SMiPE: Estimating the Progress of Recurring Iterative Distributed Dataflows

Jannis Koch, Lauritz Thamsen, Florian Schmidt, and Odej Kao
Technische Universität Berlin, Germany
{firstname.lastname}@tu-berlin.de

***Abstract*—Distributed dataflow systems such as Apache Spark allow the execution of iterative programs at large scale on clusters. In production use, programs are often recurring and have strict latency requirements. Yet, choosing appropriate resource allocations is difficult as runtimes are dependent on hard-to-predict factors, including failures, cluster utilization and dataset characteristics. Offline runtime prediction helps to estimate resource requirements, but cannot take into account inherent variance due to, for example, changing cluster states.**

We present SMiPE, a system estimating the progress of iterative dataflows by matching a running job to previous executions based on similarity, capturing properties such as convergence, hardware utilization and runtime. SMiPE is not limited to a specific framework due to its black-box approach and is able to adapt to changing cluster states reflected in the current job's statistics. SMiPE automatically adapts its similarity matching to algorithm-specific profiles by training parameters on the job history. We evaluated SMiPE with three iterative Spark jobs and nine datasets. The results show that SMiPE is effective in choosing useful historic runs and predicts runtimes with a mean relative error of 9.1% to 13.1%.

***Index Terms*—Scalable Data Analysis, Distributed Dataflows, Runtime Prediction, Progress Estimation, Iterative Algorithms**

I. Introduction

As datasets have increased in size, the use of distributed analytics systems for applications in both industry and science has become prevalent [1], [2]. Distributed dataflow systems like MapReduce [3] and Spark [4] execute programs at large scales on computer clusters. Many of these programs are iterative, including graph and machine learning algorithms such as PageRank and k-means clustering.

Productive use of distributed analytics systems often requires adherence to *Service-Level Objectives* (SLOs), as workloads often have strict latency requirements [5], [6], [7], [8]. Many production workloads are also business-critical and missing completion deadlines can incur economic penalties [9]. Moreover, workloads are often executed periodically. For example, on Microsoft's larger clusters over 60% of jobs are recurring [5]. When submitting a distributed analytics job, a user requests a specific amount of resources with the goal of keeping the runtime and execution costs to a minimum. While increasing resources often leads to a better performance, the exact relationship is difficult to estimate. Any gain in computational power may be offset by additional communication. Moreover, the runtime of a job is not only dependent on the assigned resources, but a variety of factors. This includes, for example, interference with concurrently running workloads [5] and failures at software or hardware level [9]. Other factors are connected to the algorithm itself, like the dependency between individual tasks [10] or the impact of program parameters. Lastly, the input data influences performance to a large extent. Even similarly sized datasets can yield significantly different performance [6]. Choosing a resource allocation to achieve a specific runtime goal is, thus, not trivial and users routinely overprovision resources, yielding poor overall cluster utilizations. To address this, many previous efforts have focused on predicting the runtime for a given resource configuration in order to select a configuration that meets a given SLO. Such predictions are often based on data from profiling runs [6], [11], [12], [13], executed on a subset of the input or a small number of servers. However, extrapolating from those samples is difficult and the additional runs incur an overhead. Other systems use data from previous executions [10], but it is not clear which previous runs can be used for accurate estimations. Also, predicting runtimes before the execution cannot take into account inherent variance in runtimes due to changing cluster states. Such factors can only be considered at runtime, using statistics collected during the execution. Moreover, these statistics provide opportunities to match execution properties unavailable to offline prediction. For example, predicting the impact of changes in input datasets or job parameters would require highly complex models and detailed statistics. However, the impact of these factors is reflected in current statistics, enabling online progress estimation to match similar jobs more effectively and thus increase prediction accuracy. Accurate progress estimation at runtime then allows to dynamically adjust resources or at least notify users when their jobs are at risk of missing targeted runtimes.

In this paper, we present SMiPE, a progress estimation system which matches a running job to previous executions based on similarity. SMiPE uses a black-box approach, only relies on statistics from previous runs and is therefore generically applicable to iterative distributed dataflows. This is in contrast to previous work such as Jockey [9], which uses a framework-specific job model, and efforts using profiling runs to gather sampling information, such as Quasar [6]. SMiPE's matching component selects those executions from the job history with a high similarity and uses these to predict how the current job will behave for the remaining iterations. As the job progresses, more statistics become available and the selection of similar executions becomes more precise. In contrast to offline runtime prediction systems, SMiPE is able to incorporate hard-to-predict factors that are reflected in the statistics of the running job. Thus, for example, the impact of changes in the cluster state is included in SMiPE's estimations. SMiPE can use several similarity measures such as per-iteration runtimes, convergence behavior or hardware utilization. As jobs have distinct profiles that determine which similarity measures lead to an efficient matching result SMiPE performs job-specific parameter training. The system learns automatically which measures are most useful for a job, eliminating the need for manual parameter tuning.

Contributions. The contributions of this paper are:

- An approach to progress estimation for iterative distributed dataflows, which matches a running job to previous executions based on similarity.
- The definition and evaluation of five different measures for the similarity of job runs.
- SMiPE, a practical system implementing our approach, integrated with Spark.
- An evaluation of SMiPE using cluster experiments and three iterative algorithms.

