Continuously Improving the Resource Utilization of Iterative Parallel Dataflows

Lauritz Thamsen, Thomas Renner, Odej Kao
Technische Universität Berlin, Germany
{firstname.lastname}@tu-berlin.de

***Abstract*—Parallel dataflow systems like Apache Flink allow analysis of large datasets with iterative programs. However, allocating a cost-effective set of resources for such jobs is a difficult task as the resource utilization depends on many factors such as dataset size, key value distributions, computational complexity of programs, and the underlying hardware. What's more, some of these factors are not well known before the execution. There are, for example, often no data statistics such as key value distributions available beforehand.**

For this reason, we propose to improve the resource utilization at runtime using the repetitive nature of iterative dataflow programs. Based on runtime statistics gathered in previous iterations, the resource allocation is adapted dynamically at the synchronization barriers between iterations. This approach has two advantages: First, at barriers detailed statistics can be available, even for parallelly executed task pipelines. Second, at barriers dataflows can be adapted without complex handling of intermediate task state.

This paper presents a prototype integrated with Apache Flink and an evaluation on a cluster with 480 cores. One experiment shows a 57% reduction of the job runtime by allocating more resources for a shorter time, another experiment a release of up to 40% surplus resources without significantly extending the job runtime.

***Keywords*-Parallel Dataflows, Scalable Data Processing, Resource Utilization, Dynamic Scaling**

I. Introduction

Parallel dataflow systems like Apache Flink[1] and Naiad [2] allow to iteratively analyze large-scale datasets. The same dataflow is repeatedly executed until a termination criterion is met. Examples of these often long-running and compute-intensive programs include graph computations such as PageRank and Connected Components as well as machine learning programs such as k-means and Stochastic Gradient Descent (SGD). Cost-effective execution of such iterative dataflow programs requires a good utilization of allocated resources. This differs from underprovisioning, in which case the runtime could be decreased considerably by using more resources, and overprovisioning, in which case the allocated resources are not utilized well and the runtime would not increase significantly with fewer resources. Overprovisioning resources for a single job is a problem when the performance of entire workloads should be optimal, when resources are paid-per-use, or energy consumption is of importance. Additionally, using more resources often entails more synchronization. Therefore, overprovisioning can even extend the runtime of jobs.

[1]http://flink.apache.org, accessed 2016-02-05, originated from [1].

Choosing a cost-effective set of resources is inherently difficult as the performance of parallel dataflows depends on many factors such as the logic of arbitrary User-defined Functions (UDFs), the datasets in which key values often have unknown distributions, as well as the specifications of the underlying hardware. The most cost-effective set of resources can also change at runtime. Many iterative programs can, for example, be computed incrementally, so that the size of the data that needs to be considered in each iteration decreases over time.

There has been a lot of work on dynamically scaling the resource usage of stream processing systems, including recent work on automatically scaling distributed stream processing systems [3]–[6], yet this requires complex handling of intermediate program state. Other approaches focus on learning and predicting resource requirements for jobs based on previously executed workloads or short sample runs [7], [8]. However, these approaches require similar jobs to have been executed before or dedicated sample runs, which have to be representative for the actual job. There is work on actually optimizing the execution of batch jobs at runtime with Scope [9], which continuously adapts both the Degree of Parallelism (DoP) and partitioning, but only based on data statistics.

We propose to continuously improve the resource utilization at runtime by explicitly using the iterative nature of many dataflow jobs. Based on system statistics gathered in previous iterations, the resource allocation is adapted automatically for a cost-effective job execution. The recorded statistics are well applicable for upcoming iterations as the same parallel dataflow is executed on at least highly related data. Furthermore, the barriers between iterations provide a good opportunity to re-configure the execution without the necessity to migrate intermediate task state.

This paper presents an evaluation of both the impact of resource allocation on the runtime of exemplary iterative dataflow programs as well as of a prototype that iteratively adapts the resource allocation towards a target utilization. For the evaluation we used a 60-nodes cluster with 480 cores in total, Apache Flink, and PageRank as well as k-means.

Outline. The remainder of the paper is structured as follows. Section II presents the motivation for our research. Section III explains our approach. Section IV presents first results, including our prototype and an evaluation of it. Section V

summarizes the related work. Section VI concludes this paper.

