Incremental and Parallel Analytics on Astrophysical Data Streams

Dmitry Mishin, Tamas Budavári, Alexander Szalay
Departments of Physics and Astronomy
Johns Hopkins University
Baltimore, Maryland 21218
Email: {dmitry,budavari,szalay}@jhu.edu

Yanif Ahmad
Department of Computer Science
Johns Hopkins University
Baltimore, Maryland 21218
Email: yanif@jhu.edu

***Abstract*—Stream processing methods and online algorithms are increasingly appealing in the scientific and large-scale data management communities due to increasing ingestion rates of scientific instruments, the ability to produce and inspect results interactively, and the simplicity and efficiency of sequential storage access over enormous datasets. This article will showcase our experiences in using off-the-shelf streaming technology to implement incremental and parallel spectral analysis on large telescopic image datasets from the Sloan Digital Sky Survey, to detect a wide variety of galaxy features. The technical focus of the article is on a robust, highly scalable principal components analysis (PCA) algorithm and its use of coordination primitives to realize consistency as part of parallel execution. Our algorithm and framework can be readily used in other domains, and we will present its ability to monitor the health of our computing infrastructure used for analysis.**

I. Introduction

Processing data in an online fashion is an integral part of scientific workflows. It facilitates data-driven control and steering, and enables interactive exploration with resource-hungry simulations and analyses amongst other functionality. In recent years, there has been an abundance of research and tools on scalable data analytics from the data management, scientific, and machine learning communities, where our need to parallelize analysis algorithms is driven by rapidly improving instruments and data collection capacity. For example, the Large Synoptic Survey Telescope is expected to generate data at a sustained rate of 160MB per second, nearly a 40-fold increase over the 4.3MB per second generation rate for the Sloan Digital Sky Survey (SDSS). However, much of this work has focused on parallelization in an offline setting, where data movement, synchronization and result delivery can be predetermined, and where scant attention has been paid to in-flight use of partial results.

Many popular analytics techniques including expectation maximization, support vector machines and principal component analysis are extremely well suited to scalable execution on dataflow architectures [1] . Batch parallel processing frameworks such as MapReduce, DryadLINQ and Spark [2] have been successfully used for these algorithms given their heavy use of *partial sums*, and the ease of programming partial sums in the aforementioned tools. Partial sums are central to online aggregation and approximate query processing, and form the basis of incremental and parallel methods that are capable of providing a feed of in-flight results. These early results are invaluable when processing petabytes, with large-scale data movement resulting in execution times ranging from minutes to weeks.

Partial sum computations are straightforward to parallelize in the offline setting, but require coordination and synchronization when used to compute a single result in high fan-in aggregation trees and highly pipelined dataflow architectures. When used on parallel data streams, results are produced at either predefined aggregation points, for example at fixed size time- or tuple-based window boundaries, or via data-driven synchronization where results are produced based on the semantics of the analysis algorithm, for example when local thresholds indicate a significant change in the result as seen in adaptive filtering and related techniques.

This article presents a scalable implementation of a partial sum analytic with data-driven synchronization on a parallel stream processing platform. The technical focus is a statistically robust algorithm that handles the presence of outliers in the streaming dataset to reduce the need for synchronization. Stream processing systems provide a pipelined execution platform for online aggregation, and enable the decoupling of storage and computation systems to fully take advantage of high-throughput sequential I/O and derandomize data access, particularly in the multiquery setting [3]. Our system of choice is IBM Infosphere Streams.

The algorithm we consider is principal components analysis (PCA), a popular algorithm [4] [5] for automatic dataset exploration that finds patterns based on the estimation of an eigensystem of a covariance matrix. PCA can be evaluated in parallel by randomly partitioning a data stream and maintaining independent eigensystem estimates for each substream. Synchronization involves communicating eigensystems to yield a global accurate estimate given disjoint substreams that potentially contain outliers.