This paper is structured as follows. Section II provides background on distributed data analytics, while Section VII presents related work. Section IV presents our approach to progress estimation for iterative distributed dataflows. Section V explains our similarity measures. Section VI presents our evaluation. Section III concludes this paper.

II. Background

This section gives background on distributed dataflow systems, cluster management and iterative distributed algorithms.

A. Distributed Dataflow Systems

Distributed dataflow frameworks allow to express programs using operators such as Map, Reduce, Filter or Join. These operators are configured by specifying the input data and a sequential user-defined function (UDF).

A program is an aggregation of such operators which are combined with various degrees of freedom, depending on the framework's programming model. Frameworks such as Apache Spark[1] allow a distributed program to be expressed as combinations of operators, i.e. the output of one operator serves as the input of another operator. The result is a pipelined dataflow. Some frameworks also have native support for iterations, which are in turn a group of operators combined with a defined termination condition.

Fig. 1 shows an example of a dataflow program consisting of a Map, Join, Group and a Reduce operator. The latter three are part of an iteration. The Join operator uses two inputs, one coming from another operator, the other from an input file.

The framework handles the distribution and parallelization of the dataflow program. Operators are executed in a data-parallel fashion: Input data is split up and multiple instances

[1]https://spark.apache.org

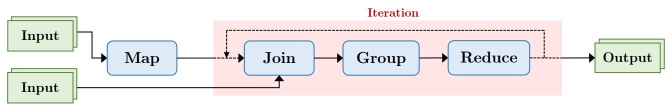


Fig. 1: Example of a dataflow program consisting of operators, including an iteration

of a single operator process the resulting partitions in parallel by calling the respective UDF. As a result, one operator is executed by several *execution tasks*, which are distributed across the computer cluster by the selected framework for parallel execution.

The input of an operator can either be the contents of a file or the output of another operator. In the first case, a file is usually read from a distributed file system (DFS). In the DFS, files are partitioned into blocks. As servers are used for both storage and computation, frameworks try to place the computation on those servers where replica of the input data are available to avoid data transfer. This principle is called *data locality*. If the input comes from another operator, the data may have to be transferred from a remote cluster node. The nature of this communication depends on the operator: For Map or Filter it is a simple one-to-one communication. A Group operator, on the other hand, requires all-to-all communication, because data records are grouped by a key and may have to be sent to any of the executing tasks [9].

Execution tasks of consecutive operators can be combined, if the operator's logic permits it. For example, it is possible to execute consecutive filter and map operations in a single step. The logic of other operators requires a barrier in the dataflow, which means that the operator cannot execute until all tasks of previous operators have completed. A Join, for example, requires the results of the two inputs to be fully available. Likewise, the beginning of an iteration is a natural barrier.

B. Cluster Management

Clusters are usually shared among and concurrently used by multiple different users and processing frameworks. Resource managing systems such as Apache Hadoop's YARN [14] or Apache Mesos [15] provide an abstraction layer for cluster resources so they can be used by various distributed frameworks. Typically, resource management systems provide shares of computational power in the form of containers. These containers can be requested by the frameworks, are then scheduled by the resource management systems, and subsequentely can be used for executing distributed programs. Containers can provide different levels of resource isolation, including no isolation at all for access to resources like CPU cores and network I/O, yet still are used for distributing workloads in clusters.

III. Related Work

We first present relevant frameworks for distributed data analysis and then related work in the context of progress estimation as well as dynamic resource adjustment.

A. Distributed Data Analysis Frameworks

MapReduce is a distributed programming and execution model [3]. Its simple programming model lacks, however, native support for iterative executions. For iterative algorithms, multiple consecutive MapReduce jobs have to be submitted which involves writing intermediate results to disk and leads to significant overhead. Systems such as Apache Spark [4] and Apache Flink [16] extend the MapReduce model with additional operators such as Filter, Group or Join and express programs as arbitrary directed acyclic graphs of such operators. They also provide native support for iterations. With GraphX [17] and Gelly, both frameworks provide graph libraries implementing the Pregel programming model [18] for iterative graph algorithms.

B. Offline Runtime Prediction

Some work has addressed offline runtime prediction, mostly to allocate resources for a specific runtime target.

AROMA [11] predicts the runtime of MapReduce jobs and is similar to our approach in that it also matches a job to previous executions. However, instead of considering multiple different factors, including factors particular to iterative algorithms, the matching is based on the jobs' resource utilization. Historic executions are put into groups according to similar CPU, network and disk utilization patterns. For each group, a prediction model is trained. A newly submitted job's hardware utilization is classified according to a sample run on a fraction of the input. According to this classification, the precomputed prediction model is used to estimate the job's runtime.

BELL [10] uses historical data from previous executions to model the scale-out behavior of jobs. The system uses both parametric and non-parametric regression on runtime data of the same job executed on similar-sized data. Then, given a job and the number of nodes, an estimate of the total runtime is available and BELL can select a resource configuration to meet a user-specified runtime target. BELL does not consider changes to program parameters or the cluster state when predicting runtimes based on previous runs.