II. Problem Statement

Many important programs are iterative. This includes many graph algorithms such as PageRank and machine learning programs such as k-means or SGD. As with other dataflow programs, key to a cost-effective execution of iterative programs is achieving a good resource utilization. For this, an appropriate set of resources needs to be selected for each job.

A. Cost-effective Resource Usage by Dataflow Jobs

Selecting a good set of resources is crucial for costs in terms of resource usage and energy consumption. Provisioning too many results in low resource utilization. In case of overprovisioning compute nodes, for example, for an ingestion rate that is bound by the network or by the read speed of disks there is not enough work to fully utilize the cores of all workers. However, in case of underprovisioning compute nodes the CPUs or memory become the bottleneck. Either CPU utilization is at 100% or out-of-core processing becomes necessary as, for example, hash tables for joins do not fit the main memory anymore. Underprovisioning results in poor application performance, overprovisioning in poor resource utilization with unnecessarily high costs and energy consumption.

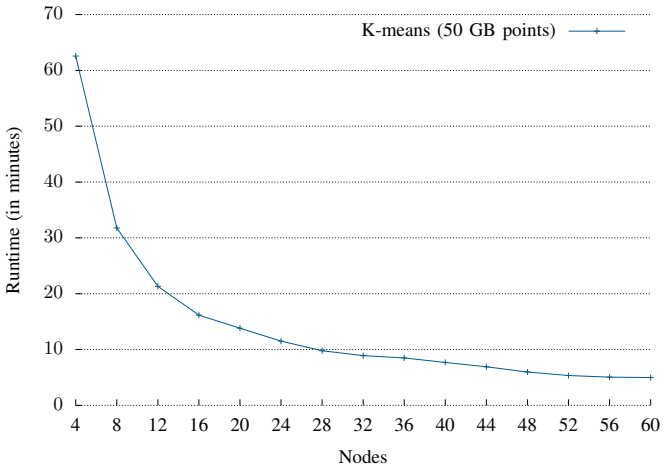

Fig. 1: Runtime of computing k-means of 50 GB of three-dimensional points using different scale-outs.

Figure 1 and 2 show two experiments that highlight the impact of resource allocations on the runtime of two parallel dataflow jobs using the experimental setup described in IV-A. The graphs each show a Flink job that is executed multiple times using different shares of the cluster, from four to 60 nodes. The job shown in Figure 1 is finding five clusters in 50 GB of three-dimensional points[2] using Lloyd's algorithm for k-means clustering [10]. k-means groups elements into k clusters in which each element belongs to the cluster with the nearest mean. The job shown in Figure 2 is computing the importance of twitter users from the follower graph of 1 GB of the *twitter.rv* dataset [11], using PageRank and 50 iterations. PageRank is an iterative algorithm for ranking nodes (pages) in a graph based on the link structure. Figure 1 shows that the benefit of using more nodes for computing the k-means decreases with larger cluster shares. Looking at Figure 2, however, running the PageRank job on increasingly large cluster shares is not just a waste of resources but actually increases the runtime significantly from a certain scale onwards.

[2]Equally many points were generated around five randomly placed cluster centers following a uniform distribution for the job.

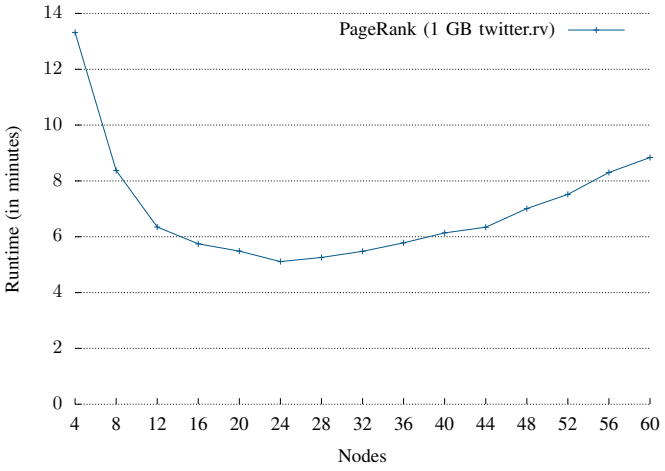

Fig. 2: Runtime of computing PageRanks of 1 GB of twitter follower relations using different scale-outs.

Choosing the optimal set of resources is a difficult task. Programs may, for example, be more I/O-bound (i.e. operators are mostly waiting for incoming elements) or more CPU-bound (i.e. elements are not processed as fast as they come in) [12]. Moreover, even for a fixed ingestion rate, knowing which DoP is most effective for the different operators and arbitrary UDFs of program plans is not trivial.

