other purpose than to transfer small amounts of data between Python and Scala, and for calling functions written in Scala.

As a comparison point, we compare performance of the implementation using Spark™ to an implementation written in C++ and Cython using OpenMP for parallelization [6]. Note that this implementation is limited to one machine only. Hence, we only draw comparisons between the C++/Cython code with simulations where Spark™ was configured to run locally only.

The calculation itself consists of three parts, 1) ray-tracing, 2) constructing the spatial data structure, and 3) querying the spatial data structure.

2.1 Ray-Tracing

In the first stage, Python hands off the parameters of the system (things like distance to the galaxy and source object, mass of the galaxy, etc.) to Spark™. Note that transferring data between Python and Spark™is exceedingly slow. Hence, when large amounts of data need to be transferred (more than 1 MB), it was found to be faster to have Python write the data to a temporary file, and then load it into Scala by reading the file. Spark™ then constructs an `RDD[DoublePair]`, where each `DoublePair` is a tuple of two doubles, describing the position of the corresponding ray in the plane of the observer. A simple call to `map` on the RDD was sufficient to perform the ray-tracing, mapping each coordinate on the observer plane to the corresponding position on the plane of the source object. This data was stored in another `RDD[DoublePair]`.

In addition to the ray-tracer just described, another was written using Spark™'s `DataFrame` abstraction as the principal distributed dataset. By using a `DataFrame`, Spark™ may perform optimizations in memory, as well as performance via its SQL engine [7]. However, Spark™ does not support custom partitioning for the `DataFrame`. Thus, to continue with the algorithm to steps 2 and 3, the data had to be transferred to `RDD` datasets after ray-tracing to continue with the computation.

2.2 Constructing the Spatial Structure

To construct the sptial data structure, we designed an interface called the `RDDGrid` which wraps an `RDD[DoublePair]`. The `RDDGrid` provides an interface for organizing and querying spatially oriented datasets. To construct an `RDDGrid`, we pass in an `RDD[DoublePair]`. The `RDDGrid` then re-partitions the data, such that each node of the cluster receives datapoints that are all near each other spatially. Two paritioning schemes were explored - one that sorts the data into equally spaced columns (the `ColumnPartitioner`), and another that creates columns of varying spacing, such that each partition receives an approximately equivalent fraction of the total dataset (`BalancedColumnPartitioner`).

Within each node, the `RDDGrid` constructs a further layer of spatial ordering, by mapping each partition to a spatial data structure, such as a kD-tree, quadtree, or spatial grid. For our purposes, a spatial grid was suitable.

2.3 Querying the Spatial Structure

To generate our magnification maps, the only method we needed for the `RDDGrid` was one to determine the number of data points that fall within a circle of arbitrary radius r centered at an arbitrary position p specified by a `DoublePair`. When a position to query is specified, the `RDDGrid` passes p to the partitioner to determine which partition(s) is pertinent to the query. For the majority of such requests, only one partition contains data relevant to the query point, and thus the spatial data structure inside that partition queries the point and returns how many data points it found. However, some query points may involve more than one partition, if the circle being searched falls on the boundary of two partitioners, or the diameter of the circle

is wider than a single partition (fig. 2). In such cases, each relevant partition queries the circle, and when each partition returns its result to the master, the master sums the count from each partition to return the total count for that query.

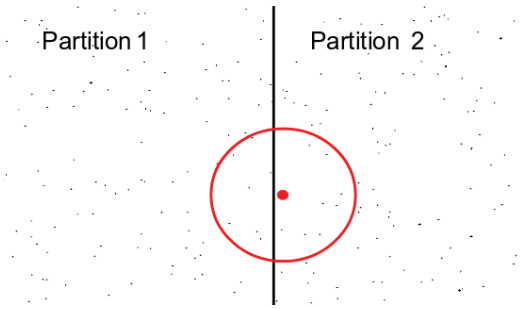

Fig. 2: Possible query point that would involve more than one partition of data. Data points are represented by black dots in the space, where the data left of the line resides in one partition, and the data to the right of the line resides in another. In this instance both partition 1 and partition 2 would have their internal spatial data structure queried, and the result of these two queries added together when each partition returns their result to the master.

