Machines Tuning Machines: Configuring Distributed Stream Processors with Bayesian Optimization

Fischer, Lorenz ; Gao, Shen ; Bernstein, Abraham

Abstract: Modern distributed computing frameworks such as Apache Hadoop, Spark, or Storm distribute the workload of applications across a large number of machines. Whilst they abstract the details of distribution they do require the programmer to set a number of configuration parameters before deployment. These parameter settings (usually) have a substantial impact on execution efficiency. Finding the right values for these parameters is considered a difficult task and requires domain, application, and framework expertise. In this paper, we propose a machine learning approach to the problem of configuring a distributed computing framework. Specifically, we propose using Bayesian Optimization to find good parameter settings. In an extensive empirical evaluation, we show that Bayesian Optimization can effectively find good parameter settings for four different stream processing topologies implemented in Apache Storm resulting in significant gains over a parallel linear approach.

DOI: https://doi.org/10.1109/CLUSTER.2015.13

Machines Tuning Machines: Configuring Distributed Stream Processors with Bayesian Optimization

Lorenz Fischer
Department of Informatics
University of Zurich
Switzerland
lfischer@ifi.uzh.ch

Shen Gao
Department of Informatics
University of Zurich
Switzerland
shengao@ifi.uzh.ch

Abraham Bernstein
Department of Informatics
University of Zurich
Switzerland
bernstein@ifi.uzh.ch

***Abstract*—Modern distributed computing frameworks such as Apache Hadoop, Spark, or Storm distribute the workload of applications across a large number of machines. Whilst they abstract the details of distribution they do require the programmer to set a number of configuration parameters before deployment. These parameter settings (usually) have a substantial impact on execution efficiency. Finding the right values for these parameters is considered a difficult task and requires domain, application, and framework expertise.**

In this paper, we propose a machine learning approach to the problem of configuring a distributed computing framework. Specifically, we propose using Bayesian Optimization to find good parameter settings. In an extensive empirical evaluation, we show that Bayesian Optimization can effectively find good parameter settings for four different stream processing topologies implemented in Apache Storm resulting in significant gains over a parallel linear approach.

***Keywords*-distributed stream processing; configuration; optimization; Storm**

I. Introduction

The configuration of a distributed system is crucial for both good performance and to prevent system failures [1]. Many modern distributed programming frameworks offer a wide range of configuration parameters for tuning purposes. The performance of a system deployment depends on the interplay between all parameters with the implementation of the application logic, the underlying hardware, as well as the data that is processed by the system. Hence, choosing suitable configuration parameters given a system implementation and associated infrastructure can be difficult and requires expert knowledge of both the problem domain and the technology used to build the system. Even experts require careful experimentation as the interactions between different parameters are hard to predict.[1]

To address this tedious manual parameter experimentation, this paper proposes an automated process based on Bayesian Optimization for finding optimal parameter configurations. Specifically, we present empirical results from a series of experiments in which we evaluated the suitability of Bayesian Optimization for the configuration of distributed stream processing systems built using Apache Storm.[2] Our contributions are:

- We present an *auto-configuration approach for distributed stream processing systems (SPS) using Bayesian Optimization.*
- We provide an *extensive empirical evaluation* showing the effectiveness of our approach on a cluster of 80 machines (320 cores) running Storm topologies (applications) of varying sizes and characteristics.
- We introduce a reusable *benchmark consisting of a set of operator graphs as well as generation approach.*

The remainder of this paper is organized as follows: next, we first present related work. We then describe the system used for the evaluations and the give an introduction to Bayesian Optimization in Section III. Our experimental setup and results are presented in Sections IV and V, respectively. We close with conclusions in Section VI.

II. Related Work

A. Configuration of distributed stream processing systems

The problem of how to configure distributed (stream) query systems [2] and how to react to dynamically changing properties of stream processors [3] has been extensively studied in the past decade. Often, cost models have been proposed to capture complexities of these systems [4], [5] to optimize the use of resources [4], [6] or query execution [7], [8]. Others have applied Covariance Matrix Adaption (CMA) [9] or searched the parameter space using an experimentally constructed parameter dependency graph [1]. The problem we tackle in this paper differs from the problem of cost-model based solutions in two aspects: first, we do not aim at changing the structure of the streaming application (or the query), but focus on tuning of configuration parameters to make the execution more efficient. Second, we do not attempt to build a complete (closed-form) mathematical model of the system, but treat the application as a blackbox function that we optimize using empirical sampling. In

[1]www.slideshare.net/miguno/apache-storm-09-basic-training-verisign

[2]https://storm.apache.org

contrast to the approaches presented in [9] and [1], we employ a probabilistic bayesian approach.

Similar to our goal, some studies have focused on one specific parameter: the degree of parallelization. One line of work investigates auto-parallelization – the process of automatically choosing the degree of parallelism for operators in a task graph [10]. It has been extensively studied in the realm of IBM's System-S [11] as a theoretical model. Schneider et al. [12], [13] extended these results and presented an algorithm to dynamically change the workload on operators in response to changes in the incoming data stream. In contrast to pure auto-parallelization, our approach treats the parallelism of each operator of the topology as only one of many system parameters that need to be tuned. Note that we do not cover dynamic auto-parallelization as we assume mostly static workloads.

B. *Applications of Bayesian Optimization*

Bayesian Optimization [14] is a probabilistic technique to optimize systems with unknown cost functions. It has successfully been applied in cases where the performance of systems is strongly dependent on configuration parameters, and no mathematical closed-form cost model is known such as finding good hyperparameter settings in machine learning problems (e.g., classification [15], [16], [17] or feature selection [16]). There are several Bayesian Optimization frameworks (e.g, Spearmint[3] [17], SMAC[4][18], HyperOpt[5], or BayesOpt[6]) available for research. We are not aware of any previous work that has investigated the applicability of Bayesian Optimization for the configuration of distributed systems or for distributed stream processing systems in particular.