Furthermore, without detailed data statistics, which often are not available for data processed directly from distributed file systems, it is unclear how well the data partitioning will work for a resource reservation. Also, the upper bound of parallel execution threads also depends on the distribution of key values when elements are grouped.

How this translates to actual hardware, especially when jobs are executed in virtualized environments, is also difficult to know beforehand.

B. Changing Factors with Delta Iterations

Which set of resources is most cost-effective can also change over time. One important factor for the optimal set of resources for a job is, for example, the dataset size. The data that has to be analyzed in each iteration can, however, change over time.

Figure 3 shows an experiment, in which 25 GB of the *twitter.rv* dataset are processed by a PageRank Flink job, using the same experimental setup as before and a cluster share of 20 nodes. PageRank computes the ranks of pages iteratively and converges when the ranks do not change anymore. The ranks do not necessarily converge at the same speed. Therefore, only pages that did change in the last iteration have to be considered in the next iteration. What is considered a significant change in the rank between iteration can be configured by the user. The user provides this threshold as a parameter with the job. The graphs in Figure 3 show how many pages are processed in each iteration for three different thresholds. This shows that the dataset size can not only change over time, but also dependent on a freely choosable program parameter, leading to different optimal sets of resources over time.

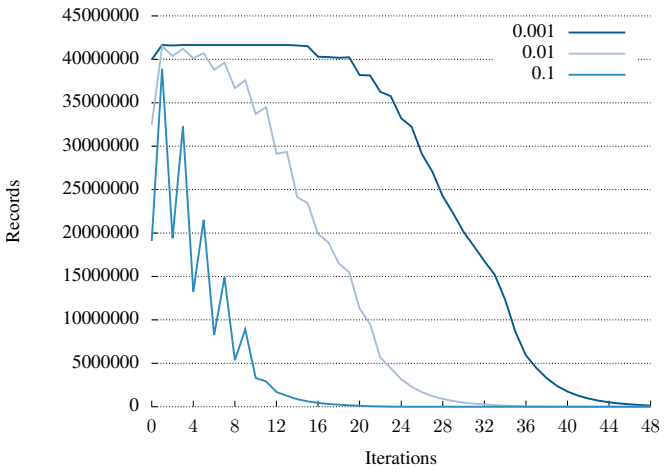

Fig. 3: Elements analyzed per iteration when computing the PageRanks from 25 GB of twitter follower relations (using different thresholds for when a page's rank is considered stable).

C. Utilization Of Production Clusters

Studies show that users are not effective at allocating appropriate sets of resources with a clear tendency to over-provisioning. A utilization analysis for a data analysis cluster in productive use at Twitter [8] shows, for example, that the aggregate CPU utilization is consistently below 20%, even though the reservations reach close to 80% of the total capacity. Memory utilization is between 40-50% but still differs a lot from the reserved capacity. Similarly, a production Google cluster managed by Google's Borg [13] achieves aggregate cpu utilization of 25%-35% and aggregate memory utilization of 40%, while reserved resources exceed 75% and 60% of the available CPU and memory capacities [14].

III. Continuously Improving the Resource Utilization of Iterative Parallel Dataflows

This section presents our approach for improving the resource utilization of iterative parallel dataflows.

A. Adapting Parallel Dataflows at Iteration Barriers

As described in detail in Section II, selecting a cost-effective set of resources upfront is inherently difficult. For this reason, we propose to adapt reservations at runtime using the iterative nature of many dataflow jobs. In particular, our approach is to learn from previous parallel dataflow iterations for future ones, as shown in Figure 4: the resource allocation for each new iteration is done based on previously recorded system statistics.

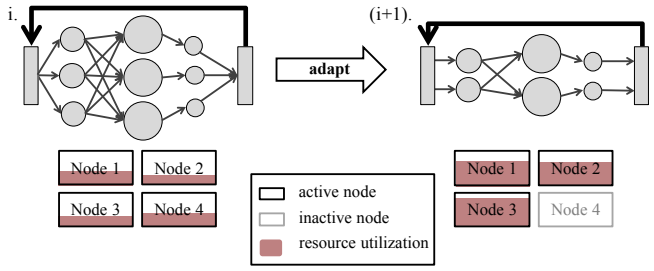

Fig. 4: Improving the resource utilization of upcoming parallel dataflow iterations based on statistics from previous iterations.