PREDIcT [13] also predicts runtimes for iterative algorithms but focuses on graph algorithms. It uses sampling on the input data: the program is run on subgraphs and execution characteristics such as messages sent per iteration are extrapolated to the whole dataset. However, this only works if the subgraph preserves the relevant properties for the given algorithm. Also, any algorithm parameters must be manually adjusted for the sample run by the user.

Ernest [12] executes training runs on a small number of servers. It captures computation and communication patterns as a function of the number of nodes. These patterns are then used to model the runtime for a higher number of nodes. However, the system must be retrained if the code or the dataset changes.

C. Dynamic Resource Adaption

Approaches for dynamically adapting a job's resources during its execution can make use of statistics of the currently running job and take into account the current cluster state. Such approaches require some form of progress estimation at runtime, similar to our approach.

Jockey [9] is a system for SCOPE, which dynamically adapts a job's parallelism. It uses historical data and a simulator to analyze the internal job structure. Jockey precomputes job statistics by simulating a job's internal dependencies. Instead of runtime targets, a utility function is used which denotes the economic value of a job in relation to its total execution time. Jockey monitors a running job and dynamically selects a resource allocation which maximizes a job's utility and minimizes its impact on the cluster.

Delimitrou et al. stress the need for a system which relies on users expressing performance constraints instead of resource reservations. They propose Quasar [6], a system which uses classification techniques to predict how different resource configurations impact a job's performance, also considering interference of currently running workloads on the cluster. The system includes statistics from short profile runs on a few servers. Quasar is designed to meet given performance constraints and to improve the cluster utilization.

IV. Approach

This section presents our approach to progress estimation. Fig. 2 gives a conceptual overview of the algorithm, which is divided into three steps: ① Similarity Matching (IV-A), ② Estimation Inference (IV-B) and ③ Final Estimate (IV-C). They are explained in the following sections.

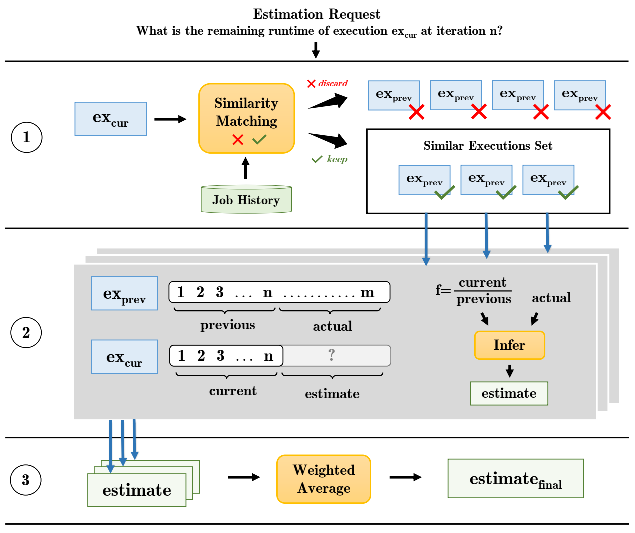


Fig. 2: Overview of the estimation approach.

A. Similarity Matching

Similarity Matching selects executions from the job history, which are similar to the current execution. The job history contains a range of statistical data for each execution, including job information (e.g. executed UDF, input data), execution information (e.g. the number of iterations, per-iteration runtimes) and hardware utilization information (e.g. periodic values for CPU, network-io and disk-io). We define several individual similarity measures for the matching, each of which is defined by a function $sim\left(ex_{cur}, ex_{prev}, i_{cur}\right) \in [0,1]$. The function takes the current execution ex_{cur}, the previous execution ex_{prev} and the current iteration i_{cur} and returns a value between zero and one. Zero represents the lowest and one the highest possible similarity.

The goal is to combine those individual measures to a final similarity value sim_{final}, which is used by the estimator to compare two executions. Each similarity has a threshold T and a weight w. The matching system discards executions, when any of the given similarity values are smaller than the threshold T. For the executions not discarded, a weighted average sim_{final} is calculated from the similarity values using the respective weights w. By testing whether sim_{final} is greater than another user-defined threshold T_{final}, we verify a high similarity. Such executions are added to the *Similar Executions Set* (SES). If all executions are discarded, the thresholds T of the similarity measures are lowered until at least one similar job execution is found.

B. Estimation Inference

The knowledge about the remaining runtime of the similar executions is used to estimate the remaining runtime for the current job. An estimate for the remaining runtime is inferred from each execution in the Similar Executions Set. We assume that the current execution will have the same number of executions as the similar execution. As the current job behavior might be different than historic jobs, we adjust our prediction by comparing the differences in runtimes of the previous iterations of the current execution and the similar execution, by calculating $f_i = runtime_{current}/\ runtime_{similar}$ for each iteration i. We assume that the differences in runtime up to the current iteration will propagate in the remaining iterations. These previous differences are represented by the average f_{total} from all f_i. Thus, our overall estimate for the remaining runtime for the current job is the remaining runtime estimation of the historic and similar jobs multiplied with the adjustment value f_{total}.