III. System Description

This section describes how we employ Bayesian Optimization to configure a distributed stream processing system based on the Storm distributed realtime computation framework. We first give a short introduction into Storm and Trident; the two technologies we use to implement our experiments. We then formally describe the process of Bayesian Optimization before presenting Spearmint, the optimizer used in the experiments.

A. *Distributed Stream Processing with Storm*

Many distributed computation frameworks have been proposed in recent years. One representative of such a framework aimed at distributed stream processing is the Storm framework. In contrast to batch-based distributed systems such as Apache MapReduce,[7] Storm ingests data continuously. As in MapReduce, a Storm application allows the user to partition the data and to distribute parts of the processing across a compute cluster.

[3]https://github.com/HIPS/Spearmint
[4]http://www.cs.ubc.ca/labs/beta/Projects/SMAC
[5]http://jaberg.github.io/hyperopt
[6]http://rmcantin.bitbucket.org/html
[7]http://hadoop.apache.org

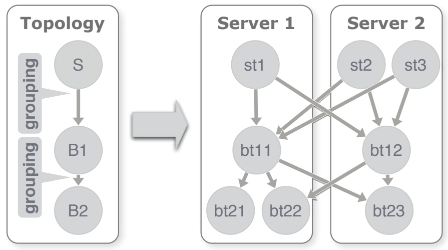


Figure 1. Logical (left) and physical (right) representation of a topology. The spout (S) and bolt nodes (B1, B2) are instantiated as spout task instances (st1-st3) and bolt task instances (bt11-bt23) across two servers.

A Storm application—a *topology*—is a directed graph consisting of spout and bolt nodes as depicted in Figure 1 on the left. Spouts emit data to downstream nodes. Bolts consume data from upstream nodes and emit data to downstream nodes. Spout nodes are typically used to connect a Storm topology to external data sources such as queues, web-services, or file systems. For each spout and bolt, the programmer defines how many instances of this node should be created in the physical instantiation of the topology – the task instances. This results in a physical topology depicted on the right of Figure 1, which is different from the logical representation. The parameter used to define the degree of parallelism of a node is called a *parallelism hint*, as Storm may change these hints for consistency purposes. The task instances, or *tasks*, are distributed across all machines of the compute cluster to which a topology has been assigned. Each edge in the topology graph defines a *grouping strategy* according to which messages that pass between the nodes—the *tuples*—are sent to downstream nodes.

Tuples are lists of key-value pairs. The programmer defines the tuple format for each edge of the topology (e.g. field1=query_terms, field2=browser_cookie, field3=timestamp). This format cannot be changed at runtime. Different grouping strategies provide different guarantees. For example, the field grouping strategy guarantees, that all tuples that share the same value in one or multiple configurable fields are sent to the same task instance.

Higher level operators such as aggregators, state handling, functions, and filters are provided by *Trident*, a programming framework that is part of the Storm distribution. Further, Trident may combine multiple operators into larger units. In such cases, Storm overrides the parallelism-hints specified by the programmer in order to prevent frequent reshuffling of data across the network. This is similar to the *SPADE* system [19], which also fuses several operators into one processing element (PE) in System-S. In Trident, tuples are processed in mini-batches, offering consistency guarantees on a per-batch basis.

Having introduced the basic building blocks of a

Parameter	Description
Worker Threads	Number of threads per worker
Receiver Threads	Number of receiver threads per worker
Ackers	Number of acker tasks
Batch Parallelism	Number of batches being processed in parallel
Batch Size	Number of tuples in each batch
Parallelim Hints	Number of task instance to create for operators

Table I: Configuration parameters.

Storm/Trident application, we will describe the various ways in which such an application can be configured and tuned in the next section.

B. Configuration Parameters

Storm offers a number of configuration parameters that allow the programmer, as well as the system administrators, to configure various aspects of the system. Table I lists the parameters that we used in our evaluations: parameters that are most commonly tuned are the already mentioned parallelism hints, the batch size, and the batch parallelism[8] of a topology. Trident guarantees consistency on a per-batch level. This means that multiple batches can run at the same time, which can increase overall performance. Note that the "parallelism hints" parameter is not one single value, but a list of values that contains one number for each node of the topology. Hence, for topologies of greater size, the parameter space, naturally, becomes large. Other parameters that we included in this study are concurrency related parameters such as the size of the thread pool available to each worker, the number number of threads each worker starts to receive messages, and the degree of parallelism of the "acker" system bolt, i.e. the number of "acker" task instances, that Storm uses for its bookkeeping facility.

While any single one of these configuration options impacts the runtime behavior, overall performance is a result of the combination of all of these parameters working together. For example, consider the situation in which we set the parallelism hint of the spout in the topology depicted in Figure 1 to 10, but the parallelism hint for all bolts to 1. In this situation, the performance will most likely be bottlenecked by the code in the bolts of the topology. If, on the other hand, the parallelism hints for the bolts are set to 100, the new bottleneck would most likely be the code in the spout node. Similarly, there are interactions between the parameters for the batch size and the batch parallelism. Because these interactions are not only dependent on the values of these parameters themselves, but also on other aspects such as the available network infrastructure, disk speed, or availability of memory storage, making predictions about the resulting performance of the overall system is difficult. Additionally, framework properties, such as the automatic operator fusion of Trident, further obfuscate the impact of any single parameter. To tackle the problem of choosing good configuration parameters, we investigate the possibility of having a computer program choose these parameters. To this end, we employ the technique of Bayesian Optimization.

[8]Batch parallelism is also called pipeline parallelism in the literature.