The examples of data streams that can be monitored using this approach and that will be demonstrated include astronomic data such as observed galaxies spectra and cluster health monitoring. Using streaming PCA for analysing the spectra of galaxies allows scalable processing of large volumes of data on a cluster in realtime without running a long-period jobs to

calculate large matrices, and allowing the flexible feeding of interesting objects to the data with immediate retrieving the result of analysis. Modern cluster installations include thousands of servers, each having multiple parameters monitored, including the computation components temperature, hard drive parameters, cooling fans RPMs and so on. Wireless sensors deployed in the server room provide additional information that also should be taken into consideration. Monitoring such a system is a complex task and our streaming PCA algorithm can indicate latent features and correlations in cluster health, where a significant eigensystem deviation could indicate a hardware failure. Prediction and timely reaction to the changes can significantly improve the cluster reliability and decrease the system downtime.

II. Robust Incremental PCA

We present a novel streaming PCA algorithm, which efficiently estimates the eigensystem. First we discuss the iterative procedure, and next the statistical extensions that enable our method applicable to cope with outliers as well as noisy and incomplete data.

We focus on a fast data stream, whose elements are high-dimensional $\boldsymbol{x}_n$ vectors. Similarly to the recursion formulas commonly used for evaluating sums or averages, e.g., the mean $\boldsymbol{\mu}$ of the elements, we can also write the covariance matrix $\mathbf{C}$ as a linear combination of the previous estimate $\mathbf{C}_{\rm prev}$ and a simple expression of the incoming vector. If we solve for the top p significant eigenvectors that account for most of the sample variance, the $\mathbf{E}_p\mathbf{\Lambda}_p\mathbf{E}_p^T$ product is a good approximation of full covariance matrix, where $\{\mathbf{\Lambda}_p, \mathbf{E}_p\}$ is the truncated eigensystem. The iterative equation for the covariance becomes

$$\mathbf{C} \approx \gamma\mathbf{E}_p\mathbf{\Lambda}_p\mathbf{E}_p^T + (1-\gamma)\boldsymbol{y}\boldsymbol{y}^T = \mathbf{A}\mathbf{A}^T \tag{1}$$

where $\boldsymbol{y}=\boldsymbol{x}-\boldsymbol{\mu}$, and we can also write this as the product of a matrix $\mathbf{A}$ and its transpose, where $\mathbf{A}$ has only $(p+1)$ columns, thus it is potentially much lower rank than the covariance matrix itself [6]. The columns of $\mathbf{A}$ are constructed from the previous eigenvalues λ_k, eigenvectors $\boldsymbol{e}_k$, and the next data vector $\boldsymbol{y}$ as

$$\boldsymbol{a}_k = \boldsymbol{e}_k\sqrt{\gamma\lambda_k}, \qquad \text{for} \quad k=1\ldots p \tag{2}$$

$$\boldsymbol{a}_{p+1} = \boldsymbol{y}\sqrt{1-\gamma} \tag{3}$$

Then the singular-value decomposition of $\mathbf{A}=\mathbf{U}\mathbf{W}\mathbf{V}^T$ provides the updated eigensystem, because the eigenvectors are $\mathbf{E}=\mathbf{U}$ and the eigenvalues $\mathbf{\Lambda}=\mathbf{W}^2$

A. Robustness against Outliers

The classic PCA procedure essentially fits a hyperplane to the data, where the eigenvectors define the projection matrix onto this plane. If the truncated eigensystem consists of p basis vectors in $\mathbf{E}_p$, the projector is $\mathbf{E}_p\mathbf{E}_p^T$, and the residual of the fit for the nth data vector is

$$\boldsymbol{r}_n = (\mathbf{I} - \mathbf{E}_p\mathbf{E}_p^T)\boldsymbol{y}_n \tag{4}$$

Using this notation, PCA solves the minimization of the average $\sigma^2=\langle r_n^2\rangle$, The sensitivity of PCA to outliers comes from the fact that the sum will be dominated by the extreme values in the data set.

The current state-of-the-art technique was introduced by [7] to overcome these limitations, which is based on a robust M-estimate [8] of the scale, called M-scale. Here we minimize a different σ that is the M-scale of the residuals, and satisfies

$$\frac{1}{N}\sum_{n=1}^{N}\rho\left(\frac{r_n^2}{\sigma^2}\right) = \delta \tag{5}$$

where the robust ρ-function is bound and assumed to be scaled to values between $\rho(0)=0$ and $\rho(\infty)=1$. The parameter δ controls the breakdown point where the estimate explodes due to too much contamination of outliers. By implicit differentiation the robust solution yields a very intuitive result, where the location $\boldsymbol{\mu}$ is a weighted average of the observation vectors, and the hyperplane is derived from the eigensystem of a weighted covariance matrix,

