University of Zurich[UZH]

Online Optimization of Job Parallelization in Apache GearPump

Master Thesis February 25, 2019

Te Tan
of Hubei, China

Student-ID: 16-725-962
te.tan@uzh.ch

Advisor: **Pengchen Duan, Dr. Daniele Dell'Aglio**
Prof. Abraham Bernstein, PhD
Institut für Informatik
Universität Zürich
http://www.ifi.uzh.ch/ddis

Acknowledgements

Firstly I would like to thank my supervisors Pengchen Duan and Dr. Daniele Dell'Aglio for their patient guidance and insightful advice. They encouraged me to overcome difficulties and helped me solve problems.

Secondly I would like to thank Prof. Abraham Bernstein and Dynamic and Distributed Information Systems (DDIS) group of University of Zurich. They provided me with resources to do experiments.

Lastly, I would like to say thank you to my family and friends, who gave me spiritual support and motivation during the thesis.

Zusammenfassung

Die Parametereinstellung im Bereich verteilter (Streaming) Systeme ist ein beliebtes Forschungsgebiet. Viele Lösungen wurden von der Forschungsgemeinschaft vorgeschlagen. Das Bayesian Optimization (BO) ist eine solche Lösung, die sich als mächtig erwiesen hat. Während die bestehende Methode zur Durchführung des BO-Prozesses ist 'offline', die Herunterfahren des Systems sowie viele ineffiziente manuelle Schritte braucht. In dieser Arbeit implementieren wir einen Optimierer, der BO-Optimierung 'online' durchführen kann. Der Optimierer wird in Apache Gearpump, einer nachrichtengesteuerten Streaming-Engine, implementiert. Da der DAG-Vorgang zur Laufzeit die Voraussetzung für die Online-Optimierung ist, untersuchen wir die vorhandene Funktion von Apache Gearpump und schlagen einen verbesserten Ansatz namens Restart vor, um Laufzeit-DAG-Vorgänge durchzuführen. Unterstützt durch den Restart-Ansatz entwerfen und implementieren wir JobOptimizer, der eine Online-BO-Optimierung ermöglicht. Die Bewertungsergebnisse zeigen: mit der Beschränkung der maximalen Anzahl von Versuchen, obwohl JobOptimizer den Parameterraum nicht ausreichend erforschen kann, es kann einen besseren Parametersatz finden als zufällige Exploration. Bei vergleichsweise größeren DAG-Anwendungen übertrifft es den Durchsatz von Linear Ascent Optimizer.

Abstract

Parameter tuning in the realm of distributed (streaming) systems is a popular research area and many solutions have been proposed by the research community. Bayesian Optimization (BO) is one of the them which is proved to be powerful. While the existing way to conduct the BO process is 'offline' and involves shutting down the system as well as many inefficient manual steps, in this work we implement an optimizer which is able to do 'online' BO optimization. The optimizer is implemented within Apache Gearpump, a message-driven streaming engine. As the DAG operation at runtime is the prerequisite for doing 'online' optimization, we inspect into the existing feature of Apache Gearpump, and propose our improved approach named Restart to do runtime DAG operations. Then supported by Restart approach, we design and implement JobOptimizer, which enables 'online' BO optimization. The evaluation results show that: with the constraint of maximum number of trials, although JobOptimizer is not able to explore the parameter space adequately, it is able to find better parameter set than random exploration. It also outperforms Linear Ascent Optimizer in terms of throughput in the case of comparatively larger DAG applications.

Table of Contents

1

Introduction

1.1 Big Data Systems and Streaming processing

Powered by the adoption of advanced information technologies such as Cloud Computing and IoT devices, the discussion of Big Data in both academic communities and the industry has experienced an exponentially increase over years. The term 'Big Data' was first used by Cox and Ellsworth (1997) to refer to the datasets which can not fit in memory or on local disk especially in the context of visualization. The easy-to-understand definition of Big Data is the digital analysis of datasets to extract insights, correlations and causations, and value from data. Other definitions are also given by various research groups, such as the '*3Vs*' definition (Volume, Variety and Velocity) proposed by Laney (2001), and the '*5Vs*' definition ('*3Vs*' + Validity and Value) proposed by Marr (2015).

As indicated in the book written by Maxim et al. (2017), one of the key areas of the Big Data framework evolution, is to continuously increase the performance to meet the demand of sub-second and real-time data analytics. This is achieved by developing and improving streaming data processing engine.

Akidau (2015) clarified the term 'streaming' data to be unbounded data which is ever-growing and infinite, as opposed to 'batch' data which is bounded. Processing bounded data is quite straightforward, which only requires fetching data from a distributed file system, running it through a batch processing engine such as MapReduce, and then writing the results back to the file system. Batch engines, although not explicitly designed for unbounded data, have been used to process unbounded data due to the comparatively high maturity of batch engines. The most common way to deal with an unbounded dataset with a batch engine is slicing the input data into fixed-sized windows, then processing each of those windows as a bounded dataset. However, in many use cases batch engine is not able to meet requirements of low latency and high throughput. Furthermore, batch engine is usually not able to process of unordered data with varying event-time skew. Streaming engines are specifically designed for reasoning about time so it can exceed batch system when processing unbounded data.

Given the advantage of processing unbounded data with low latency, streaming systems are a good choice compared to batch systems when confronting following scenarios:

- Time series data flow: stream processing naturally fit with time series data (traffic sensors data, transaction logs, social media data etc.) and can detect patterns over time.

- Approximate results are sufficient: not a complete dataset is needed to produce an accurate result, streaming processing is able to produce an approximate result at lower cost.

- Data is too huge to store: sometimes the data is too huge to fit in the current storage resources, streaming processing can handle it by doing transformation and only keeping the useful information.

Furthermore, 'streaming' is also a natural model to think about and program. In countless use cases the data are perfect streaming flow and streaming systems are the most suitable choice to handle it. Popular use cases are production line monitoring, supply chain optimization, fraud detection, traffic monitoring, sports live analytics and so on.

1.2 Parallelization in Streaming Applications

Processing massive data is traditionally time-consuming. Distributed parallel computing is the preferable way to improve the efficiency of massive data processing. Data parallelism is the notion that data is distributed among nodes and therefore can be processed in parallel. Taking the MapReduce paradigm for example, it consists of two distinct phases: Map and Reduce. Map tasks are required to be written in such a way that they can operate independently on a single chunk of stored data. Although this strict limitation could sometimes be challenging for programmers, the benefit it brings is that the tasks can be executed in a highly parallel way.

Before we explain and formulate the problem we deal with, we introduce the concept of parallelism with the example of Apache Gearpump [1], which is the target streaming system we inspect in the thesis. We will discuss Apache Gearpump in more detail in Chapter 3.

The programming model for a Gearpump application is a Directed Acyclic Graph (DAG). Users must represent their computation job as a DAG, in which a vertex stands for a computing task and an edge represents the message path. A vertex in DAG is called *processor* in Gearpump. There are three kinds of processors within a DAG of Gearpump: sources, nodes and sinks. A source is a processor without input streams, whose responsibility is emitting data to downstream processors. Sources are typically used to connect a Gearpump DAG to external data sources such as web-services and file systems. A sink is a processor without downstream processors, mainly responsible for gathering computing results. Processors other than sources and sinks are normal nodes, with both upstream and downstream processors. An example DAG is shown in Figure 1.1 left, which is a simple topology with only one source, one normal node and one sink.

The topology shown in Figure 1.1 left is only a logic representation of a DAG. The programmer will determine how many instances (parallelism) of each processor will be

[1] Apache Gearpump(incubating): Overview, https://gearpump.apache.org/overview.html

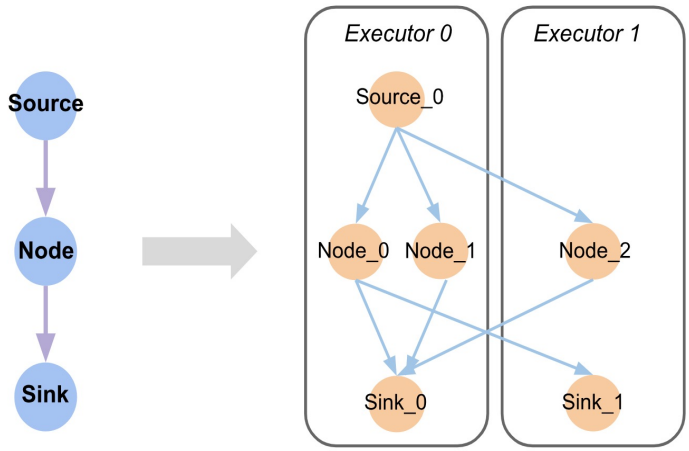


Figure 1.1: Logical and runtime representation of an example DAG in Apache Gearpump

created during runtime. Figure 1.1 right shows one possible topology of the DAG represented on the left in runtime. Two executors (each represents one JVM process) are executing this application. The source, node and sink have 1, 3 and 2 runtime instances respectively, which correspond to the degree of parallelization of one, three and two for each processor.

Streaming systems provide a number of configuration parameters that allow the programmer as well as administrators to configure the system and application. One of them is the parallelism of each DAG processor which could significantly affect the application performance. Unfortunately, it is extremely hard for programmers to determine a 'good' parameter settings of parallelism in advance. Hence parameter tuning is proved to be of great help when deciding the parameters of streaming system and application.