C. Bayesian Optimization

In this sub-section, we give a short introduction to Bayesian Optimization. We refer to [20] and [17] for a more detailed introduction into the topic. Bayesian Optimization has first been proposed by Jonas Mockus as an optimization strategy for situations in which the objective function is a non-convex blackbox function [14] (i.e., a function for which no closed-form solution or derivative is known). The function is assumed to be Lipschitz-continuous (i.e., smooth and does not change dramatically). Also, sampling the function is assumed to be costly, either in terms of time or money. Thus, it can pay off to invest computational resources into computing the point in the parameter space where to sample next. For our domain, we assume the function to be the actual system performance of our distributed stream processor, given all the configuration parameters chosen. Obviously, given the black-box nature of the system, no mathematical representation exists, and determining the value of the function given certain parameter settings is achieved by running the system on a cluster with these settings and, hence, is costly. The process of choosing the next set of parameters is conducted using a Bayesian approach, which combines our prior assumptions about the function with the observed performance from previous runs. Borrowing the notation from [20] we can describe this formally as follows:

$$P(M|E) \propto P(E|M)P(M)$$

The probability distribution over our model M (our blackbox function) given some observed evidence E (our sampling runs) is proportional to the likelihood of E given the model times the prior probability of the model. Thus, we reason about the likelihood of observing the results of an evaluation run, given our prior beliefs about how the system would change in response to parameter modifications. Using the results of each evaluation run, a posterior distribution $P(M|E)$ is computed and integrated into the model. The decision of where to sample next is made by maximizing an acquisition function. There are various ways in which this acquisition function can be modeled. Often, Gaussian Processes are used to model the noise within the acquisition function. The purpose of the acquisition function is to balance the tradeoff between exploration and exploitation. The goal is to sample the next measurement in a region where either the uncertainty of the expected performance is high, the expected performance is high, or both.

More formally, again borrowing the notation from [20], we can describe the process as follows: Bayesian Optimization is an iterative process in which we sample an objective function repeatedly. We define x_t to be the t-th sample and

$y_t = f(x_t) + \epsilon_t$ to be the measured performance of our algorithm for run t, where f is our blackbox target function and ϵ_t is noise, which is typically assumed to be Gaussian. Our prior believes about f can be expressed as a prior distribution $P(f)$. We then collect observations (measured samples) and add them to the set $D_{1:t} = \{x_{1:t}, y_{1:t}\}$ of all evidence to date. In each step, we update our posterior belief with the newly collected evidence:

$$P(f|D_{1:t}) \propto P(D_{1:t}|f)P(f)$$

The new evidence is used to fit a Gaussian Process (GP) that describes our prior believes of how f is distributed:

$$f(x) \sim GP(m(x), k(x, x'))$$

where m is the mean function at position x and k is the covariance function depending on x as well as on the closest perviously sampled point at x'. The result is a function estimating the expected performance of any parameter value combination given some confidence interval. An acquisition function $u(x|D)$ is built using these two parameters (expected performance and confidence intervals) that are derived from the data D. The goal of the acquisition function is to create a tradeoff between exploration (try points with high uncertainty/variance) and exploitation (try points with a high expected performance). Hence, the next sample point x is determined by maximizing $u(x)$ (i.e. the x where the tradeoff between exploration and exploitation is optimal):

$$x_{t+1} = argmax_x u(x|D_{1:t})$$

There are several different ways of defining the acquisition function such as *Probability of Improvement* (PI), *Expected Improvement* (EI), or *GP Upper Confidence Bound* to name the most common ones. In this paper, we use Expected Improvement [21], as it provides a good tradeoff between exploration and exploitation and it is the method implemented in *Spearmint*, the toolkit we use in our experiments. The Expected Improvement acquisition function proposed by Mockus [21] is defined as:

$$x_{t+1} = argmax_x \mathbb{E}(max\{0, f_{t+1}(x) - f^{max}\}|D_{1:t})$$

where f^{max} is the best solution in the first t samples, so the next x would be chosen at the position, where the expected improvement between the new sample point ($f_{t+1}(x)$) and the current best sample point (f^{max}) is maximized. In contrast to the original optimization problem (our blackbox function), we can derive a closed-form expression for this problem, which can then be maximized using an analytical approach. We refer to [20] for all details.

As already mentioned in section II-B, there exist a number of freely available programming toolkits that implement Bayesian Optimization. In this project, we leverage Spearmint for the following reasons: first, it showed good performance in comparison with other main-stream Bayesian Optimization frameworks [22]. Second, it is well documented and its source code is openly available. Last, it supports pausing and resuming the optimization process, a feature that turned out to be important in our evaluation setup.

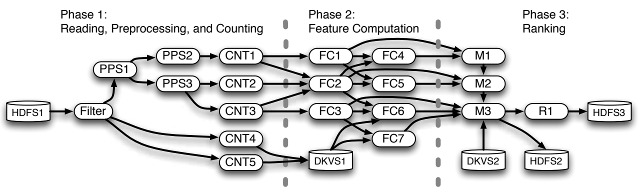


Figure 2. High-level architecture of Sundog (from [23]).

IV. Experimental Design

To evaluate the usefulness of Bayesian Optimization for parameter configuration of an SPS, we conducted a series of experiments using one real world application and three synthetic topologies. This section describes these topologies and the experimental setup.

A. Sundog: A Real World Topology

The first topology is a modified version of the Sundog entity ranking system [23]. Entity ranking systems consume search logs, tweets, etc., and rank search results based on co-occurence statistics. Figure 2 gives a a high level overview of the topology: in the first phase, input data is read from the *Hadoop Distributed Filesystem* (HDFS). Then, all input lines that do not contain at least one term of a predefined dictionary are filtered out. From this reduced data stream, statistics such as the number of term occurrences are built. These values are stored in an external distributed key-value store (DKVS1) to enable access from all compute tasks in later phases of the processing pipeline. For other statistics, we first build entity pairs from the terms in a series of preprocessing steps (PPS1-3) to count the number of search events and unique users for each entity and entity pair. Wherever possible, the relevant data is partitioned to allow parallelization to multiple compute nodes. The second phase consists of computing the actual feature metrics from the counter values (FC1-7). In the final phase (phase 3), the computed features are merged and complemented with semi-static features that are read from a table in the distributed key-value store (DKVS2 in Figure 2). Semi-static features such as the semantic type of an entity do not change often (or not at all). After merging all features, a score is computed for each entity pair using a decision tree.