3. Performance Analysis

3.1 Cluster Performance

For our simulations, we used Apache Spark™ 2.2.0, with Scala 2.11.8. The cluster used for our testing consisted of nine machines; one driver and eight executors. Each machine was equipped with an Intel(R) Xeon(R) E5-2683 processor, at 2.10 GHz and 32 cores, 32 GiB of memory, running Scientific Linux release 7.3 (Nitrogen). In total, 256 cores and 160 GB were utilized by the executors. The parameters for the gravitationally lensed system were based on those used by Pooley et al. for the system QS02237+03035, Image C [8]. Spark™ was set to use three times as many cores present for the number of partitions of the `RDD`. Thus, for our 256 cores, our dataset was broken into 768 partitions. To test performance of the system, four crucial parameters were varied - the number of individual stars to model, the number of rays to trace, the angular size of the quasar, and the resolution of the magnification map to be produced. The angular size of the quasar dictates the radius of the circle to query in step 3 of the calculation, while the resolution for the magnification map specifies the number of points to query (one query per pixel). A breakdown of the various parameter settings may be found in Table 1. In total, 54 simulations were performed to profile the parameter space, from all possible combinations of the specified settings of each of the four parameters.

Table 1: Parameters varied to measure performance, where R_g is a measure of physical size, comparable to the size of the black hole at the center of the simulated Quasar.

Setting	Star Count	Ray Count	Map Resolution	Source Size
Small	≈ 5250	1×10^8	1×10^6	$6R_g$
Medium	≈ 10500	4×10^8	2.25×10^6	$12R_g$
Large	≈ 21000	-	4×10^6	$18R_g$

3.2 Local Performance

In addition to running simulations on the cluster, the simulations were also executed locally on one machine to allow for comparison to the implementation written in C++ and Cython using OpenMP for parallelization. The C++/Cython implementation used a similar computational approach to the Spark™ implementation, though no analog to the `RDDGrid` was present. Instead, the entire dataset was stored in one large spatial grid. Furthermore, due to hardware limitations on the amount of RAM at our disposal in one machine, we were not able to perform the same scale of simulations as those on the cluster. Thus, for simulations executed locally, we cut the ray count to one fourth from those reported in Table 1 to the amounts 2.5×10^7 (S), and 1×10^8 (M). For local comparisons, we used one machine from the cluster alone.

4. Results

Table 2: Peformance times executing locally, using Spark™. Numbers reported are calculation times, measured in seconds.

Setting	Low Star (s)	Medium Star (s)	High Star (s)
MLL	294	432	570
MLM	183	327	446
MLS	122	250	372
MML	194	342	467
MMM	133	294	389
MMS	96	250	362
MSL	135	262	385
MSM	111	234	369
MSS	84	216	334
SLL	78	120	151
SLM	58	86	118
SLS	37	67	97
SML	209	250	378
SMM	128	188	321
SMS	101	168	295
SSL	106	168	299
SSM	100	157	294
SSS	93	150	290

We now report the performance times of the various simulation platforms with different parameters. For all the tables that follow, the entries in the "Setting" column refers to the parameters of ray count, map resolution, and source size, as specified in Table 1. For example, a simulation with

setting column "SML" refers to a simulation where the ray count is the small setting, the map resolution is the medium setting, and the source size is the large setting.

4.1 Local Simulations

A comparison of performance times may be found in Tables 2 and 3. We report the star count in different columns and group the other variables together because the star count had the most significant variability in a non-intuitive way. The star count has two impacts on performance. The immediately apparent impact is with more stars, each ray must calculate more deflections, such that the time of calculation for each ray goes as O(N), where N is the number of stars. Additionally, though, higher star counts leads to less uniform density of the rays upon the source plane (see Fig. 3). This results in a performance penalty during the querying phase of the computation that impact the C++ implementation more than the Spark™ implementation. Because the Spark™ has uniquely constructed spatial grids for each partition, each partition may have differently scaled grids depending on the density of data in that partition. Hence, the `RDDGrid` does a better job of fitting the data than the spatial grid implemented in C++.