Using the iteration barrier for system-level reconfigurations like resource reservations has two main advantages:

- *Detailed statistics:* At iteration barriers, detailed statistics that reflect all elements processed in the previous iterations can be available. In comparison to collecting statistics in each stage of tasks for the next stage of tasks, this approach to collecting statistics is also applicable to execution models that execute pipelines of tasks in parallel, not only to execution models in which tasks are executed stage by stage.
- *No intermediate task state:* At iteration barriers, all tasks of the previous iteration have finished and the tasks for the next iteration have not been started yet. Therefore, there is no intermediate task state that has to be migrated for any scaling decisions. This makes the implementation of dynamic scaling less complex and, thus, less error-prone. Furthermore, no intermediate task state has to be sent around and replicated, saving time and space.

Therefore, our approach allows straightforward implementations of dynamic scaling using statistics that reflect all elements of the dataflow.

B. Computing Resource Reservations

Our approach allows to adapt resource allocations using arbitrary statistics gathered in previous iterations. Relevant statistics for computing appropriate resource reservations include:

- *Current Resource Utilization*: utilization of cores, main memory, disks, and network interfaces.
- *Dataset Characteristics*: dataset size, number of elements, and key value distribution.

In general, our solution is not limited to taking only statistics from the last iteration into account, but can incorporate

information from the entire iteration history. Thus, it is, for example, possible to detect behavior like converging or oscillating dataset sizes when datasets are processed incrementally in iterations.

Currently, we compute resource allocations for each iteration based on the CPU utilization during the last iteration. In particular, we use the average CPU utilization of all involved workers in the last iteration. Based on this average and the current task parallelism, the parallelism for the next iteration is computed using the following formula:

$$DoP_{i+1} = \frac{currentCPUAvg_i}{targetCPUAvg} \cdot DoP_i$$

This DoP is then checked for sanity using the available resources as upper bounds and a sensible minimum for parallel execution in a cluster as lower bounds (i.e. using two full nodes). Afterwards, the minimal set of resources is chosen that can accommodate this number of parallel tasks. These resources are then used for the execution of the next iteration.

IV. Results

This section presents our testbed, our current Apache Flink-based prototype, and two experiments evaluating the prototype.

A. Experimental Setup

All experiments were done using a cluster of 60 machines. Each of the nodes is equipped with a quad-core Intel Xeon CPU E3-1230 V2 3.30GHz (4 physical cores, 8 logical ones), 16GB RAM, and three 1 TB disks. All nodes are connected through a single switch and 1 Gigabit Ethernet. Each node runs Linux (kernel version 3.10.0), Java 1.8.0, and a customized Flink 0.10.0[3]. We configured Flink to allocate 10 GB of the main memory, to use 2 GB for network buffers, and to provide eight task execution slots per worker.

Every experiment was executed 7 times. We report the median runtime.

B. Prototype Implementation

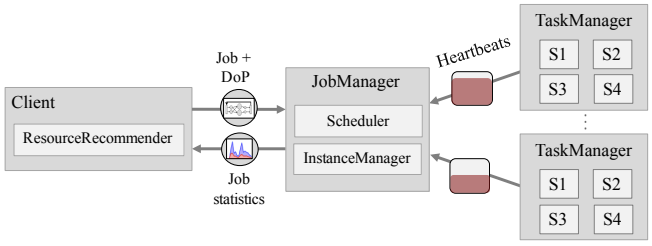

Fig. 5: The client re-computes the task parallelism between iterations using utilization statistics from all involved worker nodes.

We implemented our approach prototypically using Apache Flink. The prototype automatically optimizes the resource utilization at runtime towards a specified utilization target. For this, we instrumented Flink to periodically record the utilization on all `TaskManagers` using the Java Management Extensions (JMX)[4]. Specifically, the `TaskManagers` record the recent CPU utilization every 250 milliseconds.

As shown in Figure 5, the `TaskManagers` send these samples as heartbeats to the `JobManager`, where the utilization is collected for each iteration. At the barriers between iterations, the `JobManager` sends the utilization data of the last iteration for all involved `TaskManagers` to the client. The client uses the utilization data and the current DoP to compute the task parallelism for the next iteration. For this, the client uses the `ResourceRecommender` component, which implements the formula described in Section III-B, and then starts the next iteration with the new DoP. The DoP equals the number of execution slots that are required to schedule and execute the job. Our prototype allocates as few nodes as necessary to accomodate the parallel task instances. Given, for example, eight execution slots per TaskManager as in our experimental setup IV-A, a DoP of 40 would result in five nodes being used for the job.