While the original system is processing search log data, the modified version we used for the experiments presented in this paper uses a dump of the common crawl data[9] as input and we replaced calls to the distributed key-value store with dummy methods which always return 1. Even though

[9]http://commoncrawl.org

these changes invalidate the actual rankings that the system computes, they do not change the workload characteristics of the original system.

B. Synthetic Topologies

To gain insight into how well our proposed optimization strategy generalizes to other topology designs, we generated a series of synthetic Storm topologies and evaluated the performance gains with each of them. To this end, we used the widely used graph generator *GGen* [24] to generate three topologies. We then modified these graphs by assigning different values for time and resource complexities to each vertex of the graph.

Processing pipelines in Storm typically consist of several tasks, some of which can run independently in parallel, while others need to wait for input data from upstream nodes. For this reason we generated "layer-by-layer" graphs, as motivated in [25]. In layer-by-layer graphs, nodes are grouped in layers. Nodes in the same layer only have links to nodes of downstream layers, but no links to nodes of the same layer. As we want to test each graph over the course of 60 or more sampling runs, each run taking two to ten minutes, while varying node attributes of the graph such as necessary processing time or the use of constrained resources, we could only afford a small number of base graphs/topologies. To ascertain typical topology sizes we reviewed the literature (see Table III): we found that most currently published topologies have fewer than 60 vertices, whilst enterprise-grade application may have up to 100 components [26]. Hence, we generated topologies of three different sizes having 10, 50, and 100 vertices.

To get valid SPS and comparable graphs, we ensured that (1) all vertices of the graph are connected to at least one other vertex in the graph and that (2) the average out-degree across the whole graph is approximately constant in all the produced graphs. Since GGen allows choosing (i) the number of vertices in the graph, (ii) the number of layers in the graph, and (iii) the probability of vertex to connect to vertices of different downstream layers only, we picked parameters that would fulfill these constraints as listed in Table II. The table reports on the configuration parameters the number of vertices, layers, and probabilities to connect to vertices of the next layer as well as the typical graph statistics such as the number of edges, spout vertices (or sources), the number of bolt with an outdegree of zero (sinks), and the average outdegree of all vertices in the topology.

In the basic configuration, all operators in the topologies were configured to use the same amount of computational resources and time. As real world topologies may not be balanced, we introduced a number of ways to create imbalance. We describe these modifications in the following paragraphs. Each modification will be motivated and its application described in detail. With all of them the goal is the same: we intend to generate multiple modified graphs from a base graph, which we can then optimize using Bayesian Optimization.

Name	V	E	L	P	Src	Snk	AOD
Small	10	17	4	0.40	3	3	1.70
Medium	50	88	5	0.08	17	17	1.76
Large	100	170	10	0.04	29	27	1.65

Table II: The number of (V)ertices, (E)dges, and (L)ayers, the (P)robability to connect to vertices of different layers, the number of sources (Src) and sinks (Snk), as well as the average out-degree (AOD) of the vertices in the generated topologies.

Year	Description	# of Ops
2003	Data Dissemination Problem in [27]	40
2004	Linear Road Benchmark in [28]	60
2013	Linear Road Benchmark used in [29]	7
2013	DEBS'13 Grand Challenge Query[30]	3

Table III: Number of operators of topologies in literature.

1) Time Complexity: Each tuple takes n units of compute resources (CPU cycles) to process. The amount of compute resources each tuple requires to be processed depends, naturally, on the task the processor has to achieve. We set a target value of 20 compute resource units per tuple in our experiments. As we need to simulate actual processing, we implemented a busy wait strategy in which we empirically set the complexity of the operations, such that 1 compute resource unit corresponds to about 1ms of execution time. Hence, the processing of one data tuple takes about 20ms on a system that is not overloaded. Others have reported values of up to 60ms [11] per tuple. In addition to the balanced base configuration we also generated imbalanced ones, where the required compute resource units vary across the topology. Specifically, we used a uniform distribution of compute length with a mean of 20 compute units (between 0 and 40), resulting in an average processing time of 20 in the whole topology.

2) Resource Complexity: Bolts (or vertices) that are only constrained by CPU time are embarrassingly parallelizable and can be optimized soley by increasing their degree of parallelism. Other bolts may be constrained by resources that cannot be added by increasing their parallelism. If a task instance is slow because of a globally contentious resource, for example a central database, instantiating more tasks will not help improve the throughput and only waste resources on context switching. To simulate contentious resources, we flag a certain percentage of the processing time as being "resource contentious". This means that the time complexity of the respective bolts is multiplied with the total number of task instances for a given bolt to negate the effect of increasing parallelism for the affected bolt. To avoid unfair distribution of resource contention, this percentage is based on the number of total compute resource units (see above in section IV-B1), rather than just selecting a percentage of

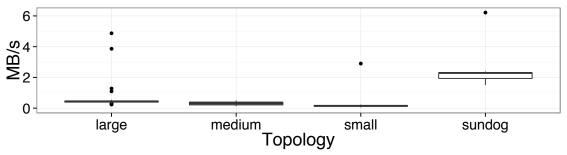


Figure 3. Average network load in MB/s per worker for each topology.

the bolts. For example, if we have a topology with 10 nodes which have an average time complexity of 20 and we want to have 25% contentious nodes, we select nodes with a total time complexity of 50 units of compute resources and flag them as ‘contentious resources bolts’.