To account for the differences in resource configurations, the runtimes of the similar execution have to be adjusted to be comparable to the runtimes of the current execution. For this, we use a scale-out model, which reflects how runtimes change with the number of cluster nodes,. We use this model by calculating the factor by which the two executions differ and multiply the runtime of the similar execution by this factor. We perform the adjustment iteration-wise and thus support jobs, which have changed resource allocations during the execution. The adjustment of the similar execution is not only necessary for the iterations up to the current iteration, but also the remaining iterations. Lastly, we weigh iterations, giving more weight to more recent iterations, assuming that recent changes in the cluster state will continue for future iterations. SMiPE uses the inverse function $1/x$ for that.

C. Final Estimate

The set of estimates calculated in the previous step are now combined to a final estimate using a weighted average. As weights, the final similarity values of the respective executions are used. Thus, executions more similar to the current execution contribute more to the final estimate. The final similarity values are greater than the threshold T_{final} as per definition. Thus, the differences of weights is relatively small. In order to make the weighting more effective, the final similarity values are adjusted using a transformation function of the form sim^b ($b > 1$), increasing the difference between weights.

V. Similarity Measures and Quality

This section introduces and defines the similarity measures. It also describes the *Similarity Quality* measure and explains how the measure is used for parameter training.

A. Similarity Measures

We use the following similarity measures for SMiPE.

1) Runtime Similarity: The Runtime Similarity compares the distribution of runtimes. It is defined as the average of iteration-wise deviations of the previous runtimes up to the current iteration. The runtime values are scale-out adjusted to enable a comparison.

2) Active Data Records Similarity: The Active Data Records Similarity is defined as the average of iteration-wise deviations of the number of records and captures the convergence behavior of executions.

3) Hardware Statistics Similarity: The measure captures how executions use hardware resources. It uses statistics of the CPU, the disk and network, which are periodically monitored on each node. We take the average of those values on every node from the previous iterations. Then, the values of the sorted sequences of averages are compared pair-wise, averaging the deviations.

4) Input Similarity: The Input Similarity models the difference of the input dataset sizes. Distributed programs often execute input datasets recurrently. The input dataset size of such jobs is usually the same.

5) Scale-Out Similarity: This similarity compares the number of cluster nodes of two executions. Although SMiPE adjusts runtimes according to their scale-out, this may introduce inaccuracies making it worth to distinguish jobs according to their resource configuration.

B. Similarity Quality

The similarity measures capture properties which *potentially* have value to the estimation. This may or may not be the case for actual job history data. We describe a concept called *Similarity Quality*, which can be used to judge this usefulness.

We perform simulations in SMiPE using executions from the job history. For a simulation we select an execution j as the current execution and a value i as the current iteration. We select another execution k and request an estimate of SMiPE for j, based on k only. We know the actual remaining runtime from the job history and can calculate the estimation accuracy. Additionally, we calculate s, the value of a similarity measure of j and k. This is repeated for all jobs k and j in the history. As a result we get P, a set of points (s, a), where s is the similarity value of the simulated current execution j and the other execution k, and a is the estimation accuracy. These points represent the relationship of a specific similarity measure and the accuracy in the estimation algorithm. The points can be plotted in a points chart to visualize this relationship. Further, we define

$$h(t) := avg \{ a \mid (s,a) \in P, \quad s \geq t \},$$
$$n(t) := 1/|P| \times | \{ a \mid (s,a) \in P, \quad s \geq t \} | \text{ and}$$
$$q(x) := h(y), \quad \text{with } n(y) = 1 - x .$$

This gives the cumulative histogram $h(t)$, which is the average accuracy we can expect if the selection threshold for a similarity is set to t. The value $n(t)$ indicates the share of selected points. Lastly, $q(x)$ is the *normalized quality* which combines the histogram and the point share into one measure to enable the comparison of two similarities. This measure normalizes the histogram such that the point share becomes the same, a straight line for every Similarity Quality.

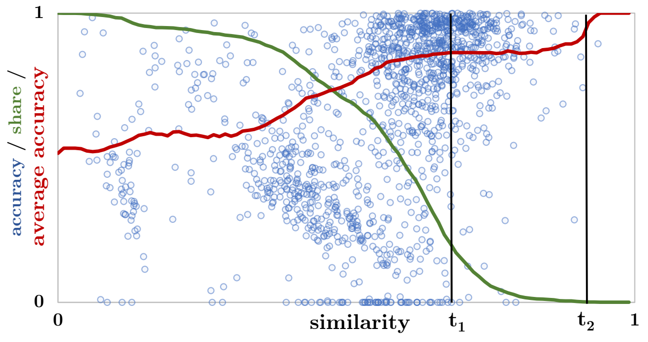


Fig. 3: Similarity Quality histogram (red), point share (green) and points (blue) of a sample measure.

We can now characterize that a similarity measure is useful to the estimation algorithm, if it is possible to find a threshold t for which the histogram $h(t)$ indicates a high expected accuracy, while share function $n(t)$ indicates that enough executions will be found in the similarity matching. Fig. 3 shows the histogram, the point share and the point chart of a sample similarity measure. If the similarity threshold is set to a high value t_2, the histogram shows that an expected average accuracy close to 1 could be reached - however the point share would be very low. A lower value t_2 still yields accurate estimations, but is expected to match considerably more executions in the algorithm.