1.3 Motivation and Research Questions

Parameter tuning in Big Data systems is a popular research area and many approaches are suggested by research groups. We will discuss these approaches in Chapter 2.

One of these promising approaches is the application of Bayesian Optimization (BO) (Mockus, 1974). BO approach typically treats the system to be tuned as a black-box function, and iteratively evaluates the objective function based on prior knowledge. We will introduce more details about BO in Section 5.1. BO has successfully been applied to the configuration of Apache Storm [2] by Fischer et al. (2015).

However, the existing way to conduct the BO process is complex and involves many manual steps. This approach is what we call as '*offline*', which means we have to do a sequence of iterations like this: running the application with one parameter set, gathering

[2] Apache Storm, http://storm.apache.org/

information, shutting down the system and then start a new iteration with next parameter set. In each iteration we have to shut down the whole system, adjusting parameter settings (manually or with the help of program) and launch the application again. The situation becomes worse when it comes to tuning the parallelism of application. Streaming application is highly customized and we cannot figure out in advance how the DAG looks like or how many processors it contains. Therefore the system configuration file usually does not provide config-options for the parallelism of an application. In other words, we need to change the parallelism of each processor by ourselves (often manually) before each optimization iteration begins. This predicament prevents the application of BO to real use case because it is extremely clumsy and inefficient.

Our initial ideal is to investigate how we can improve the optimization process of BO to make it efficient and suitable for real use case. While in the context of streaming processing, there are many parameters to be configured, we narrow down the scope and focus on the degree of parallelism of streaming application, which greatly affects the application performance. If we can find a better way than existing work to configure parallelism, it can be generalized to other parameter settings. Instead of conducting the process 'offline', we want to do it '*online*' and automatically, without shutting down the whole system and manually adjusting the parallelism of DAG.

As a comparatively new streaming engine under incubation, Apache Gearpump came into our sights as we found that it offers a feature named 'Dynamic DAG' which allows us to change the DAG of application at runtime. If 'Dynamic DAG' can help us do runtime DAG operations needed for optimization, we can make the optimization process 'online'. If 'Dynamic DAG' is not competent enough to support the optimization process, we intend to improve it. Specifically we want to implement an optimizer (named JobOptimizer) for Gearpump, which will conduct the Bayesian Optimization process automatically and without need for shutting down the system or application. Our work is guided by following research questions:

- *RQ1*: Can 'Dynamic DAG' of Gearpump be used for runtime DAG operations required by the optimization process? If not, how can we improve it?
- *RQ2*: How to design and implement an optimizer which conducts Bayesian Optimization at runtime ('online')?
- *RQ3*: How does the implemented optimizer perform compared to baselines?

The rest of this thesis is organized like this: we will give a short introduction to the related work in Chapter 2. Then we will introduce the streaming engine used in our experiment: Apache Gearpump in Chapter 3. In Chapter 4 we will inspect into the capability of the feature 'Dynamic DAG', and how we improve it to meet up our needs for runtime DAG operations. After that we will elaborate in Chapter 5 how we design and develop JobOptimizer, who enables 'online' Bayesian Optimization of parallelism based on runtime DAG operations. The evaluation and analysis of JobOptimizer will be shown in Section 5.3. Lastly we will briefly clarify the limitations and future work in Chapter 6 and Chapter 7 respectively, and summarize our work in Chapter 8.

2

Related Works

Distributed systems specialized in big data processing do not have a long history. Dating back to 2003 and 2004, Google introduced *Google File System (GFS)* and MapReduce paradigm which became a milestone of the development of big data systems. *GFS* is a distributed and fault-tolerant file system whose design goal is achieving scalable, reliable and robust massive data storage (Ghemawat et al., 2003). MapReduce is both a powerful programming paradigm and a distributed data processing engine, which allows users to easily develop and run data parallel applications on a large unreliable cluster (Dean and Ghemawat, 2004).

Apache Hadoop is an open-source project first developed by a group of researchers at Yahoo and then managed by Apache Software Foundation [1]. The two main components of Hadoop are *Hadoop Distributed File System (HDFS)*, which is designed to replicate the functionality of GFS, and its own implementation of MapReduce runtime (Shvachko et al., 2010). Various research teams have then enriched the Hadoop ecosystem by developing many other systems such as HBase, Cassandra, Storm and Flink.

Grew from UC Berkeley, Apache Spark has gained heavy attention within the big data realm as a new generation of data processing framework. It was designed to mitigate some of the disadvantages of MapReduce, specifically when it comes to using the same dataset for iterative computing. Spark has been optimized for tasks that require the reuse of the same dataset across multiple parallel computations such as machine learning and data analytic tasks. Thanks to *Resilient Distributed Dataset (RDD)*, the abstraction for distributed data access in memory, it can achieve higher precessing speed compared to MapReduce tasks (Zaharia et al., 2010).

2.1 Parameter Configuration of Big Data Systems

The problem of parameter configuration in the field of distributed (streaming) systems has been studied over years. A number of cost-based models were proposed to estimate the potential outcomes and describe the complexity of streaming systems (Cammert et al., 2008; Daum et al., 2011; Heinze et al., 2014). Then the parameter settings of a

[1] Apache Hadoop, http://hadoop.apache.org/

system can be decided based on the predicted outcomes of an appropriate model. Take the work of Heinze et al. (2014) for instance. The elastic data stream processing engine allocates dynamically new resources with the help of scaling policies depending on the current query load. This resource allocation involves the movement and restoring of processors, which creates a latency spike. For the purpose of minimizing the latency violations according to the Service-Level Agreement (SLA) in cloud computing services, Heinze et al. (2014) introduced a model to estimate the cost of processor movements in terms of end to end latency. The scaling decision and configuration will then be determined according to both the utilization of resources, and the cost of doing certain processor movement.

The foundation of the cost-based model is that we assume the relationship between the parameters we care about, and the outcome can be described by a closed-form model. When dealing with problems with prior knowledge, and there is a comparatively simple relationship between parameters and the outcome, cost-model would be powerful. However, in our case of parameter tuning, we do not know if a mathematical model exists and well fits the system, so we treat the application as a black box.

Besides the cost model approach, another kind of approach is what Babu (2010) named as 'late-binding' named. While the cost model-based approach relies on predicted performance of an operation before the operation being performed, 'late-binding' delays the setting of parameters until the operation being conducted and some results being observed. Since we are going to do 'online' optimization by observing the execution of the Gearpump application, our approach belongs to 'late-binding' category.

As an example of 'late-binding' approach, a self-tuning system was proposed by Wu and Gokhale (2013) in order to help data scientists who are lack of expertise to tun the Hadoop system. The proposed system, named Profiling and Performance Analysis-based System (PPABS), is based on machine learning. After collecting the characteristics of previous applications such as memory, CPU usage, they group these historical applications into clusters using a modified k-means clustering algorithm. Then they search optimal configuration settings for each center of these trained clusters and label each center with the optimal setting found. To look for the best setting of a newly-submitted application, in the first step PPABS lets the application run, observing and gathering its characteristics. In the second step, PPABS classifies this application into one of the cluster, and loads the tuned configuration which is the optimal setting of the center of this cluster. The entire process is done automatically after user submitting the application.

Similar to our interest, there are also studies with emphasis on extracting the appropriate degree of parallelization (Schneider et al., 2012; Gordon et al., 2006; Schneider et al., 2009). Schneider et al. (2009) enabled the elasticity of operators on IBM's System S (a streaming processing middleware) which can adjust the parallelism of operators in response to the dynamic streaming workload. Afterwards the same team Schneider et al. (2012) was dedicated to developing a compiler to determine the safe parallel subgraphs of an application DAG by doing code analysis. Based on that runtime scalability is enabled and the parallelization is determined according to the runtime workload. Faced with constantly-changing streaming workload, they launch the application with default or random degree of parallelization, without paying attention to whether this initial

settings generate the best performance. Then they will adjust the parameters during runtime. In contrast to focusing on dynamical change of input workload and adjusting the parallelism, our research treats the input workload as static and tries to find the best initial degree of parallelization.

Our work differentiates from both the work of Wu and Gokhale (2013) and Schneider et al. (2009) mentioned above, in the sense that we treat the application as a black box. Wu and Gokhale (2013) cluster applications according to the application characteristics, and Schneider et al. (2009) analyse the application code. However, we do not look into details of the submitted application but only observe its performance given a parameter setting.

2.2 Bayesian Optimization and its Application

Bayesian Optimization (BO) (Mockus, 1974) has successfully been applied to configuration problems where no mathematical closed-form cost model is known in advance. One of the most popular area of its application in computer science is hyperparameter tuning in machine learning algorithms (Bergstra et al., 2013; Snoek et al., 2012).