3) Selectivity: For every incoming tuple, a task instance produces 0 to n outgoing tuples. This is called the *selectivity* value of a bolt. Selectivity is not susceptible to the degree of parallelism. In contrast to processing time, the selectivity value not only influences the workload on downstream operators, it also incurs network traffic. However, in setups where the network is not the bottleneck, selectivity can be simulated using the value for time complexity: having a selectivity of more than 1 incurs increased workload on all downstream bolts in the topology. Hence, to simulate a higher selectivity value, we can as well increase the time value of all downstream nodes. Analogously, a selectivity value of less than 1 reduces the workload of all downstream nodes. For the experiments presented in this paper, we took care not to overload the network by using sufficiently large processing time values and omitted a special selectivity flag. Figure 3 shows the network utilization in megabytes per second (MB/s) as an average across all worker nodes in the cluster for all four types of topologies we used in our evaluations. Note that the network was not saturated in any of our experiments, as the cluster nodes are equipped with gigabit network cards that allow a theoretical upper limit of 128MB/s.

4) Topology Generation: The topology generation for the synthetic topologies consists of (i) generating the base graphs using GGen, (ii) modifying the resulting graphs by randomly (but uniformly) changing the time complexity values and resource contention flags, and finally, (iii) generating Storm topologies. The bolts in these topologies are linked using shuffle-grouping, meaning tuples are evenly shuffled among downstream bolts. This completes the description of the topology modifications. The concrete degree to which we applied these modifications will be described below in the section V.

C. Cluster Configuration

This section describes the cluster hardware and software used for the experiments.

1) Hardware: Many compute clusters that are in production in industry consist of several thousand commodity computers [31]. While we did not have a cluster of this magnitude at our disposal, we made an effort to simulate such a cluster by connecting the work station computers that our department offers to our students to work on, into an 80 machine Hadoop cluster. The student computers are iMac computers with Intel Core i5 CPUs (4 cores with each 2.7GHz), 8GB ram, and 250GB SSD hard drives. The iMacs are distributed over two rooms, in rows of at most 8 computers (some rows contain fewer computers). Each row is connected using a 1Gbps switches. All rows are connected over at most 2 Cisco Catalyst 4510R+E (48Gbps) switches. We scheduled our evaluations during off hours. However, we cannot exclude that there were students using the iMacs systems during the evaluations. We compensated for this by running each evaluation multiple times.

In version 0.23, Hadoop introduced support for other applications than MapReduce through its YARN[10] resource scheduler. For our experiments, we used the Storm-Yarn project,[11] which is an effort to run Storm inside a Hadoop cluster. In order to prevent the Hadoop cluster from going down because of a student accidentally shutting down his work station, we ran the Hadoop Job Tracker as well as the Zookeeper[12] instance on a separate machine. For this, we used a virtual machine with 4 simulated 2.6GHz CPUs.[13]

2) Software: All iMac computers ran OS X 10.9.5, having Java 1.8.0_11.jdk installed. The virtual machine running the job tracker ran on Debian 7.8 (wheezy). We used Hadoop 2.2.0 as the base system and Storm 0.9.2-incubating through Storm-Yarn 1.0-alpha orchestrated by Zookeeper 3.4.5.

V. Results

This section discusses the results of our evaluations. First, we compare throughput performance achieved when tuning parallelization. Second, we explore the practicality in terms of convergence speed of using Bayesian Optimization. Next, we investigate the robustness of our approach against problem size. We close with a discussion of the tuning of additional parameters.

A. Configuring Parallelism

In a first set of experiments, we were interested in finding out if the parallelism hints of a topology can effectively be chosen using Bayesian Optimization. We used Spearmint to choose a parallelism hint for each node in the topology and decide over the maximum number of task instances (“max-tasks”) that Storm should instantiate. To ensure that the sum of tasks is smaller than max-tasks, we normalized the chosen hints using the max-task parameter. As a baseline we implemented a naive parallel-linear ascent (pla) optimizer, which sets the same parallelism hint on all spout/bolt nodes in the topology and increases them in parallel. We set the maximum number of evaluation runs

[10]http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site
[11]https://github.com/yahoo/storm-yarn
[12]http://zookeeper.apache.org
[13]We use KVM as the virtualization plattform with storage on iSCSI.

to be 60. To prevent unnecessary evaluation runs for the pla strategies, we stopped the optimizer after measuring zero performance in three consecutive runs. As we possess detailed topological information for the synthetic topologies, we additionally created a set of experiments in which we leveraged the topological information. For these experiments we recursively calculated a "base parallelism weight" value for each node in the topology. For bolts, this base weight is equal to the sum of the weights of all their parent nodes. All spout nodes have a base weight of 1. The optimizer then only had to choose a multiplier for these base-parallelism weights. We denote optimizers working with this additional topological information with the letter "i" for "informed".

Figure 4 serves as an overview over the results from this comparison. We list results achieved using the bayesian optimizer (bo, we will discuss the bo180 values below), the parallel linear ascent optimizer (pla), the informed bayesian optimzier (ibo), and the informed parallel linear ascent optimizer (ipla). For each optimization step, we had the cluster process data for two minutes. Starting and stopping the topology took between 40 and 100 seconds. The duration of the optimization steps depends on the size of the topology and took between 13 and 518 seconds (see Section V-C below). We then ran the best configuration for each topology-optimizer combination 30 times. Given that our approach is probabilistic, we repeated the procedure and graphed the better of the two optimization passes in the figure, which shows the average of the 30 repetitions with the best configuration (error bars represent the minimum/maximum values).

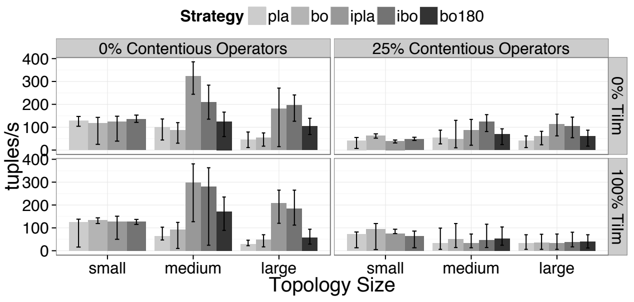


Figure 4. *Throughput*: Average performance running synthetic topologies with varying time complexity imbalance and resource contention on an 80 node cluster (TiIm = time complexity imbalance, pla = parallel linear ascent, bo/bo180 = Bayesian Optimization, ipla = informed parallel linear ascent, ibo = informed Bayesian Optimization).