C. Iteratively Optimizing CPU Utilization

Our current prototype iteratively optimizes the resource utilization towards a utilization target. We set this target to 70% for the experiments presented here.

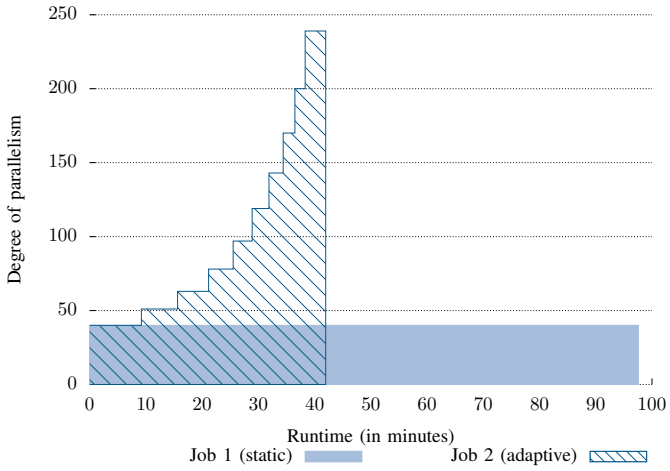

Fig. 6: Automatic adaptation of the resource allocation for k-means on 100 GB input data compared to static underprovisioning.

Figure 6 presents the results of an experiment in which 100 GB of three-dimensional points[5] were clustered into 10 groups using a Flink job implementing k-means. We initially allocated five of the sixty nodes, representing a significant

[3]http://github.com/apache/flink/tree/release-0.10.0, accessed 2016-01-21

[4]https://docs.oracle.com/javase/7/docs/jre/api/management/extension/com/sun/management/OperatingSystemMXBean.html, accessed 2016-01-29

[5]Equally many points were generated around ten randomly placed cluster centers following a uniform distribution.

underprovisioning for the job. The experiment shows how our prototype dynamically adapts the resource allocation and reserves more nodes over time compared to the static reservation. In the adaptive case, up to 30 nodes were used during the execution of this job. In the static case only the five initially reserved nodes were used. The runtime of the two median runs were 2518.8 seconds (adaptive) compared to 5855.0 seconds (static). Adding up how many nodes were used for how long–yielding the areas of the jobs in Figure 6–shows that in the adaptive case 5.4% more resources were used (30868 instead of 29276 node-seconds). However, this 5.4% increase in resource usage coincides with the job finishing in less than half the time (230% speedup).

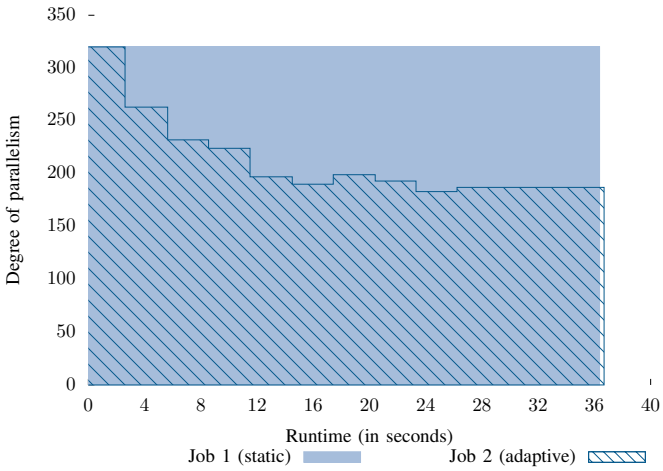

Fig. 7: Automatic adaptation of the resource allocation for k-means on 2 GB input data compared to static overprovisiong.

Figure 7 presents the results of two experiments in which 2 GB of three-dimensional points were clustered into 10 groups using the same k-means Flink job. We initially allocated 40 of the 60 nodes, representing a significant overprovisioning for the job. This experiment shows how the prototype dynamically scales-in towards using a smaller share of the cluster. In the adaptive case, the reservation is decreased by 16 nodes towards 24 at the end of the job since each node provides eight execution slots. The runtime of the two median runs were 36.7 seconds (adaptive) compared to 36.3 seconds (static). However, only two thirds of the resources were used in the adaptive case (978 versus 1455 node-seconds).