Snoek et al. (2012) empirically analyse a few machine learning algorithms and compare Bayesian Optimization to existing strategies on a number of challenging problems. In the experiments, they develop a few modified versions of Expected Improvement (EI) as acquisition functions, namely GP EI MCMC, GP EI Opt and GP EI per second. For example, they compare these algorithms to Tree Parzen Algorithm (TPA) on a logistic regression classification task on the popular MNIST data. The task is choosing four hyperparameters, the learning rate for stochastic gradient descent, on a log scale from 0 to 1, the l2 regularization parameter, between 0 and 1, the mini batch size, from 20 to 2000 and the number of learning epochs, from 5 to 2000. The result shows that GP EI significantly outperforms TPA, finding the target minimum in less than half as many evaluations. Another experiment is about Convolutional Neural Networks. Multi-layer neural networks require careful tuning of numerous hyperparameters, which are usually computationally expensive. In their work, they tune nine hyperparameters of a three-layer convolutional network on the CIFAR-10 benchmark dataset. This model has been carefully tuned by a human expert to achieve a highly competitive result of 18% test error. The parameters include the number of epochs to run the model, the learning rate, four weight costs (one for each layer and the softmax output weights), and the width, scale and power of the response normalization on the pooling layers of the network. The best hyperparameters found by GP EI MCMC achieve a test error of 14:98%, which is over 3% better than the expert.

The hyperparameter tuning of machine learning algorithms shares two similarities with the configuration of Big Data systems. Firstly, different from mathematical black-box function optimizations, the function evaluation of machine learning problems require a variable amount of time. For example, training a big neural network with many hidden units could take days or even months. The function evaluation of Big Data systems in-

volves running the system, observation and analysis of results, so it may also take a long time. Secondly, machine learning experiments often run in parallel on multiple cores or multiple machines in a cluster. The same goes in running Big Data systems.

2.3 Applying Bayesian Optimization to Big Data Systems Configuration

Despite the successful application of Bayesian Optimization in hyperparameter tuning of machine learning algorithms, the research on applying Bayesian Optimization to the configuration of Big Data system has rarely been reported before 2015. Fischer et al. (2015) published a paper about applying BO to the configuration of streaming systems. They try to find the best config-settings of Apache Storm including number of threads per worker, number of acker tasks, number of batches being processed in parallel, number of tuples in each batch, and the parallelism of operators. A real-world application and three synthetic topologies are used to evaluate the usefulness of BO for configuration. Compared to the baseline which is a naive linear-ascent optimizer, within a predefined number of steps, the BO algorithm can find a better parameter hints which generates much higher throughput. BO algorithm can also effectively find a better solution than linear-ascent optimizer within a fixed number of steps when configuring other parameters such as batch size. However, the convergence speed of BO is worse than linear-ascent optimizer since it will explore the parameter space adequately before convergence. Similarly, Jamshidi and Casale (2016) implement a automate-tuning tool named Bayesian Optimization for Configuration Optimization (BO4CO) to do configuration tuning in the context of Apache Storm. They go further than Fischer et al. (2015) in a sense that they not only tun the configuration of the system, but also update the hyperparameter of Gaussian Process kernel and the prior mean function during the optimization process.

Fischer et al. (2015) run BO with a fixed number of iterations and select the best performance among those iterations. The convergence speed they measure is the number of iteration in which the best performance occurs at the first time. In other words, this approach does not try to find the globally optimal parameter set, and the convergence speed does not measure how long it takes to find this globally optimal value. In our approach, we try to find out the globally optimal value by defining convergence criteria. Furthermore, while Fischer et al. (2015) try to optimize the parameter hints specified in the configuration file, our work is application-oriented instead of system-oriented, which means we do not need to specify any parameters in the system configuration file before the application actually runs. Both Fischer et al. (2015) and Jamshidi and Casale (2016) conduct the optimization process in a way which we call as the '*offline*' approach. 'Offline' means we have to do a sequence of iterations like this: running the application with one parameter set, gathering information, shutting down the system and then start a new iteration with next parameter set. In each iteration we have to shut down the whole system, adjusting parameter settings (manually or automatically) and launch the

application again. This inefficiency motivates us to look for an optimization solution which is '*online*' and automatically, without shutting down the system and manually adjusting the parameters.

3

Apache Gearpump: A Promising Streaming Engine Built on Akka

Apache Gearpump is a big data streaming engine built on Akka [1]. It has become an Apache incubator project since March 2016. Gearpump engine is event/message based. Powered by Akka, Gearpump provides extremely high throughput and low latency. From benchmarks it is able to process 18 million messages per second with a 8ms latency on a 4-node cluster[2]. To understand how Gearpump works, first we need to know its cornerstone: Akka. We will at first introduce Akka Actor Model and then discuss the features of Gearpump.

3.1 Akka Actor Model

Akka is a set of open-source libraries for designing scalable, resilient systems. According to its documentation, Akka provides multi-threaded behavior without the use of low-level concurrency constructs like atomics or locks, transparent remote communication between systems, and scalable high-availability architecture for building distributed systems.

The underline key of Akka's power is its Actor model, which provides an abstraction that makes it easier to write concurrent parallel programs. Today the challenges of building distributed systems cannot be fully solved with the traditional Object-Oriented Programming (OOP) model. Three crucial problems are identified when we try to program using the traditional model in a modern multi-threaded computing architecture:

- *The challenge to encapsulation*: the encapsulation (protection of invariants) can only be ensured in the case of single-thread. Multi-thread execution often leads to corrupted internal state. The synchronization lock is an inefficient solution because it leads to suspending of threads and risk of deadlocks.

- *The illusion of shared memory*: CPU cores are indeed writing to their respective local cache instead of directly to the shared memory. Hence sharing local changes between threads leads to the shipment of cached data which is very costly. This

[1] Akka, https://akka.io/

[2] Performance Evaluation, https://gearpump.apache.org/releases/latest/introduction/performance-report/index.html

data shipment has been made transparent through marking the variables as atomic on JVM. As it is difficult for programmers to use the costly denotion 'atomic' carefully, it is more reasonable to passing necessary messages explicitly between threads.

- *The problem of work delegation*: To achieve any meaningful concurrency on current systems, threads must delegate tasks among each other. After the main thread (who delegates a work to another) delegates a work to the worker thread, how can it be notified even the worker thread died. If the task fails with an exception, where does the exception propagate to? and how to recover the failed task (note that the task state is lost due to the exception reaching to the top, unwinding all of the call stack).

To handle these problems brought by the traditional OOP model, the Actor model is proposed. An actor is a role who can send messages, receive messages, update its state and change its behavior according to received message. Next we discuss the two distinct properties in order to explain how Akka tries to solve the problems mentioned above.

Message passing instead of method calling

In OOP model, the enforcement of invariants occurs on the same thread. If multiple threads try to access the same instance by method invocation, the encapsulation of invariants will be violated. Instructions of the two invocations can be interleaved in arbitrary ways, hence the intactness of invariants cannot be guaranteed.

In Actor model, actors communicate with each other by sending messages. Sending a message does not release the control of executing thread. The actor who wants to delegate a task to another will send a message and continue without blocking. The receiver actor reacts to messages at its convenience and return execution when it finish processing the current message. If results are expected, the receiver actor will reply with another message. Specifically what happens when an actor receives a message is:

1. The receiver actor adds the message to the end of its mailbox (a queue),
2. If the receiver actor is not scheduled for execution, it is marked as ready to execute,
3. A scheduler takes the actor and starts executing it by picking (exactly) one message from the head of the mailbox,
4. The actor updates its state and sends messages to other actors.
5. The actor is unscheduled.

The application is driven forward by each actor sending, processing and receiving messages. In this way, actors can finish more work in the same amount of time, and the whole system can process as many messages simultaneously as the hardware can support. Encapsulation is protected as any time at most one message will be processed per actor, and different actors work independently.

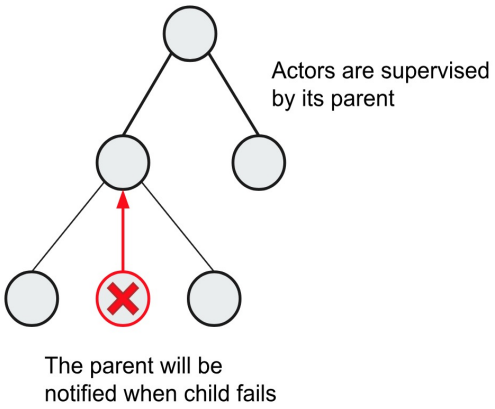


Figure 3.1: Actors are organized by a hierarchical supervision tree

Hierarchical supervision of actors

Actors are organized as a hierarchical tree structure (as shown in Figure 3.1). Each actor is created and supervised by its parent actor (the root actor created by user is supervised by system daemon actor). When an actor fails, its parent actor will be notified and it can react to the failure. If the parent actor fails or is stopped on purpose, all of its children will be recursively stopped. No actor dies silently. This hierarchical supervision ensures that the error or exception is handled properly:

- If the delegated task on the receiver actor fails due to task itself, the error or exception information will be sent as ordinary messages to the sender actor. The sender actor then processes the message and react.
- If the receiver actor dies, its parent actor will be notified. Then the parent actor can decide to restart its child actors on certain types of failures or stop them completely. Restarts are kept invisible from the outside.

3.2 Introduction to Apache Gearpump

Inspired by advances brought by Akka framework, a group of engineers in Intel developed Gearpump, a real-time big data streaming engine. Benefiting from the message-driven architecture and actor model, Gearpump achieved very high performance with high throughput and low latency. In this section we talk about its actor architecture and a few feature highlights.