The top-left bar plot shows the results for topoloies for which the variance of time complexity is zero. In these homogenous topologies, each spout and bolt consumes the same number of CPU cycles to process a single incoming tuple. Also, we ignore resource contention. Under such conditions, setting all parallelism hints to the same value and increasing them in parallel is a prudent optimization strategy. *ipla* dominates the field for medium and large topologies. The bayesian optimization strategies (*bo* and *ibo*) are unable to find a better configurations. For small topologies, all optimization strategies arrive at equally good solutions.

The lower-left bar plot in Figure 4 shows the results for the case in which the required CPU cycles to process tuples varies for each bolt. We observe that having topological information is still of use, however, Bayesian Optimization can partially compensate for the absence of such information (pla vs. bo) for medium and large topologies. For small topologies, all strategies arrive at equally good parallelism configurations.

In the upper-right plot of Figure 4 we experimented with the case in which temporal complexity is zero (e.g. homogenous bolts), however, we randomly selected 25% of the compute time to be dependent on "contentious resources" (see section IV-B2). Essentially we bottlenecked 25% of all bolts. This experiment shows that topological information is still of value in such cases, however, Bayesian Optimization can help increase performance substantially for medium and large topologies.

In the lower-right corner of Figure 4, we finally tested the case in which we have both, heterogeneous time complexity, as well as 25% bottlenecked bolts. As we can see, topological information does not allow for any better configurations. In fact, for the large topologies all optimizers set values of or very close to 1 for all nodes the topology. The small topology configuration with time complexity imbalance and contentious resources, Bayesian Optimization without topological knowledge arrived at the best throughput results.

B. Convergence Speed

To assess the convergence speed we plotted the step at which we first measured the best performance for each experiment (Figure 5). As we ran each optimizer twice, we show minimum-maximum-average numbers over the two runs. Naturally, the bayesian optimizer needs many more steps than the linear parallel approach. Interestingly, having topological information, not only improved the overall result of the configuration, but also shortened the number of evaluation runs necessary, to arrive at this result. In four cases, the best configuration was only found in the 60st run. For this reason, we ran four configurations for 120 more steps. The best result achieved in 180 steps is depicted in 4 as the *bo180* strategy. We observe that giving the bayesian optimizer more time to find good parallelism settings, yields better results in all cases. In Figure 6, we plotted the LOESS regression smoothing with span 0.75 for these experiments. The trendlines are consistent with the performance values in Figure 4: for the small and the medium topologies, good parallelim settings can be found within the first 50 and 100 optimization steps, respectively. For the large topologies, for which over 100 parameters need to be set, the setting with time imbalance (lower-left) seems to have benefitted most

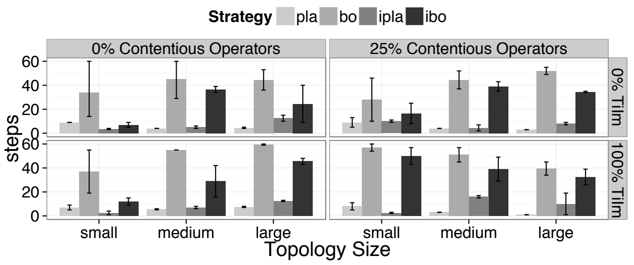


Figure 5. *Convergence Speed*: Number of steps required to arrive at the maximum performance in terms of throughput for each experiment (TiIm = time complexity imbalance, pla = parallel linear ascent, bo = bayesian Optimization, ipla = informed parallel linear ascent, ibo = informed Bayesian Optimization).

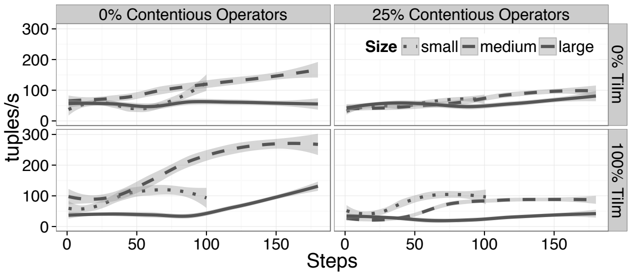


Figure 6. Loess regression smoothing of the optimization steps of the bayesian optimizer setting parallelism hints.

from the additional time and the trend line increases after 100 time steps.

C. Scalability

To assess the suitability of our approach for large parameter spaces, we measured the average optimizer run-time and plotted it in Figure 7. The pla and ipla times are barely visible, they lie all between 0 and 1 second. As we can see, the time required to choose the next configuration increases dramatically as we increase the topology size, and hence, the number of parameters to optimize. Spearmint needed an average of 35, 90, and 173 seconds for each optimization step for the small, medium, and large topologies (bo runs). Recalling that these topologies have 10, 50, and 100 bolts, and hence, parallelism hints to optimize, we note that this increase is sublinear. The informed Bayesian Optimizer (ibo) required slightly more time with 36, 168, and 253 seconds per step, respectively. We assume this is due to the fact that we used floating points values for the weights as opposed to the simple integer values. These numbers increase also sublinearly. We observe increasing spreads between best and worst-case durations. However, even these numbers increase only sublinearly. Hence, all results indicate that the use of our approach is practical in terms of run-time.