V. Related Work

This section first presents systems that support iterative parallel dataflows, use statistics to optimize the execution at runtime, or automatically select resources for a job.

A. System Support for Iterative Parallel Dataflows

Multiple parallel dataflow systems provide dedicated support for iterative programs. Examples include Twister [15] and HaLoop [16], both extensions to the execution and programming model proposed with MapReduce [17]. Systems like Flink [1] and Naiad [2] also support cyclic task graphs and iterative programs, but further support incremental processing. Flink, for example, distinguishes two different types of iterations: *Bulk Iterations* and *Delta Iterations* [18]. With Bulk Iterations, each iteration fully consumes the previous iteration's result and computes a new result. In contrast, Delta Iterations evolve the result by changing only parts of the data between iterations. An example for a program that can make use of these Delta Iterations is Connected Components, in which only a change in a vertex's component membership requires a vertex to propagate its membership in the next iteration. Thus, only vertices which changed in an iteration have to be considered in the next one, saving a lot of computation compared to always considering all vertices. Naiad provides a special kind of incremental processing, called *Differential Dataflow* [19], which is based on transferring only and computing on differences. These differences are also stored indexed by two-dimensional timestamps, indicating both the iteration and the input epoch from which they stem. By storing the differences for each iteration and epoch, Naiad is able to quickly provide new results whenever the original input changes.

B. Runtime Optimization of Parallel Dataflows

Many works (e.g. [3]–[6]) have investigated dynamic scaling for large-scale distributed stream processing. In contrast to the solution here, dynamic scaling for stream processing needs to handle intermediate program state appropriately. This increases the complexity of the systems' implementation and also introduces an overhead before dynamic scaling can be done without losing intermediate state.

Also in the context of large-scale stream processing, there is work on adaptively placing and migrating tasks at runtime based on statistics such as to place tasks that exchange comparably large amounts of data onto the same hosts [20]. This solution, however, does neither alter the task parallelism nor the size of the resource reservation.

In batch processing, Scope [9], [21] is a system that alters the task parallelism at runtime based on statistics. Furthermore, Scope also uses these statistics to continuously optimize program plans, to select optimal physical operators, and to adapt the partitioning at runtime. These optimizations are, however, only based on data statistics and the solution is assuming an execution in stages, in which statistics about all elements of a dataflow can be gathered before the next stage is started. This is not the case with pipelined execution of subsequent tasks. In contrast, our approach is based on gathering statistics for an entire batch of elements in previous iterations.

The Jockey scheduler [22] for Scope uses a simulator for predicting the remaining runtime at different resource allocations and in different stages of the job. The prediction is done using previous runs of a job and using a utility function that links the resource allocation to job performance. The reservation for the next stage is then computed according to user-specified performance constraints. In contrast, our solution optimizes towards a utilization target without requiring

knowledge about previous runs of the job.

C. Automatic Resource Allocation for Large-Scale Data Analysis Jobs

Quasar [8] is a resource management systems that automatically makes resource reservations. For this, Quasar uses a model trained on the previous workload and short dedicated sample runs. The automatic reservation is then done according to user-defined performance constraints. Quasar considers the number of nodes, the types of nodes, and interference between jobs, but is dependent on the quality of the trained model.

The ThroughputScheduler [23] is a scheduler for Hadoop workloads that uses a Bayesian learning scheme to determine the resource requirements of jobs on the fly after probing the capabilities of the available hardware with short sample jobs. The scheduler, however, specifically assumes MapReduce as execution model.

VI. Conclusion

In this paper, we proposed to dynamically optimize the resource utilization of iterative parallel dataflows. Based on runtime statistics from previous iterations, each iteration's resource reservation is recomputed to use resources more efficiently. The presented prototype based on Apache Flink shows that dynamically scaling resource allocations of iterative dataflows based on CPU utilization is a promising approach. The presented experiments show that overprovisioned compute nodes are discarded without any significant increase in runtime, while more compute nodes are automatically added to the reservation in case of underprovisioning, speeding up the execution significantly.

In the future, we want to optimize the utilization of more resources than just compute cores. We are also planning to use machine learning to find the most cost-effective utilization targets for jobs. Furthermore, we want to investigate how the internal allocation of reserved resources towards different system functions can be improved at runtime.