3.2.1 Gearpump Actor Architecture

An actor is the smallest functional unit in Gearpump system (so called 'actor everywhere'). The simplified hierarchical structure of Gearpump actors is shown in Figure

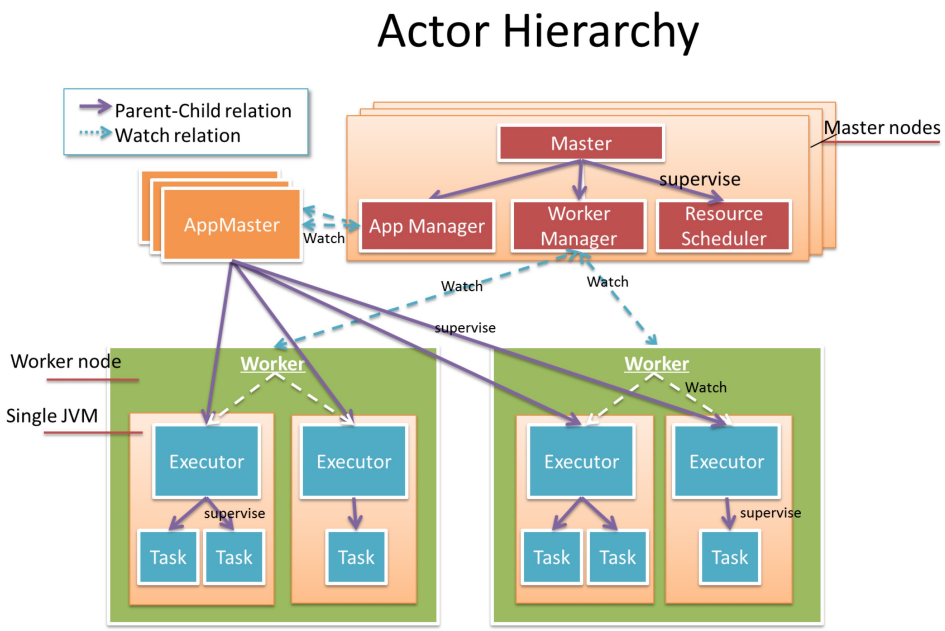


Figure 3.2: The simplified hierarchy of Gearpump actors (Zhong, 2014)

3.2. Note that the full actor hierarchy is much more complicated than the one shown here, with many more functional actor roles such as executor manager, DAG manager, Metrics services. There are two categories of actors: the cluster actors and the application ones. The cluster actors are Master and Worker. The Master is responsible for managing cluster resources and all applications. The Worker represents a a physical (virtual) machine who does the actual work as well as managing its own resources and keeping reporting to the Master. The Master creates a few supportive children actors(App Manager, Worker Manager as in Figure 3.2) among whom the responsibility is split.

Application actors mainly have following roles:

- AppMaster: Each streaming application running on Gearpump has a one-to-one responsible AppMaster actor, which manages the state of the application and schedule tasks to workers.
- Executor: represents a JVM process and is the child of AppMaster. It will manage the lifecycle of tasks and responsible for their recovery when failure occurs.
- Task: represents a thread, does the real job and is the child of Executor.

With the well-split duty among actors, the application submission process is simple. As shown in Figure 3.3, when a new application is submitted to the Master, it will allocate a worker with sufficient resources and launch an AppMaster for the specific application. The AppMaster calculates resources required for the application and

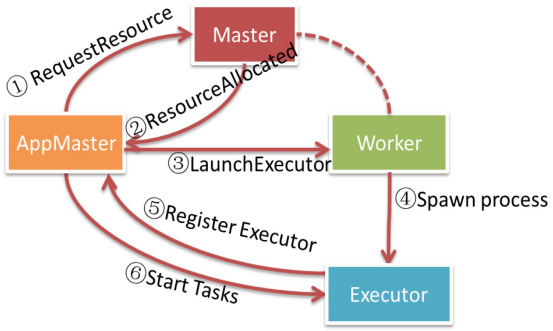


Figure 3.3: The application submission processs in Gearpump (Zhong, 2014)

sends a request to the Master. After allocating resources, the AppMaster will notify assigned workers to launch executors. Successful registration of launched executors will be followed by starting tasks.

3.2.2 Technical Highlights of Gearpump

Here we briefly present a few attractive features of Apache Gearpump, which can help us better understand the Akka actor model and Gearpump.

Safe work delegation and fault folerance

With the help of Akka's hierarchical supervision tree, Gearpump has been built as a resilient system. Firstly, the problem of work delegation mentioned in Section 3.1 has been solved. Children actors who have been assigned with tasks from parents are liable to report the status of the executing tasks especially when errors or exceptions occur. The parent will decide whether and where to restart the failed tasks in case of failure.

Secondly, fault tolerance is achieved. The parent keeps watching on its children. If a child is down due to internal errors, it will request new resource to restart it. Also, the crashing of an actor will automatically lead to shutting-down of all its children, which is efficient and transparent to users. For instance, each Gearpump application has its own AppMaster. If an AppMaster crashes, Master will schedule new resources to restart the AppMaster, important states of the application such as Global Minimum Clock(see below) will be restored.

But what if Master itself crashes? The high availability of Master is insured by an Akka cluster. The cluster consists of several Master nodes (typically three), who will maintain and update a replicated Master state locally. In case of failure one standby Master node will be activated and take over the responsibility. All workers are communicating with a Master proxy instead of a real Master node, so the Master recovery is transparent to workers.

Efficient message passing

In streaming use case, the typical message size is usually less than 100 bytes per message, which makes the efficiency of the network (bandwidth usage) very low. Another problem is the message overhead is too big as each message sent between two actors contains the complete long actor path.

To make as much as use of the bandwidth, the Gearpump engineers group and send multiple messages together. Furthermore, they implemented a transportation layer named *Netty* whose job is to translate the actor path. Before the message is sent, Netty translates the complete actor path into the task id. When receiving a message, Netty translates the actor path back from task id. Only task ids will be transported on the wire. Hence the efficiency of message passing has been greatly improved.

At-Least-Once and Exactly-Once message delivery

Trustworthy message delivery is a prerequisite for a streaming engine. Gearpump provides configurable message delivery options, by which users can choose from At-Least-Once message delivery mode or Exactly-Once mode. While At-Least-Once mode is insured internally within Gearpump, Exactly-once mode needs support of resources to do trustworthy backup of checkpoints.

At-Least-Once delivery is achieved by Global Minimum Clock(GMC) mechanism. Global Clock Service tracks the minimum timestamp of all pending messages in the system. Tasks update their own minimum clock during processing messages, and report to Global Clock Service. The value of GMC indicates that all messages with a timestamp smaller than the GMC have been successfully processed. When message loss occurs, the GMC will not move ahead. The sender will be notified to send the lost messages again. The GMC value is stored and updated in Master cluster. If the application crashes, it will be restarted and initialized with the latest GMC value. Then the source will replay message from the reloaded GMC timestamp. Exacly-Once delivery is achieved by continuously taking snapshot of the application states and storing them into a database. When the application is restored from checkpoints, the states will also be restored among tasks so repeat messages can be detected.

3.2.3 A few hints on our implementation

Our target is to implement a module to enable 'online' optimization of Gearpump applications. The above technical characteristics about Gearpump gives us some hints about how our module should be implemented.

Firstly, the module should be an actor or a set of actors, who dive itself or themselves into the Gearpump actor architecture. That means the implemented actor should be created by one of the application actors. Then the parent actor will take over the management of its lifecycle automatically without our concern. Secondly, the programming diagram of the message-driven actor model should be followed, in which there is no

method invocation chain between actors. The communication among our implemented actors, as well as between our actors and Gearpump actors should be achieved through message passing. Thirdly, thanks to the supervision tree, we can focus on our business logic, without worrying about the failure recovery and resource reclaim. However, the important actor states should be stored to the Master and regularly updated. In this way can we ensure the recovered actor is able to work from break points.

The feature most relevant to our research questions, is 'Dynamic DAG' which means we can do runtime DAG operations without shutting down the system or application. Next we will inspect into the source code of Gearpump and check if it is capable enough to support our goal: achieving 'online' optimization of parallelization.

4

Runtime DAG Operation in Apache Gearpump

To do automatic and 'online' optimization of parallelization, we decide to implement a module named *JobOptimizer*. We will discuss the details of its implementation in Chapter 5. The JobOptimizer's job is to observe the application performance, based on which it makes decisions about parallelisms and conduct runtime DAG modifications. It iteratively repeats this process until some stop-conditions are met, ideally it stops when the optimal parallelization of the application is found. In this chapter we discover the solution for our JobOptimizer about how to conduct runtime DAG operations.

One attractive feature provided by Gearpump which is to our interest, is Dynamic DAG. Backed by the knowledge of Akka Actor model we discussed above, we will look into it and try to answer *RQ1*.

4.1 Dynamic DAG provided by Gearpump

The functionality of Dynamic DAG allows users go to the web user interface (UI) of Gearpump after starting the web server, checking the status of the submitted application and doing operations on the graphic DAG. An example screenshot of the graphic web UI is shown in Figure 4.1. Possible DAG operations include modifying processor parallelism and changing the jar file of a subgraph of DAG. Note that the functionalities of adding or deleting processors are not yet supported as they stated in the provisioning documents.