$$\boldsymbol{\mu} = \Big(\sum w_n\boldsymbol{x}_n\Big)\Big/\Big(\sum w_n\Big) \tag{6}$$

$$\mathbf{C} = \sigma^2\Big(\sum w_n(\boldsymbol{x}_n-\boldsymbol{\mu})(\boldsymbol{x}_n-\boldsymbol{\mu})^T\Big)\Big/\Big(\sum w_n r_n^2\Big) \tag{7}$$

where $w_n = W(r_n^2/\sigma^2)$ and $W(t) = \rho'(t)$. The weight for each observation vector depends on σ^2, which suggests the appropriateness of an iterative solution, where in every step we solve for the eigenvectors and use them to calculate a new σ^2 scale; see [7] for details. One way to obtain the solution of eq.(5) is to re-write it in the intuitive form of

$$\sigma^2 = \frac{1}{N\delta}\sum_{n=1}^{N} w_n^\star r_n^2 \tag{8}$$

where $w_n^\star = W^\star(r_n^2/\sigma^2)$ with $W^\star(t)=\rho(t)/t$. Although, this is not the solution as the right hand side contains σ^2 itself, it can be shown that its iterative re-evaluation converges to the solution. By incrementally updating the eigensystem, we can allow for the solution for σ^2 simultaneously.

B. Running in Parallel

There is a strong motivation to parallelize the incremental PCA to process fast streams that a single analysis engine cannot possibly handle. The convenient solution is to drop a given percentage of the stream and process only a fraction of the data. Leaving data items out is undesirable for several reasons. One example is tracking time-dependent phenomena. Keeping all elements is vital to learn the changes in the stream in a timely manner. Dropping 90% of the data would yield new solutions in 10x longer times. Another example is filtering of data points. Often the goal is to flag outliers for further processing. Dropped items are not even considered.

We tackle very large volumes by partitioning the input stream in a simple round-robin fashion. These sub-streams are processed independently, and their eigensystems are periodically combined. For example, the common location of

two systems is the weighted average $\boldsymbol{\mu}=\gamma_1\boldsymbol{\mu}_1+\gamma_2\boldsymbol{\mu}_2$, where $\gamma_1+\gamma_2=1$. The covariance becomes

$$\mathbf{C}=\gamma_1\left(\mathbf{C}_1+\boldsymbol{\mu}_1\boldsymbol{\mu}_2^T\right)+\gamma_2\left(\mathbf{C}_2+\boldsymbol{\mu}_2\boldsymbol{\mu}_2^T\right)-\boldsymbol{\mu}\boldsymbol{\mu}^T \quad (9)$$

When the means are approximately the same, we can further simplify this using the eigensystems

$$\mathbf{C}\approx\gamma_1\left(\mathbf{E}_1\boldsymbol{\Lambda}_1\mathbf{E}_1^T\right)+\gamma_2\left(\mathbf{E}_2\boldsymbol{\Lambda}_2\mathbf{E}_2^T\right)=\mathbf{A}\mathbf{A}^T \quad (10)$$

which once again can be more efficiently decomposed with the help of a low-rank $\mathbf{A}$ matrix.

The transfer of eigensystems from separate PCA instances is coordinated to follow different syncronization strategies, e.g., peer-to-peer or broadcast. The incremental update formula includes an exponential decay term to phase out old data. This also means that after a while the separate solutions become independent. The nodes verify every time that the eigensystems are statistically independent. The result of this is that we do not need to keep track of the contributions of other streams, hence our parallel solution can scale out to arbitrary large clusters.

C. Algorithm Extensions

When dealing with the online arrival of data, there are several options to maintain the eigensystem over varying temporal extents, including a damping factor or time-based windows. These issues are orthogonal to the parallel PCA computation and can both be applied provided a single source of data that provides a well-ordered stream. Both approaches can be implemented, exploiting sharing strategies for sliding window scenarios, as well as data-driven control and coupling of the duration of the window with the eigensystem size. Our algorithm is also capable of handling missing data and gaps in the data stream.