Acknowledgments

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] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke, "The Stratosphere Platform for Big Data Analytics," *The VLDB Journal*, vol. 23, no. 6, pp. 939–964, December 2014.

[2] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A Timely Dataflow System," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, November 2013, pp. 439–455.

[3] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez, "StreamCloud: An Elastic and Scalable Data Streaming System," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 12, pp. 2351–2365, December 2012.

[4] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating Scale Out and Fault Tolerance in Stream Processing Using Operator State Management," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. ACM, June 2013, pp. 725–736.

[5] B. Lohrmann, P. Janacik, and O. Kao, "Elastic Stream Processing with Latency Guarantees," in *Proceedings of the 35th IEEE International Conference on Distributed Computing Systems*, ser. ICDCS'15, June 2015, pp. 399–410.

[6] B. Satzger, W. Hummer, P. Leitner, and S. Dustdar, "Esc: Towards an Elastic Stream Computing Platform for the Cloud," in *Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing*, ser. CLOUD '11. IEEE, July 2011, pp. 348–355.

[7] C. Delimitrou and C. Kozyrakis, "Paragon: QoS-aware Scheduling for Heterogeneous Datacenters," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. ACM, March 2013, pp. 77–88.

[8] ——, "Quasar: Resource-efficient and QoS-aware Cluster Management," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. ACM, March 2014, pp. 127–144.

[9] N. Bruno, S. Jain, and J. Zhou, "Continuous Cloud-scale Query Optimization and Processing," *Proc. VLDB Endow.*, vol. 6, no. 11, pp. 961–972, August 2013.

[10] S. Lloyd, "Least Squares Quantization in PCM," *IEEE Transactions on Information Theory*, vol. 28, no. 2, pp. 129–137, September 2006.

[11] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a Social Network or a News Media?" in *Proceedings of the 19th International Conference on World Wide Web*, ser. WWW '10. ACM, April 2010, pp. 591–600.

[12] D. Battre, M. Hovestadt, B. Lohrmann, A. Stanik, and D. Warneke, "Detecting Bottlenecks in Parallel DAG-based Data Flow Programs," in *2010 IEEE Workshop on Many-Task Computing on Grids and Supercomputers*, ser. MTAGS'10. IEEE, November 2010, pp. 1–10.

[13] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale Cluster Management at Google with Borg," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15. ACM, April 2015, pp. 18:1–18:17.

[14] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis," in *Proceedings of the Third ACM Symposium on Cloud Computing*, ser. SoCC '12. ACM, October 2012, pp. 7:1–7:13.

[15] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: A Runtime for Iterative MapReduce," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. ACM, June 2010, pp. 810–818.

[16] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "HaLoop: Efficient Iterative Data Processing on Large Clusters," *VLDB*, vol. 3, no. 1-2, pp. 285–296, September 2010.

[17] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation*, ser. OSDI'04. USENIX Association, January 2004, pp. 10–10.

[18] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl, "Spinning Fast Iterative Data Flows," *Proc. VLDB Endow.*, vol. 5, no. 11, pp. 1268–1279, July 2012.

[19] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard, "Differential Dataflow," in *Proceedings of the 6th Conference on Innovative Data Systems Research*, ser. CIDR'13. CIDR 2013, January 2013.

[20] L. Aniello, R. Baldoni, and L. Querzoni, "Adaptive Online Scheduling in Storm," in *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, ser. DEBS '13. ACM, June 2013, pp. 207–218.

[21] N. Bruno, S. Agarwal, S. Kandula, B. Shi, M.-C. Wu, and J. Zhou, "Recurring Job Optimization in Scope," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12. ACM, May 2012, pp. 805–806.

[22] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca, "Jockey: Guaranteed Job Latency in Data Parallel Clusters," in *Proceedings of the 7th ACM European Conference on Computer Systems*, ser. EuroSys '12. ACM, April 2012, pp. 99–112.

[23] S. Gupta, C. Fritz, B. Price, R. Hoover, J. Dekleer, and C. Witteveen, "ThroughputScheduler: Learning to Schedule on Heterogeneous Hadoop Clusters," in *Proceedings of the 10th International Conference on Autonomic Computing*, ser. ICAC'13. USENIX, June 2013, pp. 159–165.