After inspecting into the source code, we found that the Dynamic DAG feature is realized by enabling runtime replacement of processors. The implementation involves many actors and complex message communications. We extracted from the source code the process of replacing a processor and drew the message passing flow as shown in Figure 4.2. AppMasterService receives HTTP request sent by user from web UI and forward it to AppMaster. AppMaster then sends a ReplaceProcessor message to DagManager, which is a child actor of AppMaster and responsible for most of the DAG operations. Afterwards TaskManager communicates with JarScheduler, send the new version of DAG and fetch the resource requirement from JarScheduler. Then the resource allocation process begins, starting a new executor for the modified processor, and lastly tasks which belong to the modified processor is launched.

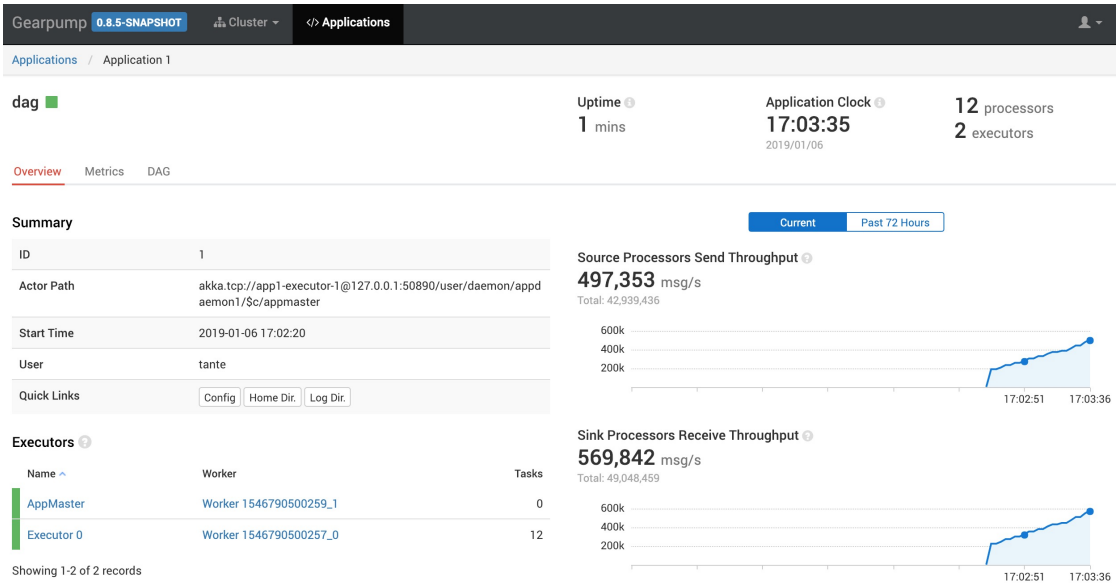


Figure 4.1: An example of the web UI provided by Gearpump

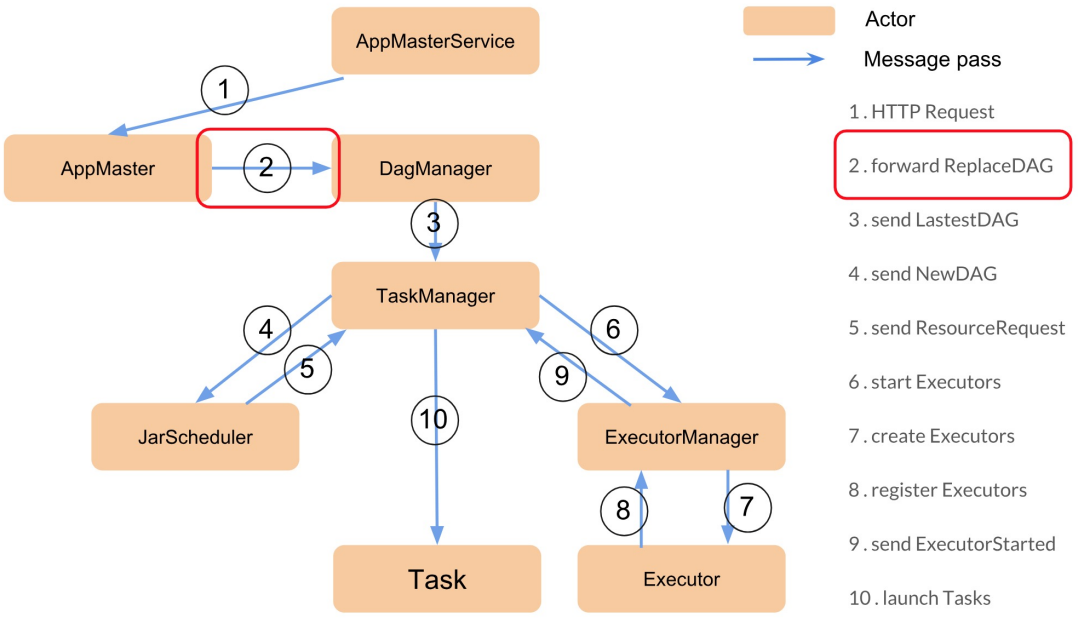


Figure 4.2: The message passing flow of replacing a processor

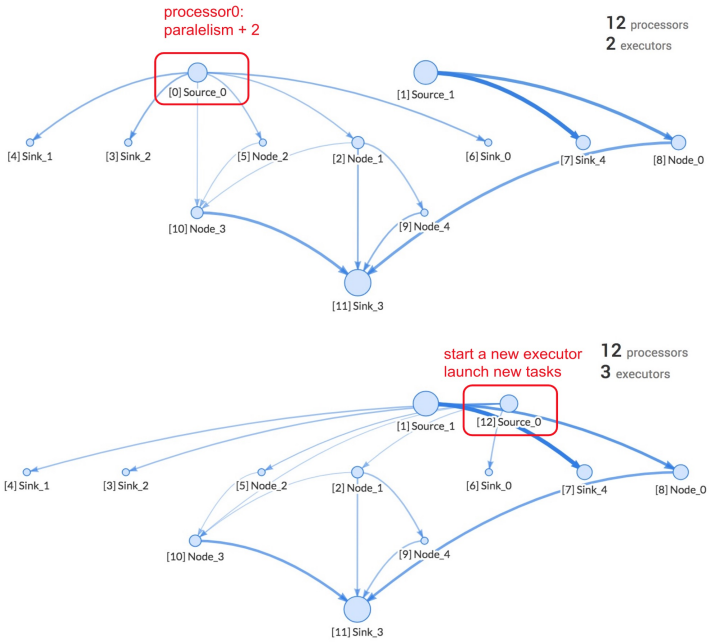


Figure 4.3: An example of DAG processor replacement process enabled by Gearpump

To illustrate and visualize how it works we have run an example Gearpump application provided by Gearpump. The DAG topology of this application can be seen in Figure 4.3. These two pictures are screenshots of the Gearpump web UI dashboard. Processors are denoted by circles with its name(for example *source_0*) and its unique id within the square brackets. Arrows represent the message pass channel, the thicker an arrow line is, the higher message passing rate of this channel. There are 2 sources, 5 intermediate processors and 5 sinks in this DAG. All processors have an initial parallelism of 1. To observe how Gearpump change the parallelism of processors, here we want to increase the parallelism of *source_0* by 2, as shown in Figure 4.3 above. The system will start a new executor(now there are 3 executors in contrast to 2 previously, Figure 4.3 below), and launch a new processor(id 12) with the modified parallelism of 3. These 3 new tasks are new instances of *source_0*.

Be careful that a new executor(corresponds to a JVM process) is started to launch the changed processor no matter how small the change is. This lets other tasks unaffected and continue to work. The benefit is obvious: modifying one processor does not disturb others work. Figure 4.4 shows throughput of the example application. We did the replacement operation at *10:45:55*. As we replaced *source_0*, an observable small decrease of source send throughput happened at *10:45:55*. But the other source processor, *source_1* has not been influenced, so the decrease is limited, not sharply going down to 0.

However, there are two big problems with this solution of DAG operation. Firstly each time only one processor of the DAG is allowed to be modified. That means if we want to

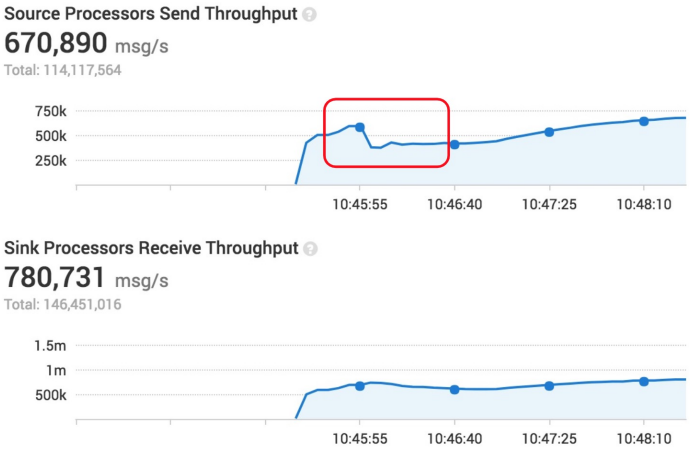


Figure 4.4: The source send throughput and sink receive throughput of the example application

modify parallelisms of multiple processors, we have to do it sequentially by changing one processor at each time. Secondly, starting a new executor which involves starting a new process is resource-consuming, especially when we want to do multiple changes required by the optimization. Imagine that we have to change parallelisms of 5 processors, we should repeat the replacement process 5 times and it ends up with 5 more running JVM processes. To summarize this section as well as answering the first sub-question of *RQ1*, we concluded that this Dynamic DAG feature provided by Gearpump is not sufficient for us to conduct runtime DAG operations required by optimization due to following drawbacks:

1. Resource consuming: each time we replace a processor, it starts a new executor.

2. Only support replacing a single processor each time, not suitable for doing experiments involving multiple changes.

3. A few bugs: during the our exploration we found a few bugs, two of them are relevant to DAG operations. We solved them with great efforts. In order to keep the explanation of this article smooth, we will briefly mention these bugs in Appendix.