Generally speaking, we see remarkably comparable performance between the C++ and Spark™ implementations. In total, the average ratio of time spent calculating with Spark™ to C++ was found as 1.5, with a standard deviation of 0.8. Delving into the individual columns, for the small, medium and large star counts, the ratios are 1.7 ± 0.8, 1.4 ± 0.8, and 1.4 ± 0.8, respectively.

Table 3: Peformance times executing locally, using C++/Cython with OpenMP. Numbers reported are calculation times, measured in seconds.

Setting	Low Star (s)	Medium Star (s)	High Star (s)
MLL	129	247	522
MLM	142	240	596
MLS	137	227	529
MML	120	251	458
MMM	126	236	492
MMS	161	254	447
MSL	121	220	505
MSM	388	288	439
MSS	127	217	445
SLL	37	57	118
SLM	61	61	139
SLS	42	65	119
SML	91	58	115
SMM	33	61	125
SMS	49	60	148
SSL	31	56	113
SSM	32	72	118
SSS	37	57	111

4.2 `RDD` implementation

The performance measurements for our `RDD` implemented cluster computations may be found in Table 4. Unfortunately we have no comparison point for performance. However, we can see that the patterns of run times increasing up and right through the table is consistent with the local runs. The first row, however does have abnormally high computational times. It is unclear if that is due to the implementation, a momentary network issue causing slowdowns, or some other random issue.

Table 4: Peformance times executing with Spark™ utilizing all eight executors. Numbers reported are calculation times, measured in seconds.

Setting	Low Star (s)	Medium Star (s)	High Star (s)
MLL	1284	1275	1325
MLM	530	555	647
MLS	182	263	311
MML	806	951	1130
MMM	465	576	866
MMS	300	413	698
MSL	488	619	895
MSM	352	468	735
MSS	260	378	664
SLL	311	337	381
SLM	174	198	268
SLS	88	120	195
SML	213	225	277
SMM	119	153	217
SMS	78	106	182
SSL	122	146	208
SSM	88	120	194
SSS	68	113	173

4.3 `DataFrame` implementation

The performance measurements for the `DataFrame` implementation may be found in Table 5. This implementation performed very poorly, and consistently encountered errors causing the program to crash. For the `DataFrame` to calculate intermediate values, it cannot do that in a transient way - instead it must allocate memory for adding columns to the frame. As such, the memory footprint of this implementation was significantly larger than the implementation using `RDD`s. Hence, we often encountered issues where the JVM could not supply enough memory to support the simulation. In addition to the memory issues, Additionally, we often saw warnings from Spark™ that we were exceeding the number of operations it could tabulate to evaluate lazily.

Of the computations that did finish successfully (we attempted all computations but the majority crashed because of the before mentioned errors), comparing the time frames to those present in the `RDD` implementation it is clear that speed of the `DataFrame` suffers for numerical simulation.

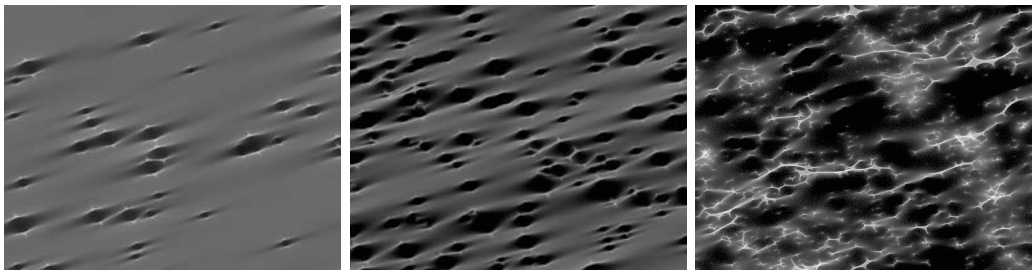

Fig. 3: Magnification map densities with increasing star counts. How uniformly shaded the magnification map is correlates to the uniformity of the density of data present in our simulation. Coloring is logarithmically scaled.