The recursion equation for the mean is formally almost identical to the classic case, and we introduce new equations to propagate the weighted covariance matrix and the scale,

$$\boldsymbol{\mu} = \gamma_1\boldsymbol{\mu}_{\text{prev}}+(1-\gamma_1)\boldsymbol{x} \quad (11)$$

$$\mathbf{C} = \gamma_2\mathbf{C}_{\text{prev}}+(1-\gamma_2)\ \sigma^2\boldsymbol{y}\boldsymbol{y}^T/r^2 \quad (12)$$

$$\sigma^2 = \gamma_3\sigma^2_{\text{prev}}+(1-\gamma_3)\ w^\star r^2/\delta \quad (13)$$

where the γ coefficients depend on the running sums of 1, w and wr^2 denoted below by u, v and q, respectively.

$$\gamma_1 = \alpha v_{\text{prev}}/v \quad \text{with} \quad v=\alpha v_{\text{prev}}+w \quad (14)$$

$$\gamma_2 = \alpha q_{\text{prev}}/q \quad \text{with} \quad q=\alpha q_{\text{prev}}+wr^2 \quad (15)$$

$$\gamma_3 = \alpha u_{\text{prev}}/u \quad \text{with} \quad u=\alpha u_{\text{prev}}+1 \quad (16)$$

The parameter α introduced here, which takes values between 0 and 1, adjusts the rate at which the evolving solution of the eigenproblem *forgets* about past observations. It sets the characteristic width of the sliding window over the stream of data; in other words, the effective sample size.[1] The value $\alpha=1$ corresponds to the classic case of infinite memory. Since our iteration starts from a non-robust set of eigenspectra, a procedure with $\alpha<1$ is able to eliminate the effect of the initial transients. Due to the finite memory of the recursion, it is clearly disadvantagous to put the spectra on the stream in a systematic order; instead they should be randomized for best results.

[1]For example, the sequence u rapidly converges to $1/(1-\alpha)$.

It is worth noting that robust "eigenvalues" can be computed for any eigenspectra in a consistent way, which enables a meaningful comparison of the performance of various bases. To derive a robust measure of the scatter of the data along a given eigenspectrum $\boldsymbol{e}$, one can project the data on it, and formally solve the same equation as in eq.(5) but with the residuals replaced with the projected values, i.e., for the kth eigenspectrum $r_n=\boldsymbol{e}_k\boldsymbol{y}_n$. The resulting σ^2 value is a robust estimate of λ_k.

D. Missing Entries in Observations

The other common challenge is the presence of gaps in the observations, i.e., missing entries in the data vectors. Gaps emerge for numerous reasons in real-life measurements. Some cause the loss of random snippets while others correlate with physical properties of the sources. An example of the latter is the wavelength coverage of objects at different redshifts: the observed wavelength range being fixed, the detector looks at different parts of the electromagnetic spectrum for different extragalactic objects.

Now we face two separate problems. Using PCA implies that we believe the Euclidean metric to be a sensible choice for our data, i.e., it is a good measure of similarity. Often one needs to normalize the observations so that this assumption would hold. For example, if one spectrum is identical to another but the source is brighter and/or closer, their distance would be large. The normalisation guarantees that they are close in the Euclidean metric. So firstly, we must normalise every spectrum before it is entered into the streaming algorithm. This step is difficult to do in the presence of incomplete data, hence we also have to "patch" the missing data.

Inspired by [9], [10] proposed a solution, where the gaps are filled by an unbiased reconstruction using a pre-calculated eigenbasis. A final eigenbasis may be calculated iteratively by continuously filling the gaps with the previous eigenbasis until convergence is reached [11]. While [10] allowed for a bias in rotation only, the method has recently been extended to accommodate a shift in the normalisation of the data vectors [12]. Of course, the new algorithm presented in this paper can use the previous eigenbasis to fill gaps in each input data vector as they are input, thus avoiding the need for multiple iterations through the entire dataset.