To summarize these evaluations with synthetic data, we conclude that while Bayesian Optimization can be practically used to configure the parallelism of a distributed stream processor, it can only partially compensate for missing topological information. In situations, however, where this

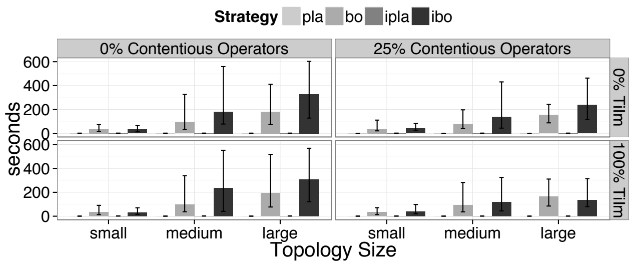


Figure 7. *Scalability*: Average time elapsed between runs in seconds as a measure for how long one optimization step takes (TiIm = time complexity imbalance, pla = parallel linear ascent, bo = Bayesian Optimization, ipla = informed parallel linear ascent, ibo = informed Bayesian Optimization).

information is expensive to obtain or topologies are complex (e.g., due to joins or filters), Bayesian Optimization offers itself as a viable tool.

D. Optimizing Other Configuration Parameters

To assess the usefulness of Bayesian Optimization for configuring other aspects of a distributed stream processor for a real-world topology in combination with the degrees of parallelism of its operator nodes, we conducted an additional set of experiments, which we present in the following sections.

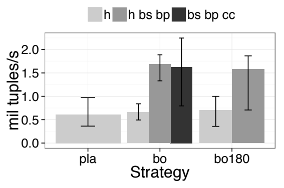
In these experiments, we used the Sundog topology presented in section IV-A. As we did not have topological information about the topology readily available (and they are non-trivial to derive), we only employed the parallel linear ascent (pla) and the bayesian optimizer (bo). We ran three different combinations of parameter sets: in a first set or experiments, we had the optimizers choose the parallelism hints as in our previous evaluations to get a baseline to compare to. Then, we created configurations for Spearmint to optimize parameter sets that include the parallelism hints along with the batch parallelism, batch size, and finally a set of concurrency related configuration parameters: the batch-size parameter lets us set the number of lines of text that Sundog ingests in one mini-batch. Batches can be processed in parallel. The parameter for batch-parallelism defines how many such batches can be in the processing pipeline concurrently. The last set of parameters that we included in the setup were all concurrency (cc) related parameters from Table I: the number of worker and receiver threads, as well as the number of "acker" tasks that Storm should instantiate.

We present the results obtained from running these experiments in Figure 8. The best configuration of each optimizer was run 30 times. We present average throughput values in million tuples per second, denoting the maximum and minimum measured results with error bars.

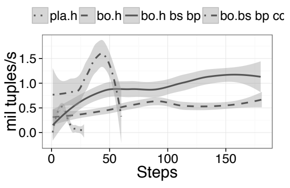
In a first comparison, and to get a baseline for later experiments, we looked at the performance that can be achieved by setting parallelism hint (h) values. For these experiments, we used a batch-size of 50.000 lines and a batch-parallelism of 5, as these were the values used when Sundog was developed and manually tuned. As our cluster machines have 4 cores, we set a worker thread pool to 8. We did not set a value for the number of acker instances, so the default of one per worker host was used: 80 total in our case. We left the default value of 1 for the worker receiver thread count. Looking at the results in Figure 8a, we note that all three approaches (pla, bo, and bo180) achieve very similar average results (611k, 660k, and 699k tuples/s). A two-sided t-test deemed these differences statistically insignificant (p=0.05).

In a second set of experiments, we added the parameters for batch-parllelism (bp) and batch-size (bs) and had Spearmint choose values for these settings in addition to the parallelism hints resulting in substantial performance gains. We measured a throughput of 1.68 million tuples per second. This amounts to an improvement of 2.8x compared to the 611k tuples/second throughput measured when only optimizing the parallelism hints using pla. When looking at the parameter configurations we found that the bayesian optimizer changed the batch-parallelism from 5 to 16 and increased batch-size from 50.000 to 265.312 tuples. The Sundog developers reported that they never set these values that high, as the time it takes to process a batch of this size seemed unreasonably high.

In a last experiment, we explored if not spending the time on optimizing parallelism, but instead on fully concentrating on other parameters, would yield better performance. In this experiment, we fixed the parallelism hint for all bolts to the best value that the pla strategy yielded (11), and had Spearmint search the parameter space of all parameters listed in Table I except the parallelism hints. The result can be seen in Figure 8a (bs bp cc): even though the bayesian optimizer could spend 60 optimization steps on this reduced parameter space, the highest throughput measured in this experiment is comparable to the one achieved in the "h bs bp" cases. Indeed, two-sided t-tests revealed that the throughput of the "bs bp cc" run (1.63mio tuples/s) was not significantly different from the performance measured when searching the extended parameter space over 60 (1.68mio tuples/s) or 180 (1.58mio tuples/s) steps (p=0.05). Figure 8b shows the progress of the approaches: concentrating on only optimizing parallelism did not result in good performance even after 180 steps (dashed line). Configuring parallelism as well as batch size and batch parallelism (solid line), did yield good results, eventually. The fastest way seems to be a combination of both approaches, where we first configured parallelism using the parallel-linear approach and enhanced the settings by optimizing the batch-size, batch-parallelism, as well as the number of threads used by the various subsystems (dot-dashed line).



(a) Throughput



(b) Convergence

Figure 8. Throughput and convergence speed for Sundog using parallel linear ascent (pla) and Bayesian Optimization (bo) to optimize the parallelism hints (h) with and without the batch size (bs) and the batch parallelism (bp), as well as a set of concurrency (cc) related parameters.

VI. Conclusions and Future Work

We presented and evaluated an approach for the configuration of distributed stream processors. We implemented the approach using a set of synthetic and real-world Storm topologies. We had a bayesian optimization framework find optimal parameter settings to achieve high throughput and compared against a parallel linear optimization approach. Our results suggest that our approach is viable and can find parameter configurations that lead to substantial throughput improvements by a factor of up to 2.8 in the best case.

There are some limitations to our work. First, Bayesian Optimization using Gaussian Processes assumes that the objective function is continuous. This may not always be the case when configuring the parallelism of a distributed stream processor. To what extent this negatively influenced the results in our auto-parallelization experiments is subject to future work. Second, as even small sample differences influence the decision process of the bayesian optimizer, our setup could be improved by running each sampling run multiple times and by using the average performance for each tested parameter configuration.