Table 5: Performance times using the `DataFrame` abstraction specified as part of `SparkSQL`

Setting	Low Star (s)	Medium Star (s)	High Star (s)
LSS	1440	-	-
MMS	292	-	-
MSS	227	-	-
SMS	80	-	-
SSS	69	-	-

5. Discussion

The decision to use Spark™ to enable cluster-computing for our simulation was one motivated out of a desire for access to more computational power without significant investment of time and energy in a more standard system using C++ and MPI. While C++ and MPI would undoubtedly have better performance over the high-level Spark™ framework and the JVM, writing the code for Spark™ was a fairly straightforward process. No implementation has been written of our algorithm using MPI, so we cannot make a direct comparison between the difficulties associated with each approach to cluster computing. We can, however, compare the difficulty of implementing the algorithms in C++ and Cython for local execution and assume that an MPI implementation would be similar to that, though significantly more complicated. Comparing the two implementations we wrote, the Spark™ was of comparable difficulty to write to the C++/Cython implementation designed for running locally only. Most of the difficulties we encountered had more to do with writing optimized Scala code, rather than difficulties with using Spark™ efficiently. Assuming the coder has experience writing optimized Scala code to run on the JVM, modifying a codebase to take advantage of Spark™ and the computational power it unlocks should be relatively simple. That being said, there were some lessons learned from using Spark™ for numerical simulation. Among them were:

- Optimal performance was found to occur when there were three times more partitions then cores available to the cluster. For our cluster of 256 CPU cores, this meant we found best performance when setting Spark™ to divide distributed data across 768 partitions.
- The largest sticking point we experienced was with additional overhead associated with creating objects in the JVM, compared to primative types. This was further exasecbated by type erasure associated with how Java compiles polymorphic code. For simulations severly constrained by available memory, this may be a serious barrier preventing viability of Spark™. While developing this code, we often ran into issues with memory because we were unintentionally doubling or quadrupling the amount of memory necessary just from object overhead. However, Spark™ does provide a remedy by providing functionality to specify how data should be stored, including allowing it to spill over to disk when available memory is low.
- It was mentioned prior that we discovered when transferring large amounts of data between Python and the JVM, it was faster to write the data to a file in Python, then have Scala read the file to receive the data rather than pipe the data from Python to the JVM directly. While not the cleanest of approaches, this approach was simple enough to implement and suitable for our needs. We use this technique for transferring data both to the JVM from Python, as well as from Python to the JVM.
- Spark™ has functionality for sharing data across all nodes of a cluster via *Broadcast variables*. However, when trying to broadcast the query points to the `RDDGrid` (An array of approximately 10^6 `Doubles`), we experienced significant slowdown. Serializing and distributing the data could take up to minutes to complete. Hence, we took a functional approach, writing objects with methods to generate the data. These objects were then distributed as broadcast variables, and the pertinent data to each node generated on the node. Naturally, this method is not suitable for all applications, though for easily calculated data, such as ours (evenly spaced locations to query), a functional approach sufficed. Note that this slowdown may have been an issue with our cluster, and not a shortcoming of broadcast variables in general. Furthermore, it was not a consistent issue - slowdowns were only experienced on some simulations, but it occurred often enough that

we opted to take a functional approach.