The other problem is a consequence of the above solution. Having patched the incomplete data by the current best understanding of the manifold, we have artificially removed the residuals in the bins of the missing entries, thus decreased the length of the residual vector. This would result in increasingly large weights being assigned to spectra with the largest number of empty pixels. One solution is to calculate the residual vector using higher-order eigenspectra. The idea is to solve for not only the first p eigenspectra but a larger $(p+q)$ number of

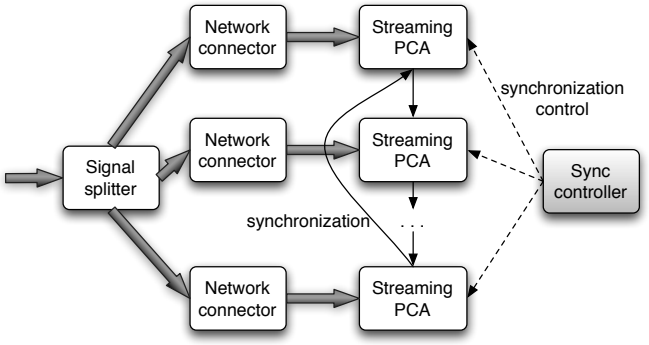

Fig. 1. Application architecture

components and estimate the residuals in the missing bins using the difference of the reconstructions on the two different truncated bases.

III. InfoSphere implementation

IBM InfoSphere Streams is a high-performance computing platform that allows user-developed applications to rapidly ingest, analyze, and correlate structured data streams as it arrives from real-time sources. The data is a stream of structured blocks – tuples, having the data structure specified by the application. The application components logic defines the data flow graph and processing routines applied to the data blocks.

A. System architecture

The application contains four logical blocks connected by three major data streams (Figure 1). Each stream consists of the specific type tuples processed by the application components. To send the tuple between the components located on one node, the tuple address can be shared, while sending between nodes involves the actual data transmission over the network.

1) Input data:

InfoSphere is flexible in choosing the source of data stream. The observations can be artificially generated by the application component for testing purposes, for example, random data matching the predefined properties. Local regular text or binary file with CSV, formated tuples, or a folder of such a files can feed the data. Side service can feed the data using piped stream file, and InfoSphere will block on the stream end until new data is streamed. Network TCP sockets and html files are also supported out of the box as a source of data. Additional data sources can be implemented using a custom operator, such as relational database. The input data for Streaming PCA is defined as a time series stream of observations – constant-length vectors of double values.

2) Load balancer:

We split the input stream by time to a number of streams that are rerouted to a corresponding PCA engine. The order of target instances is random and is chosen by the splitting component to equally balance the cluster nodes. InfoSphere provides the multi-threaded **Signal splitter** component to push the data to multiple targets without blocking the queue. Using this scheme, faster nodes will get more data than slower ones in a period of time, which should be taken into consideration in the eigensystems synchronization algorithm. The number of data streams can be adjusted to match the the rate of incoming stream tuples and the size of data vector allowing the realtime processing of the data and efficient load-balancing of available cluster nodes. The split data stream comes to a corresponding PCA engine for processing using a **Network connector**.

Although the InfoSphere SPL language provides a set of tools for maintaining the data stream and performing basic data processing, the support of efficient arrays and matrices manipulation in SPL is limited. Implementing complex algorithms dealing with matrix operations required the custom C++ **streaming PCA** operator to be implemented. This enables the freedom of choosing a matrix manipulation library having needed features, performance and scalability. Currently we use Eigen library for all matrix operations and SVD decomposition. SPL provides the developer with API to receive the incoming tuple data to the operator and send back the results of processing to be converted to SPL structures. The C++ operator is refactored into internal InfoSphere code during the project build and distributed in a cluster for running along with other components used. The stateful Streaming PCA operator stores the eigensystem matrix and other state variables as a class members and updates it as a new tuple comes to the process method. The stateful operator cannot be moved between the nodes during application execution and therefore needs the optimal placement configuration before running. Thread synchronization for PCA instances is handled by using a mutex class from InfoSphere API in the operator's process method.

Replaceable application components and flexible data flow management make it easy enough to include different partial sum analytics algorithms beyond streaming PCA into the application workflow. The implemented parallelization system will allow distributing the components on multiple nodes to process the incoming data stream in parallel. However, the synchronization criteria for parallel instances must be developed independently in each case. The synchronization schemes (token ring, broadcast, group-based) can be used or new ones can be implemented by **Sync controller**.

The application components placement on the cluster nodes significantly affects the overall system performance. To avoid the bottleneck of packet latency for the high-rate tuples exchange between the system components, we should optimize the components grouping and reduce the network exchange traffic by using the local memory data exchange between application components where possible. Limiting the network data exchange should be balanced with capabilities of scaling the computational power by increasing the number of instances running on distributed computational nodes.