4.2 How can we do runtime DAG operations better?

Besides the 'online' requirement, the new solution should also overcome the drawbacks of the existing Dynamic DAG functionality. We propose and implement an approach named Restart, which is the answer to the second sub-question of *RQ1*. Firstly, the proposed substitution of original Dynamic DAG should be more resource-efficient than

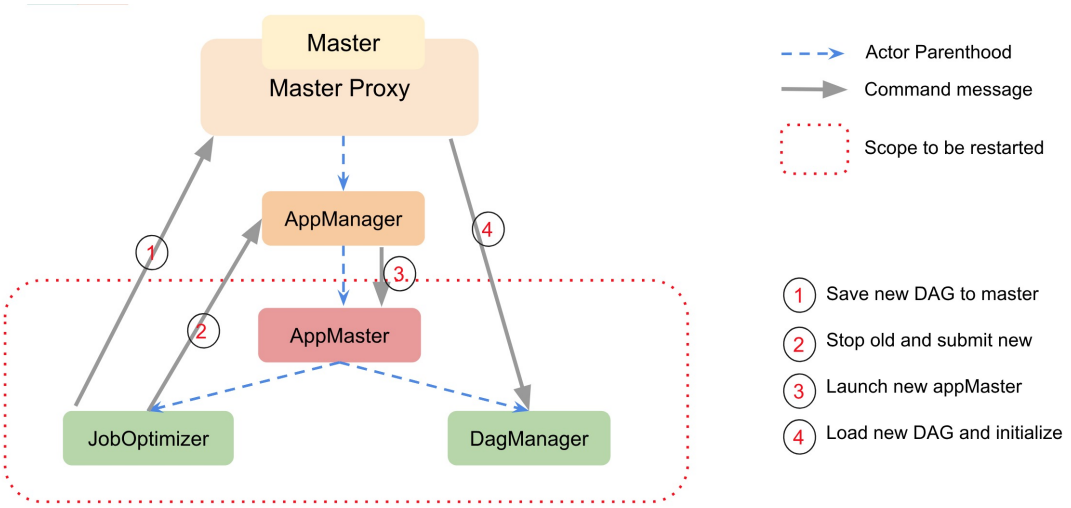


Figure 4.5: The mechanism of Restart approach

existing one. Hence restarting a new executor for each modified processor is not preferable. Secondly, the proposed solution should be able to do an operation which involves modification of multiple processors. Generally speaking, Restart is stopping the application and recovering it with the new DAG and old application states. By stopping the old application and deploying a new one, we reclaim resources and allocate again which avoids launching new executors. Furthermore, we enable the recovery of arbitrarily-modified DAG when initializing the application by storing it to Master. Next we are going to interpret Restart approach, and evaluate its performance with empirical studies.

4.2.1 Restart approach

Figure 4.5 illuminates the mechanism of Restart approach. In the hierarchical supervision tree, JobOptimizer is created by AppMaster, just as DAGManager and other application-specific functional actors. The dash-blue arrows represent the actor parenthood relationship of these actors. In this way, the lifecycle of JobOptimizer is managed by AppMaster. In the implementation of actor system, if an actor wants to send a message to another actor, it should hold an actor reference instance of the targeted actor (named ActorRef). ActorRef is similar to an address pointer which indicates the location of an object. AppMaster keeps an overview of the running application and contains all ActorRef instances of actors relevant to this application. As a child of AppMaster, it is convenient for JobOptimizer to communicate with other actors such as Master Proxy.

The DAG operation enabled by Restart approach involves 4 steps. In the first step, JobOptimizer saves application states to Master. As all actors related to the application will be stopped and restart (see step 2), the intermediate states will lose during this process. So important states must be stored to Master. Master provides an in-memory storage service, by which applications can store key-value pairs. The key has two levels:

application level and attribute level. When the application submits a key-value pair to Master, Master determines the id of this application as the first level key, then stores the key-value pair into a map structure. Therefore the stored data from different applications does not affect each other, and is safe because of the master high availability.

Specifically, JopOptimizer saves following states of the running application as key-value pairs to Master cluster:

1. the new DAG: in-memory representation of the modified DAG. The new version of DAG is constructed by JobOptimizer according to the parallelism which is decided by an optimization algorithm.

2. transition start time: mark the timestamp of the beginning of transition. The transition process is from the time when the new version of DAG is constructed, to all tasks of this new DAG are started.

3. current global minimum clock (GMC): after the new DAG starts to run, the application continues to process messages from GMC. Only messages with timestamp larger than GMC will be processed.

4. optimization states: intermediate data of the optimization process, is discussed in Chapter 5.

In step 2, the subtree with AppMaster as root is stopped. JobOptimzier sends a message to AppManager who is the father of AppMaster. AppManager then stops the old AppMaster and launch a new one. Note that the stopping process of the old AppMaster is very efficient and require no extra human intervention thanks to the hierarchical supervision structure of Akka Actor model. All children actors of AppMaster will be recursively stopped after AppMaster is stopped. Right after that AppManager requests resources from Master, and immediately start a new AppMaster (step 3). During the start process of AppMaster, it fetches GMC from Master and create functional children including JobOptimizer and DAGManager. Then JobOptimizer fetches transition start time and optimization states mentioned above. DAGManager initializes itself with the new DAG got from Master (step 4). Resources are allocated by Master according to the new version of DAG. Now all the application actors run on the new DAG. The transition phase ends when all tasks are started.

We can see that the scope of actors to be restarted is small (the red dash square in Figure 4.5), involving AppMaster and its children, no influence on system-level actors or other running applications. Furthermore, it is apparently an 'online' approach because we construct and save the DAG in-memory representation, without shutting down the system and modifying any code.

We have to point out that this approach is specifically designed for DAG application with stateless processors. Since it stops all processors completely, all intermediate states of processors lose. We will deal with how to save and recover states of each processor in future work.

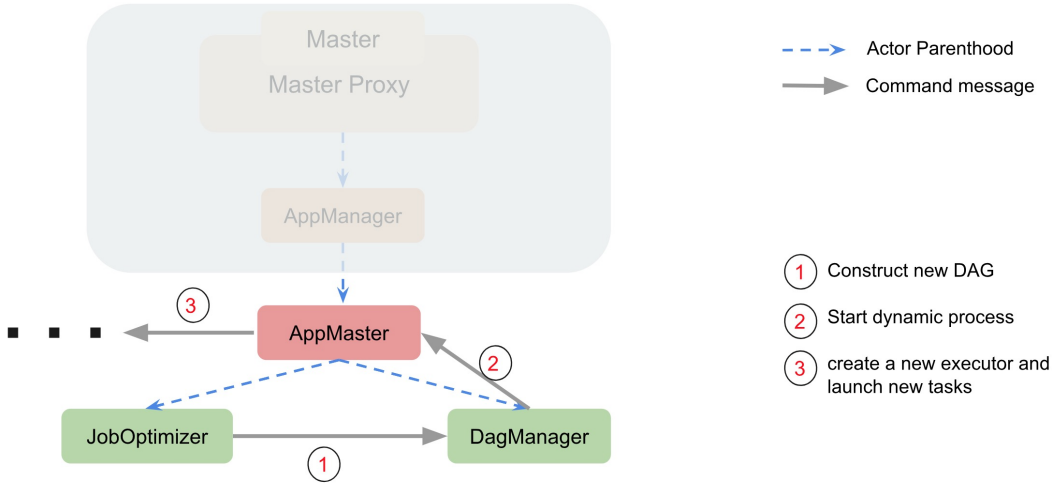


Figure 4.6: The mechanism of Dynamic DAG Update approach

4.2.2 Dynamic Update approach

As a comparison of our Restart approach, Dynamic Update is implemented as an enhanced version of the existing feature 'Dynamic DAG'. It starts a new executor, but launching all changed processors on the new executor instead of a single one.

The mechanism of Dynamic DAG Update is explained in Figure 4.6. The grey area which consists of Master, AppManager and their children will not be involved. JobOptimizer constructs the new version of DAG, sending it to DagManager. Then similarly to the existing feature, DagManager begins the dynamic process by starting a new executor and launching tasks. The difference from the original feature is that now we can make multiple changes and a new DAG with all these changes is deployed. With this approach, there is no need to stop the AppMaster or any other unaffected working tasks.