C. Parameter Training

The similarity matching is influenced mainly by the thresholds and the weights of the similarity measures. Instead of setting these heuristic parameters manually, SMiPE trains them using the job history.

1) Similarity Thresholds: From the job history we infer the Similarity Quality charts, which are used for training the thresholds. We set a minimum accuracy a_{min} as well as a minimum point share n_{min} to avoid the scenario that the threshold is too high for the matching to find enough executions. Then, for each similarity measure the highest possible threshold t is chosen such that $h(t) \geq a_{min}$ and $n(t) \geq n_{min}$.

2) Similarity Weights: An optimal solution must be found for the vector $w = (w_1, w_2, ..., w_n)$, weighting the similarity measures. Again, we make use of simulations: For every execution of a job, we simulate a currently running job using different current iterations and estimate the remaining runtime based on all other executions of the job, matched with different weights. To find optimal weights we use numerical optimization: we use the mean relative error of estimations as objective function and minimize this using Powell's BOBYQA algorithm [19]. We perform the optimization repeatedly using different initial guess values for the weights. The result is a weights vector yielding the minimum mean relative error.

VI. Implementation

This section describes how the SMiPE implementation integrates with existing systems and gives an overview of its components.

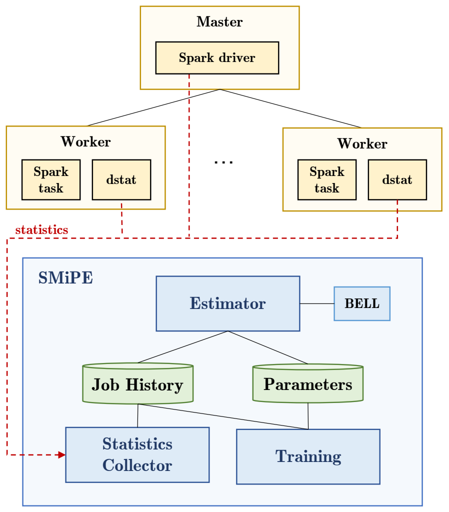


Fig. 4: Integration with a Spark cluster.

A. Integration

SMiPE provides its functionality via interfaces which are not dependent on a specific distributed dataflow framework. We use Apache Spark as an example framework to illustrate how these interfaces are used.

Fig. 4 shows how a Spark job is executed and how it interacts with SMiPE. Spark follows the *Master-Worker* architecture. On the master node, the Spark *driver program* is executed, which runs and coordinates the Spark program and schedules Spark *tasks* on the worker nodes. These perform the actual processing by calling the program's UDFs.

B. Statistics Collector

Statistical data of executing jobs is collected and persisted to the job history repository by the *Statistics Collector*. This component provides an interface which can be used by other components to submit statistics to SMiPE. In our implementation, the Spark program calls this interface at the beginning and at the end of each iteration, and provides the number of data records computed in the iteration. Spark provides APIs to retrieve such statistics for the currently executing job. For hardware statistics, we use Dstat[2], which runs on every worker node. It collects statistics of CPU, network and IO periodically and writes the values to a file. These are read and submitted to the Collector.

C. Estimator

The *Estimator* component implements our runtime estimation approach. It receives estimation requests and returns the predicted remaining runtime. The Estimator uses the statistical data from the Job History. SMiPE uses a scale-out model to adjust runtimes of executions run on a different number of cluster nodes. The BELL system [10] provides such a model by using parametric and non-parametric regression on data of previous run times. Originally, BELL uses the runtime of entire executions as the training data. We adjust BELL such that all runtimes of a particular iteration are used as training data to support iteration-wise comparison.

D. Training

This component performs the training for the similarity thresholds and weights (V-C). It uses the Apache Commons Math library[3] which provides an implementation of the BOBYQA algorithm. The training is executed periodically and stores the resulting parameters in a repository. The precomputation of parameters enables SMiPE to satisfy estimation requests more efficiently.

VII. Evaluation

This chapter evaluates SMiPE. We describe the cluster experiments, analyze the similarity measures using the Similarity Quality and finally evaluate the estimation accuracy.

A. Experiments

Table I gives an overview of all experiments and shows algorithms, datasets and parameters. Each setting was executed on up to 40 nodes, and repeated 4 (SGD), 6 (PageRank) or 9 (Connected Components) times.

Algorithm	Input Dataset	Size	Parameters
PageRank	LiveJournal	1.00 GB	d=.01, d=.001, d=.0001
	Kronecker24	1.52 GB	d=.01, d=.001
	Kronecker25	3.43 GB	d=.01, d=.001
	Wiki	5.74 GB	d=.09, d=.01
Connected Components	LiveJournal	1.00 GB	–
	Kronecker24	1.52 GB	
	Kronecker25	3.43 GB	
	Wiki	5.74 GB	
SGD	50-100M	99.4 GB	step size = 1.0 convergence delta = 0.001
	25-250M	124.1 GB	
	25-500M	247.9 GB	
	50-500M	497.0 GB	
	50-750M	745.5 GB	

TABLE I: Overview of all experiments.