B. Synchronization

While load balancing sends the tuples to the instances in random round-robin manner, some instances can have the

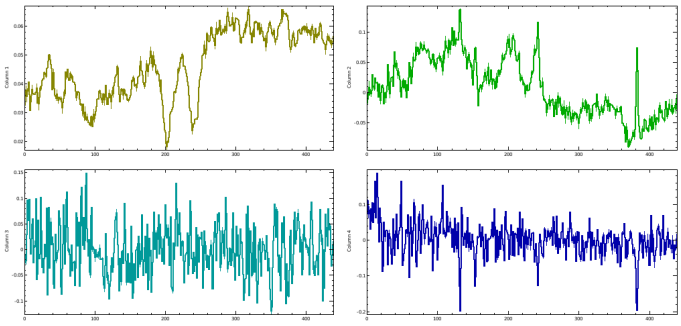

Fig. 2. Spectra analysis start

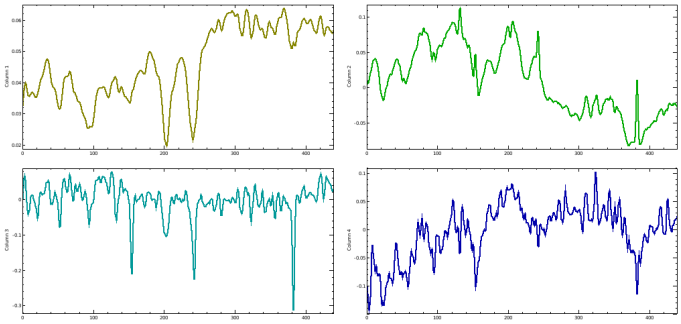

Fig. 3. Spectra analysis during the convergence process

eigensystem values different to the rest of the instances. This may be caused by improper application initialization process when an outlier is in the data, some unusual pattern of incoming data and so forth. Also we want to detect outlier values arriving at each PCA instance. The idea of the algorithm is keeping the eigensystem in sync between the nodes, so that the result eigensystem can be received from any node at any period of time. To achieve this, a mechanism of periodical eigensystems synchronization messages is implemented in the application. Different synchronization scenarios can be implemented depending on the input data and frequency of processed results reading from the application. For example, token ring synchronization involves each instance sending its state to a next one, with the n^{th} instance sending the state to the first. Synchronization signals are generated by a synchronization manager component that sends it to selected PCA engines using a control port. When a control signal is received, our PCA component shares the current eigensystem state with a set of other instances defined in the control message and continues the data processing.

REFERENCES

[1] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun, "Map-reduce for machine learning on multicore," in *NIPS*, 2006, pp. 281–288.

[2] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI*, Apr. 2012.

[3] K. Kanov, E. A. Perlman, R. C. Burns, Y. Ahmad, and A. S. Szalay, "I/o streaming evaluation of batch queries for data-intensive computational turbulence," in *SC*, 2011, p. 29.

[4] I. Jolliffe, *Principal Component Analysis*. Springer Verlag, 1986.

[5] J. Sun, D. Tao, S. Papadimitriou, P. S. Yu, and C. Faloutsos, "Incremental tensor analysis: Theory and applications," *TKDD*, vol. 2, no. 3, 2008.

[6] Y. L. Li, L.-Q. Xu, J. Morphett, and R. Jacobs, "An integrated algorithm of incremental and robust pca," in *ICIP*, 2003, pp. 245–248.

[7] R. Maronna, "Principal components and orthogonal regression based on robust scales," *Technometrics*, vol. 47, no. 3, pp. 264–273, Aug. 2005. [Online]. Available: http://dx.doi.org/10.1198/004017005000000166

[8] P. J. Huber, *Robust statistics*. Wiley, New York, 1981.

[9] R. Everson and L. Sirovich, "J. opt." pp. 12, 8, 1995.

[10] A. J. Connolly and A. S. Szalay, pp. 117, 2052, 1999.

[11] C. W. Yip and et al., "AJ," pp. 128, 585, 2004a.

[12] V. Wild, G. Kauffmann, T. Heckman, S. Charlot, G. Lemson, J. Brinchmann, T. Reichard, and A. Pasquali, "MNRAS," pp. 381, 543, 2007.