We give a brief comparison of pros and cons between these two ideas in Table 4.1. But to know which one is more preferable during runtime, we have to implement them and do experiments.

Table 4.1: A comparison between concepts of Restart and Dynamic Update approach

	pros	cons
Restart	straightforward and simple, resource efficient	processors without changes will also be stopped and recovered
Dynamic Update	processors without changes can still work, donot restart the AppMaster hierarchy	resource-consuming(start extra executors), complex details during implementation

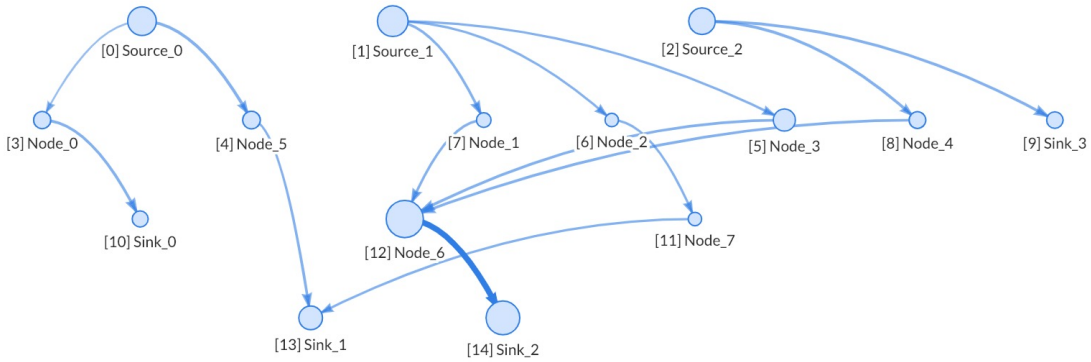


Figure 4.7: Topology of the randomly-generated DAG for experiment

4.2.3 Empirical comparison of two approaches

To evaluate our Restart approach against Dynamic Update approach, we conducted a set of experiments with two different DAG topologies. The first one (Topology 1) is the same application as shown in Figure 4.3 which is an example application provided by Gearpump. The initial degrees of parallelization of all processors are 1.

The second topology (Topology 2) is randomly-generated with 15 processors as shown in Figure 4.7. The topology is produced from a random DAG creation package named 'random-dag' available in npm [1]. It is easy to install and use. For DAG creation with random-dag, we can specify 5 parameters: *max_per_rank* (how 'fat' the DAG should be), *min_per_rank*, *max_ranks* (how 'tall' the DAG should be), *min_ranks*, *probability* (chance of having an edge). We generate the experiment topology with *max_per_rank* of 8, *min_per_rank* of 2, *max_ranks* of 4, *min_ranks* of 3 and *probability* of 0.1. All processors except sources and sinks output exactly one message to downstream processors after receiving one message from upstream processors. Note that both topologies are composed of stateless processors.

The goal is to compare the transition time and sink receive-throughput of Restart and Dynamic Update approach. The transition time is defined as the time duration from the start of the DAG operation, to the time that every task including the changed ones begin to work normally. Sink receive-throughput is the summation of throughput of all sink processors. We do experiments both at local environment and in the cluster. Local machine is a MacOS system with a CPU of 2.7 GHz Intel Core i5, and memory of 8G 1867 MHz. The cluster consists of two Amazon EC2 instances. The Master node is a Linux system with 2 virtual CPUs and 8G memory and the worker node is an Ubuntu system with also 2 virtual CPUs and 16G memory.

[1] random-dag package, https://www.npmjs.com/package/random-dag

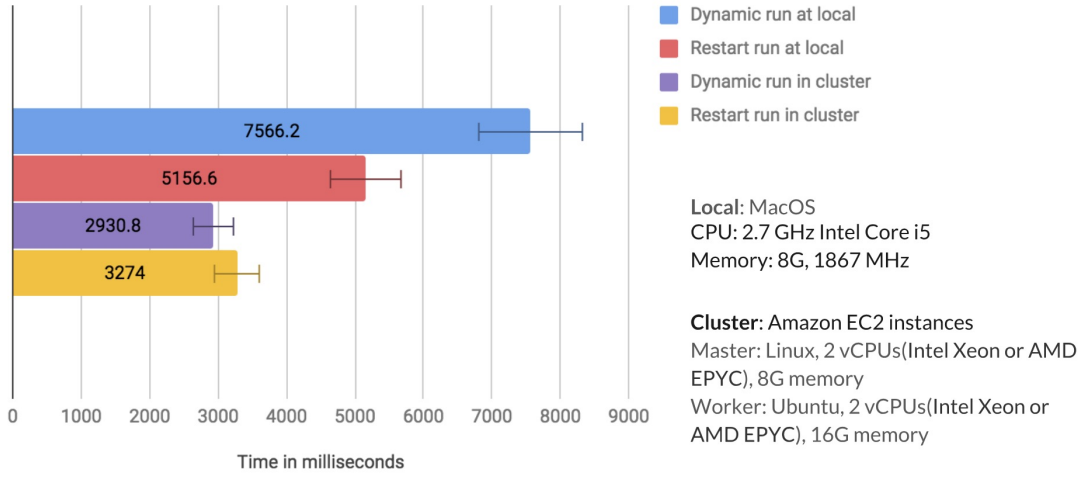


Figure 4.8: The transition time of Restart and Dynamic Update for Topology 1

Transition time

We modified the source code to track the transition time of each approach and run the application both at local environment and in the cluster. Each approach runs in both environments 3 times and the average transition time is computed.

The testing DAG operation for Topology 1 is increasing the parallelism of *source_0* by 2 and *Node_0* by 1. The transition time of this operation is measured in four cases: Restart approach running at local, Restart running in the cluster, Dynamic Update running at local, Dynamic Update running in the cluster. The result is shown in Figure 4.8. As it indicates, running at local basically takes more time than in the cluster to finish the transition due to the limitation of resources. The Dynamic Update approach is more sensitive to resources, as running at local has the longest transition time(more than 7.5 seconds) while running in the cluster has the shortest one(less than 3 seconds) in these four cases. Even though the Restart approach takes a bit more time than Dynamic Update when running in the cluster, it takes much less time(5.1 seconds) when running at local.

In addition to increasing the parallelism, the optimization process also requires decreasing it. Although the Restart approach has capabilities of adding or removing processors, we are not going to implement experiments on these two operations. One reason is that as an extended solution of original feature, Dynamic Update cannot do processor adding or removal. So there is no baseline for these two cases. The other reason is the optimization process only involves increasing and decreasing the degree of parallelization, no need for adding or removing processors. Taking all of these into consideration, our experiments on Topology 2 consider two approaches (Restart and Dynamic), two kinds of DAG operations (increase parallelism, decrease parallelism), and two deployment environments (local and cluster). The increase operation involves rising the parallelism of *Node_4* by 2 and *Node_12* by 1 simultaneously. The decrease operation is reducing

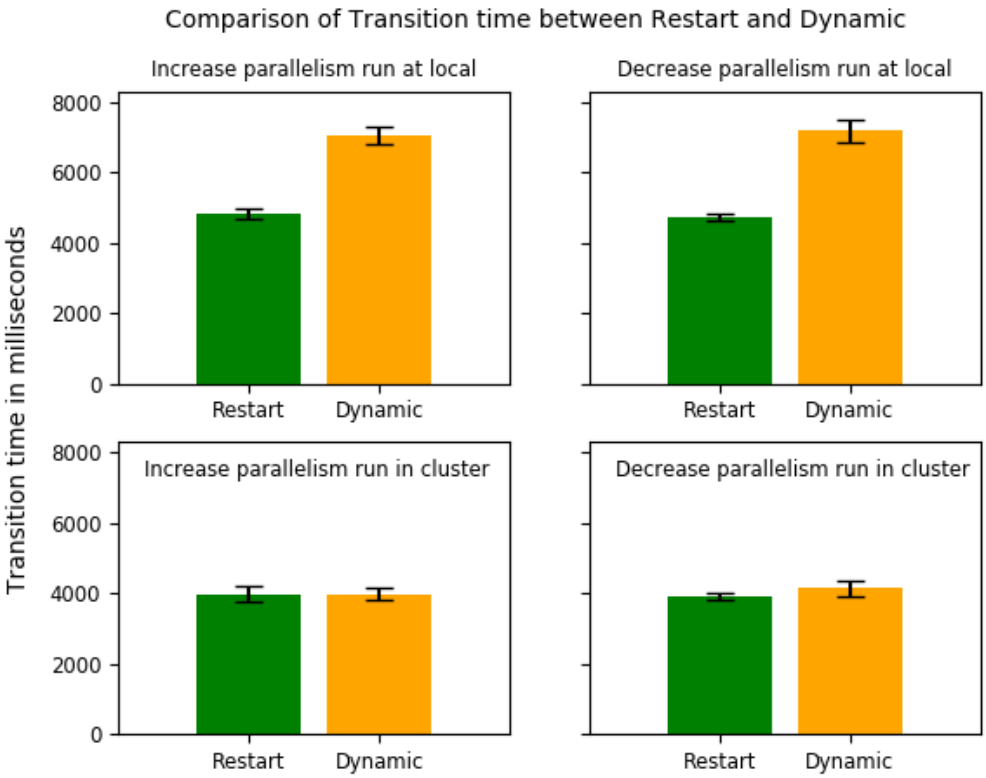


Figure 4.9: The transition time of Restart and Dynamic Update for Topology 2

the parallelism of *Node_4* by 2 and *Node_12* by 1 simultaneously. The initial degrees of parallelization of all processors and the targeted processors for operation are randomly selected.