[2]http://dag.wiee.rs/home-made/dstat/

[3]http://commons.apache.org/proper/commons-math/

1) Cluster: We conducted experiments on a cluster consisting of 40 servers, each equipped with a quad-core Intel Xeon CPU with 3.3 GHz and 16 GB main memory, connected through a single 1 Gigabit Ethernet switch. The servers' operating system is Linux (Kernel 3.10.0). For the experiments, we used Apache Spark 2.0.0 in conjunction with Apache Hadoop 2.7.1 and Java 1.8. For the collection of hardware statistics, we used Dstat 0.7.2. During the experiments, the servers were only used for the SMiPE evaluation, without interference with other workloads.

2) Algorithms: We used the algorithms PageRank (PR), Connected Components (CC) and Stochastic Gradient Descent (SGD) in the experiments. PageRank is a prominent eigenvector centrality algorithm, calculating the importance of every vertex within a graph. The Connected Components algorithm finds all connected components of a graph using iterative label propagation. Stochastic Gradient Descent iteratively finds the minimum or maximum of an objective function, using gradient approximation for increased efficiency. We used Spark's GraphX library[17] implementation for PR and CC and Spark's machine learning library MLlib[4] for SGD. For PR, we used different convergence deltas d.

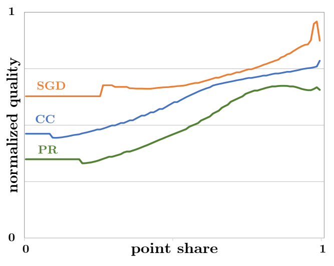
3) Datasets: We used both real world and generated datasets of different sizes. The Wikipedia dataset is a graph of encyclopedia pages of the English Wikipedia.The LiveJournal dataset is a social graph from the LiveJournal social network, representing friendship relationships. Both datasets were taken from the KONECT graph collection [20]. The Kronecker datasets were generated with the graph generator of the Big Data Generator Suite (BGDS) [21], using 24 and 25 iterations. For SGD, features datasets were generated using our own generator, explicitly creating a Vandermonde matrix to generate multi-dimensional feature vectors that fit a polynomial model of a certain degree with added Gaussian noise. The generated datasets contain 100 to 750 million points and 25 or 50 features. The dataset name *50-100M* stands for a dataset with 50 features and 100 million points.

B. Similarity Measures

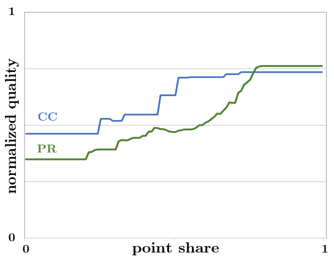
We evaluate the similarity measures by using the normalized quality chart as defined in V-B. In the following, it is simply referred to as *(similarity) quality*. The quality graphs of the measures are shown in Fig. 5.

Fig. 5a) shows that the Runtime Similarity increases the expected accuracy significantly with an increased point share. The differences in the accuracy are greater for PR and CC than for SGD. The Active Data Records Similarity (5b) is compared for PR and CC only. The similarity is effective for both datasets, but more so for PR. We analyze the hardware similarity in Fig. 5 c) to e) for the CPU, network and disk IO. For the CPU, only very high point share values lead to a higher expected accuracy. This effect is most visible for PageRank, where the line is flat until it rises sharply for a point share value close to 1. The network and disk IO charts perform

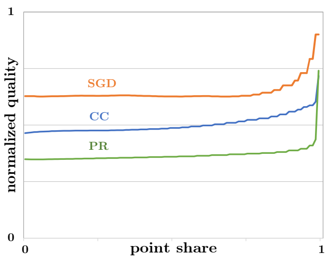
[4]http://spark.apache.org/mllib/, accessed 2017-08-17



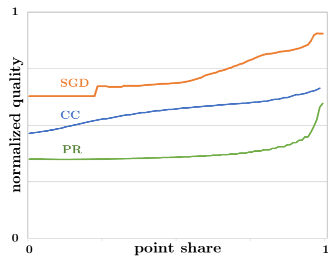
(a) Runtime Similarity



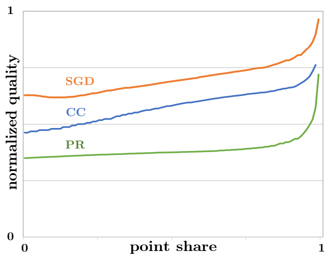
(b) Active Data Records Similarity



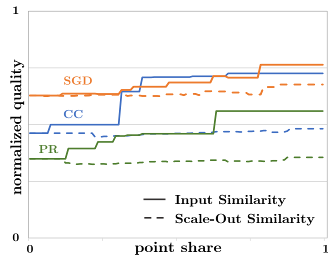
(c) CPU Similarity



(d) Network-IO Similarity



(e) Disk-IO Similarity



(f) Input and Scale-Out Similarity

Fig. 5: Quality charts of the similarity measures.

better in that they achieve a higher expected accuracy also for point share values below the top end of the scale. They are effective for all algorithms in increasing the expected accuracy. The Input Similarity (5d, solid lines) is able to distinguish executions according to their accuracy for all algorithms. This effect is more notable for CC and PR than for SGD. The Scale-Out Similarity (5d, dotted lines) only leads to a minimal increase in the average accuracy for higher point share values in all three cases.