- When calling `aggregate` or `reduce` on an RDD, the methods `treeAggregate` and `treeReduce` are almost always preferred, as they involve putting far less data on the master at one time. When calling `aggregate`, each partition sends its result back to the master, and the master then aggregates the results together for the final result. In a prior implementation, we were using `aggregate` to return all the results of a query of approximately 10^6 points at once and stitch them together to form a magnification map. With the way that `aggregate` is written, though, this meant the master needed space in memory to fit an array of 10^6 `Long`s sent from each individual partition, which often caused our simulation to crash. The tree alternatives of these methods, when used correctly, allow for data to be stitched together on each executor before sending off to the master. This significantly reduced the memory requirements for the master, as the master now only receives 8 arrays, rather than 768. In the end, though, for our algorithm we found a simple `map` and `collect` called afterward to be the most efficient.
- An earlier implementation utilized `pyspark` more heavily for the calculation itself, rather than leaving the calculating to Scala. Even when running modules written in pure C and imported into Python, `pyspark` was prohibitively slow, and required significant memory overhead due to the necessity to continually pickle and unpickle data stored in `RDD`s. Thus, an early idea of using `pyspark` as a viable framework for distributing calculations performed in C over a cluster with C-like speed was quickly abandoned.
- Crucial to good performance of any clustered system is good balancing of the workload across machines. Using Spark™, this is especially important as Spark™ currently does not support dynamic load balancing. Thus it is extremely important that the programmer have a robust partitioner to ensure good load balancing. Note that the package Spark Streaming™ may provide a way to dynamically balance load after more development of the package [9]. While developing this implementation, this was a significant problem for us that was never resolved in a truly satisfactory way.
- Beyond discussion of success of implementation, one significant advantage of Spark™ is its resilience. In the case of a node disconnecting from the cluster, Spark™ tabulates what data is on each node, allowing it to re-calculate lost progress and resume the computation without crashing. For our cluster of eight executors this advantage was never seen - we never had nodes disconnect. However, for simulations warranting thousands of executors this could be a significant advantage of Spark™.

Lastly, while assessing the viability of Spark™ for numerical simulation, it is important to note that our simulation is somewhat unique in that it is entirely immutable. Once the rays are traced, data does not need to be mutated anywhere. Thus we cannot adequately assess Spark™'s viability for a simulation such as an N-body simulator, where the data is continually mutated. This may lead to serious performance issues, as the `RDD` is a completely immutable data structure.

6. Conclusion

In this paper we explore the viability of Apache Spark™ as a framework for cluster-based numerical simulation. Based on the performances measured, Spark™ is not as powerful as a lower-level language such as C++ for cluster-based numerical simuation. That being said, the implementation of our simulator with Spark™ was quite fast from the perspective of the coder. Most of that time was dedicated to learning how to write efficient code for the JVM, rather than facing Spark™-specific difficulties. Especially with Spark™'s `pyspark` interface, as Python grows in popularity for scientific computing, it is our opinion that Spark™ is a quite attractive option for fast development of code to solve problems warranting use of a cluster. Furthermore, the slowdowns experienced were never more than an order of magnitude, which for many applications may not be prohibitively slower.

The last consideration worth mentioning of Spark™ for numerical simulation is that it is widely supported by cloud-computing services, such as Amazon Web Services™. While it cannot compete with C++ in terms of raw power, with cloud-computing resources it does not necessarily need to.

References

[1] R. Narayan and M. Bartelmann, "Lectures on Gravitational Lensing," 1996. [Online]. Available: http://arxiv.org/abs/astro-ph/9606001

[2] B. Paczynski, "Gravitational microlensing at large optical depth," *The Astrophysical Journal*, vol. 301, p. 503, 1986. [Online]. Available: http://adsabs.harvard.edu/abs/1986ApJ...301..503P

[3] J. Wambsganss, "Gravitational lensing: numerical simulations with a hierarchical tree code," *Journal of Computational and Applied Mathematics*, vol. 109, no. 1-2, pp. 353–372, sep 1999. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0377042799001648?via%3Dihub

[4] "The Apache Spark™ website." [Online]. Available: https://spark.apache.org

[5] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark : Cluster Computing with Working Sets."

[6] D. Sverre, "Fast numerical computations with Cython," no. SciPy, pp. 15–22, 2009.

[7] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, M. Zaharia, and U. C. Berkeley, "Spark SQL : Relational Data Processing in Spark."

[8] D. Pooley, S. Rappaport, J. A. Blackburne, P. L. Schechter, and J. Wambsganss, "X-ray and optical flux ratio anomalies in quadruply lensed quasars. II. Mapping the dark matter content in elliptical galaxies," *Astrophysical Journal*, vol. 744, no. 2, 2012.

[9] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized Streams: A Fault-Tolerant Model for Scalable Stream Processing," 2012. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-259.html