We believe that Bayesian Optimization is a viable tool for the field of distributed computing. Especially for tuning systems with a large configuration parameter space in which the impact of every single parameter cannot easily be predicted. As such, we are convinced that our work is of interest to the community.

Acknowledgments

We would like to thank Hanspeter Kunz and Enrico Solca for the many hours they spent helping us setup and run the cluster we used for the evaluations.

References

[1] W. Zheng, R. Bianchini, and T. D. Nguyen, "Automatic configuration of internet services," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 1, 2007.

[2] D. Kossmann, "The state of the art in distributed query processing," *ACM Computing Surveys*, vol. 32, no. 4, 2000.

[3] A. Deshpande, Z. Ives, and V. Raman, "Adaptive query processing," *Foundations and Trends in*, vol. 1, no. 1, 2006.

[4] M. Cammert and J. Kramer, "A cost-based approach to adaptive resource management in data stream systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 20, no. 2, 2008.

[5] M. Daum, F. Lauterwald, P. Baumgärtel, N. Pollner, and K. Meyer-Wegener, "Black-box determination of cost models' parameters for federated stream-processing systems," *Proceedings of the Symposium on International Database Engineering and Applications*, 2011.

[6] T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer, "Latency-aware elastic scaling for distributed data stream processing systems thomas," in *DEBS 2014*, 2014.

[7] J. Gomes and H. A. Choi, "Cost-based solution for optimizing multi-join queries over distributed streaming sensor data," *Int. Conference on Collaborative Computing: Networking, Applications and Worksharing, CollaborateCom*, 2006.

[8] S. Schmidt, "Quality-of-service-aware data stream processing," Ph.D. dissertation, 2007.

[9] A. Saboori, G. J. G. Jiang, and H. C. H. Chen, "Autotuning configurations in distributed systems for performance improvements using evolutionary strategies," *The 28th International Conference on Distributed Computing Systems*, 2008.

[10] S. Schneider, M. Hirzel, B. Gedik, and K.-L. Wu, "Auto-parallelizing stateful distributed streaming applications," in *PACT*. New York, New York, USA: ACM Press, 2012.

[11] S. Wu, V. Kumar, K.-L. Wu, and B. C. Ooi, "Parallelizing stateful operators in a distributed stream processing system: How, should you and how much?" *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, 2012.

[12] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K. L. Wu, "Elastic scaling of data parallel operators in stream processing," *Proceedings of the 2009 IEEE International Parallel and Distributed Processing Symposium*, 2009.

[13] B. Gedik, S. Schneider, M. Hirzel, and K. L. Wu, "Elastic scaling for data stream processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, 2014.

[14] J. Mockus, "On bayesian methods for seeking the extremum and their application." in *IFIP Congress*, 1977.

[15] J. Bergstra, D. Yamins, and D. Cox, "Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures," *Proceedings of the 30th International Conference on Machine Learning*, 2013.

[16] C. Thornton, F. Hutter, H. H. Hoos, K. Leyton-Brown, and K. L.-B. Chris Thornton, Frank Hutter, Holger H. Hoos, "Auto-weka: Combined selection and hyperparameter optimization of classification algorithms," *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2013.

[17] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," *NIPS*, 2012.

[18] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *Learning and Intelligent Optimization*. Springer, 2011.

[19] B. Gedik, H. Andrade, and K.-L. Wu, "A code generation approach to optimizing high-performance distributed data stream processing," *Proceeding of the 18th ACM conference on Information and knowledge management - CIKM '09*, 2009.

[20] E. Brochu, V. M. Cora, and N. de Freitas, "A tutorial on bayesian optimization of expensive cost functions , with application to active user modeling and hierarchical reinforcement learning," University of British Columbia, Department of Computer Science, Tech. Rep., 2010.

[21] J. Mockus, V. Tiesis, and A. Zilinskas, "The application of bayesian methods for seeking the extremum," *Towards Global Optimization*, vol. 2, no. 117-129, 1978.

[22] K. Eggensperger, M. Feurer, F. Hutter, J. Bergstra, and J. Snoek, "Towards an empirical foundation for assessing bayesian optimization of hyperparameters," in *Advances in Neural Information Processing Systems Workshop on Bayesian Optimization in Theory and Practice2*, 2013.

[23] L. Fischer, R. Blanco, P. Mika, and A. Bernstein, "Timely Semantics: A Study of a Stream-based Ranking System for Entity Relationships," in *Procesings of the 14th International Semantic Web Conference (ISWC)*, 2015.

[24] D. Cordeiro, P. Swann, D. Trystram, J.-m. Vincent, G. Mounié, S. Perarnau, and F. Wagner, "Random graph generation for scheduling simulations," in *SIMUTools*, 2010.

[25] T. Tobita and H. Kasahara, "A standard task graph set for fair evaluation of multiprocessor scheduling algorithms," *Journal of Scheduling*, vol. 5, no. 5, sep 2002.

[26] M. Hajjat, X. Sun, Y.-w. E. Sung, D. Maltz, and S. Rao, "Cloudward bound: Planning for beneficial migration of enterprise applications to the cloud," in *SIGCOMM*, 2010.

[27] D. Abadi, D. Carney, U. U. G. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: a new model and architecture for data stream management," *The VLDB Journal The International Journal on Very Large Data Bases*, vol. 12, no. 2, 2003.

[28] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts, "Linear road : A stream data management benchmark," in *VLDB*, 2004.

[29] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in *SIGMOD*. New York, New York, USA: ACM Press, 2013.

[30] L. Aniello, R. Baldoni, and L. Querzoni, "Adaptive online scheduling in storm," in *International conference on Distributed event-based systems - DEBS '13*. New York, New York, USA: ACM Press, 2013.

[31] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *SIGCOMM*. New York, New York, USA: ACM Press, 2008.