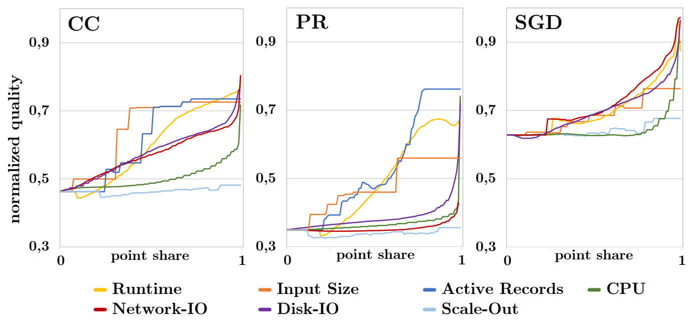


Fig. 6: Summary of all quality charts.

Fig. 6 shows the quality charts of all similarity measures for each algorithm. We observe that each algorithm has a distinct profile that determines which similarity measures are the most useful. For PR, the Active Data Records Similarity

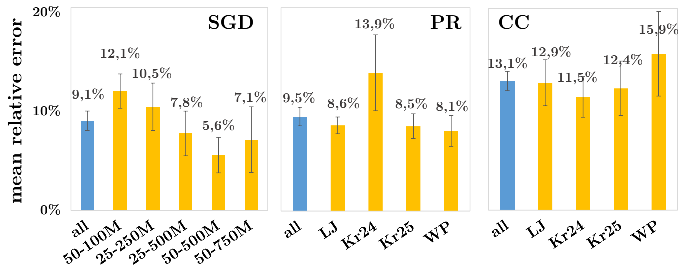


Fig. 7: Relative mean errors by algorithm and dataset.

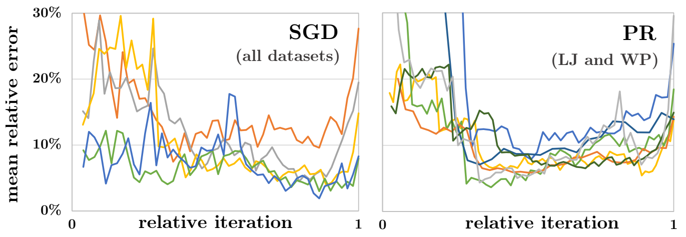


Fig. 8: Iteration-wise estimation accuracy.

is the most important measure while for SGD, the Network Similarity provides the highest contribution. This highlights the importance of the parameter training, which allows SMiPE to adapt to those algorithm-specific profiles. Additionally, judging the usefulness of similarities is not always intuitive, and it proves practical to automate the analysis using the quality measures.

C. *Estimation Accuracy*

For every execution in the job history, we select representative sample iterations and calculate the relative error of the estimate. Depending on the execution's total number of iterations, up to three sample iterations are chosen with equally spaced distance and an additional padding of 15% of the total iterations at the beginning and the end. For each selected sample iteration, we use SMiPE to estimate the remaining runtime using all runs done for the experiments as job history and compare this to the known actual remaining runtime to calculate the estimation error.

To judge the accuracy, we calculate the mean relative estimation error. Fig. 7 shows the mean relative error by datasets (yellow) and in total (blue) for all algorithms. The whiskers in the graph represent the 95% confidence interval. SGD has the lowest mean relative error with 9.1%, followed by PR with 9.5%. The mean relative error of CC is 13.1%. However, since the runtimes of CC are short, even estimates with a low absolute error yield high relative errors. The mean absolute error for CC is only 1.1 seconds. If we compare individual datasets, for SGD the individual mean relative errors range from 5.6% to 12.1%, while for CC the values vary between 11.5% and 15.9 %. For PR, the values are between 8.1% and 8.6% with the exception of the Kronecker24 dataset, which has a mean relative error of 13.9%. The reason for this high error lies in the specific convergence behavior of PR with Kronecker24. The number of Active Records with this dataset remains almost the same for the entire execution, with the values sharply decreasing towards the end. The Active Records Similarity cannot distinguish executions with different delta parameters until the very end of the execution. Thus, executions with lower parameter are considered similar, but have more iterations and a longer remaining runtime, leading to a considerable overestimation.

Fig. 8 shows the iteration-wise relative error for all iterations for SGD and PR. Each line represents executions with the same input dataset and the same algorithm parameters. The iteration on the x-axis is relative for the curves to be comparable. The accuracies are mostly between 5% and 15% in the middle of the executions. Both algorithms exhibit considerably higher errors in the beginning and in the end of the execution. In the beginning, only little data is available for a running execution, thus selecting similar jobs is bound to be inaccurate. As the execution progresses, the algorithm can better distinguish similar jobs. It takes around 30% of the total iterations until the the estimation reaches the peak accuracy. The higher errors at the end are due to the remaining runtimes becoming small, so even small absolute errors lead to higher relative error rates.

VIII. Conclusion

In this paper, we presented SMiPE, a system that accurately estimates the remaining runtime of recurring iterative distributed dataflows. SMiPE's core is a matching algorithm which selects executions from a job history based on their similarity to the current execution. We defined several similarity measures, which are used by SMiPE. We further presented techniques to evaluate these measures using a similarity quality measure. Based on the finding that algorithms have distinct profiles determining which of those measures are useful, we use algorithm-specific training of system parameters to adapt the similarity matching automatically to different jobs.

Currently, we include the current cluster state only implicitly from historical data and we assume a homogeneous cluster environment. In the future, SMiPE could explicitly incorporate data such as cluster-wide utilization, information on concurrent workloads or hardware specification of servers. Also, we want to integrate SMiPE with other work. For example, the progress estimation is useful for systems that dynamically adjust resource allocations when detecting given runtime targets are at risk of being missed. Further, many systems infer predictions from a job history, which can be improved by using the presented matching techniques for filtering relevant data. Although SMiPE is less accurate at the beginning of executions due to the absence of current data, the matching is already effective in selecting useful historic runs. Our cluster experiments show that SMiPE is able to predict remaining runtimes with a mean relative error of 9.1% to 13.1%.

Acknowledgements

This work has been supported through grants by the German Science Foundation (DFG) as FOR 1306 Stratosphere and by the German Ministry for Education and Research (BMBF) as Berlin Big Data Center BBDC (funding mark 01IS14013A).

References

[1] C. P. Chen and C.-Y. Zhang, "Data-intensive applications, challenges, techniques and technologies: A survey on Big Data," *Information Sciences*, vol. 275, 2014.

[2] M. Chen, S. Mao, and Y. Liu, "Big Data: A Survey," *Mobile Networks and Applications*, vol. 19, no. 2, 2014.

[3] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, 2008.

[4] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. USENIX, 2010.

[5] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, Í. Goiri, S. Krishnan, J. Kulkarni *et al.*, "Morpheus: Towards Automated SLOs for Enterprise Clusters." in *Symposium on Operating Systems Design and Implementation*, ser. OSDI'16. USENIX, 2016.

[6] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-Efficient and QoS-Aware Cluster Management," in *ACM SIGPLAN Notices*, vol. 49, no. 4. ACM, 2014.

[7] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger, "TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters," in *Eleventh European Conference on Computer Systems*. ACM, 2016.

[8] G. Mishne, J. Dalton, Z. Li, A. Sharma, and J. Lin, "Fast Data in the Era of Big Data: Twitter's Real-Time Related Query Suggestion Architecture ," in *2013 ACM SIGMOD International Conference on Management of Data*. ACM, 2013.

[9] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca, "Jockey: Guaranteed Job Latency in Data Parallel Clusters," in *7th ACM European Conference on Computer Systems*. ACM, 2012.

[10] L. Thamsen, I. Verbitskiy, F. Schmidt, T. Renner, and O. Kao, "Selecting Resources for Distributed Dataflow Systems According to Runtime Targets," in *2016 IEEE 35th International Performance Computing and Communications Conference (IPCCC)*. IEEE, 2016.

[11] P. Lama and X. Zhou, "Aroma: Automated Resource Allocation and Configuration of MapReduce Environment in the Cloud," in *9th International Conference on Autonomic Computing*. ACM, 2012.

[12] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics," in *Symposium on Networked Systems Design and Implementation*, ser. NSDI'16. USENIX, 2016.

[13] A. D. Popescu, A. Balmin, V. Ercegovac, and A. Ailamaki, "PREDIcT: Towards Predicting the Runtime of Large Scale Iterative Analytics ," *VLDB Endowment*, vol. 6, no. 14, 2013.

[14] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache Hadoop YARN: Yet Another Resource Negotiator," in *4th Annual Symposium on Cloud Computing*, ser. SOCC '13. ACM, 2013.

[15] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A Platform for Fine-grained Resource Sharing in the Data Center," in *Conference on Networked Systems Design and Implementation*, ser. NSDI'11. USENIX, 2011.

[16] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink: Stream and Batch Processing in a Single Engine," *IEEE Data Engineering Bulletin*, vol. 38, no. 4, 2015.

[17] 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 *Symposium on Operating Systems Design and Implementation*, ser. OSDI'14, vol. 14. USENIX, 2014.

[18] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A System for Large-Scale Graph Processing," in *2010 ACM SIGMOD International Conference on Management of Data*. ACM, 2010.

[19] M. J. Powell, "The BOBYQA Algorithm for Bound Constrained Optimization without Derivatives," *Cambridge NA Report NA2009/06, University of Cambridge, Cambridge*, 2009.

[20] J. Kunegis, "Konect: The Koblenz Network Collection," in *22nd International Conference on World Wide Web*. ACM, 2013.

[21] Z. Ming, C. Luo, W. Gao, R. Han, Q. Yang, L. Wang, and J. Zhan, "BDGS: A Scalable Big Data Generator Suite in Big Data Benchmarking," in *Workshop on Big Data Benchmarks*. Springer, 2013.