The average transition time of experiments on Topology 2 is shown in Figure 4.9. At first we can see that the transition time of increasing and decreasing parallelism are almost the same for Restart and Dynamic respectively. This comes without surprise as these two operations in both approaches are indeed homogeneous. In Restart approach, the newly started DAGManager fetches new DAG from in-memory storage on Master and launches it. New DAGManager remember no previous states and does not care what operations have been done on the previous version of DAG. All it needs to know is how the new DAG looks like. In Dynamic case, JobOptimizer constructs a new DAG and send the changed processors to DAGManager. All DAGManager need to know is which processors have been changed and what is the parallelism now. There is no need for it to know what operations have been done either.

Same as Topology 1, in Topology 2 case running at local generally takes more time than in the cluster due to the limitation of resources. Although when running in the cluster, the transition time of Dynamic approach is close to Restart, Dynamic Update

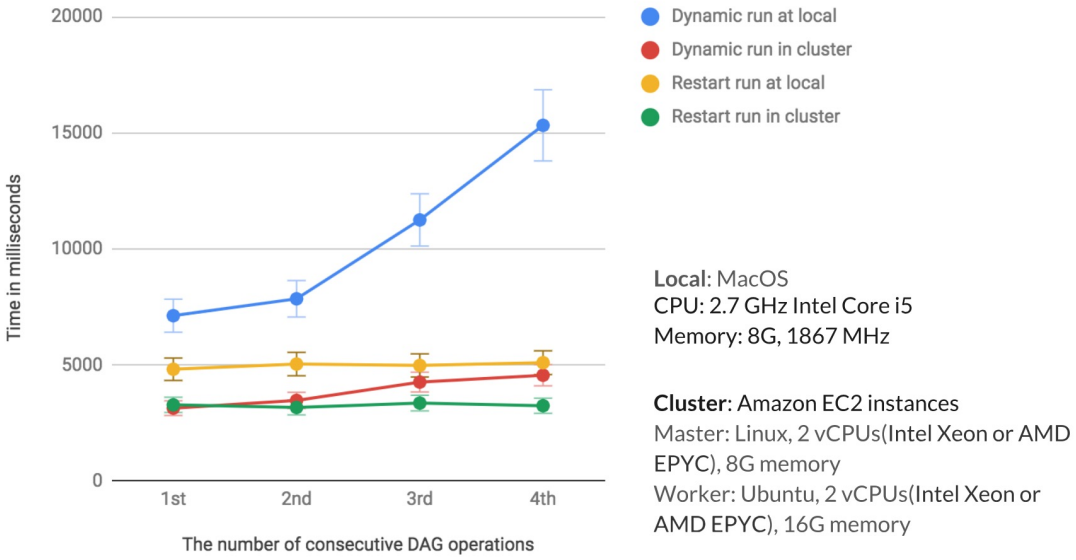


Figure 4.10: The transition time over 4 consecutive DAG operations for Topology 1

running at local has much longer transition process than Restart, with around 7s for increasing parallelism and 7.1s for decreasing parallelism.

It involves sequential changes to DAG as the optimization process goes. If the Dynamic Update approach is sensitive to resources, we could suppose that the transition time will arise over the number of DAG operations. The reason for that is every DAG operation leads to a new executor being started. As the number of DAG operations goes up, so does the number of running executors. Hence there will be less and less available resources.

To examine our assumption mentioned above, we conducted another set of experiments on both Topology 1 and Topology 2, in which we do 4 consecutive DAG operations in a row for each approach. Of course the sequence of operations in each running case is kept the same. In the experiment of Topology 1, in each DAG operation we increase the parallelism of *Node_3* and *Node_0* by 1. In the experiment of Topology 2, the operation each time is increasing the parallelism of *Node_1* by 1 while at the same time decreasing the parallelism of *Node_7* by 1. Targeted processors are again, randomly picked.

The results are shown in Figure 4.10 for Topology 1, and 4.11 for Topology 2, respectively. Sharing the same trend, these two figures validate our assumption. In both topologies, as the number of DAG operations goes up, the transition time of Dynamic Update running both at local and in the cluster increase. Especially when running at local, the transition time for Dynamic Update experiences a fast and sharp going-up. While the time arises much more slowly when the Dynamic Update approach running in the cluster. In contrast, the transition times of Restart approach in both local and cluster cases are very stable as the number of operations goes up. The operations in

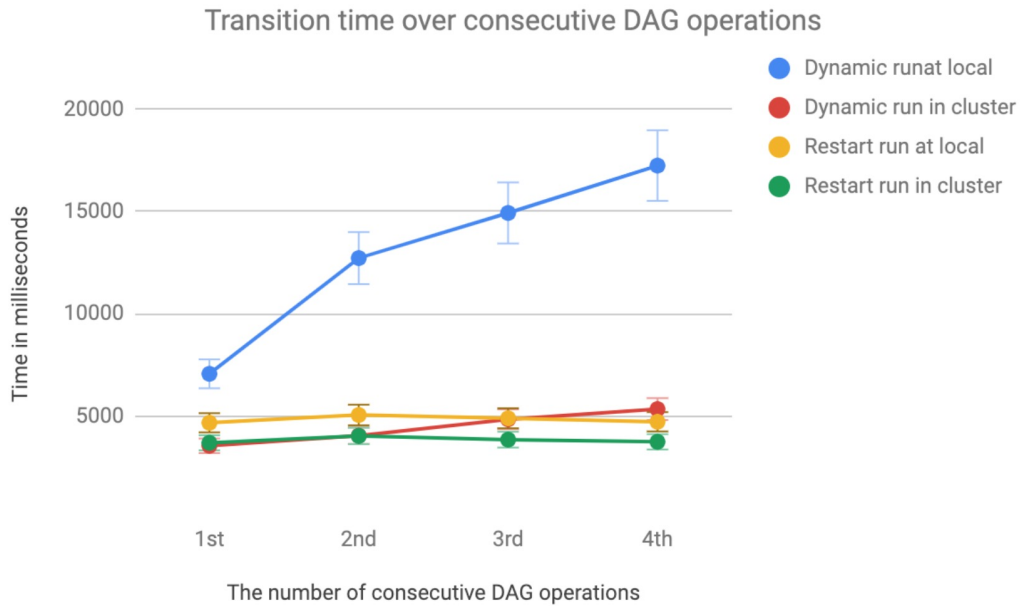


Figure 4.11: The transition time over 4 consecutive DAG operations for Topology 2

Topology 2 case generally take more time then their counterparts in Topology 1 case, since the DAG of Topology 2 is more complex than the on of Topology 1 in terms of size (15 processors compared to 12).

Sink receive-throughput

Gearpump provides system metrics such as send- and receive- throughput for monitoring purpose. Each throughput value is the accumulated number of messages during a time duration divided by this duration. As the time goes forward, the throughput value is calculated every 5 seconds (the system default interval). We explain in Chapter 5 how to fetch the metrics.

Experiments in this section are conducted in the cluster. In order to control the variables of our throughput experiments, we have to gather the throughput metrics in two approaches within the same time window. Figure 4.12 shows the schematic time window in which we gather the throughput metrics. We first submit the application, letting it run for while. Then both transitions in two approaches begin simultaneously. The beginning of the transition marks the start of the time window and it ends after a self-defined time duration.

In order to observe how DAG operations of both approaches affect throughput in long term, we conduct one DAG operation on Topology 1 for each approach, with observation time window as 60 seconds. The operation is increasing the parallelism of *Node_3* and *Node_0* by 1. Furthermore, we also observe the influence of 3 consecutive DAG operations on the sink throughput. This is conducted with Topology 2. Also, the DAG operations

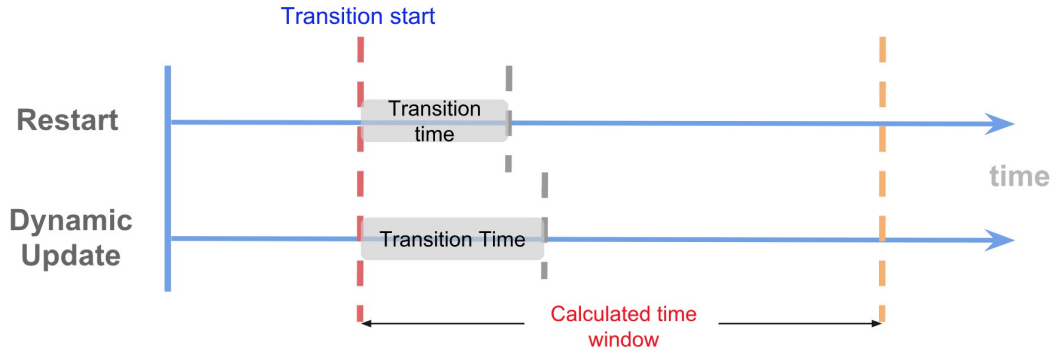


Figure 4.12: The calculated time window of sink receive-throughput

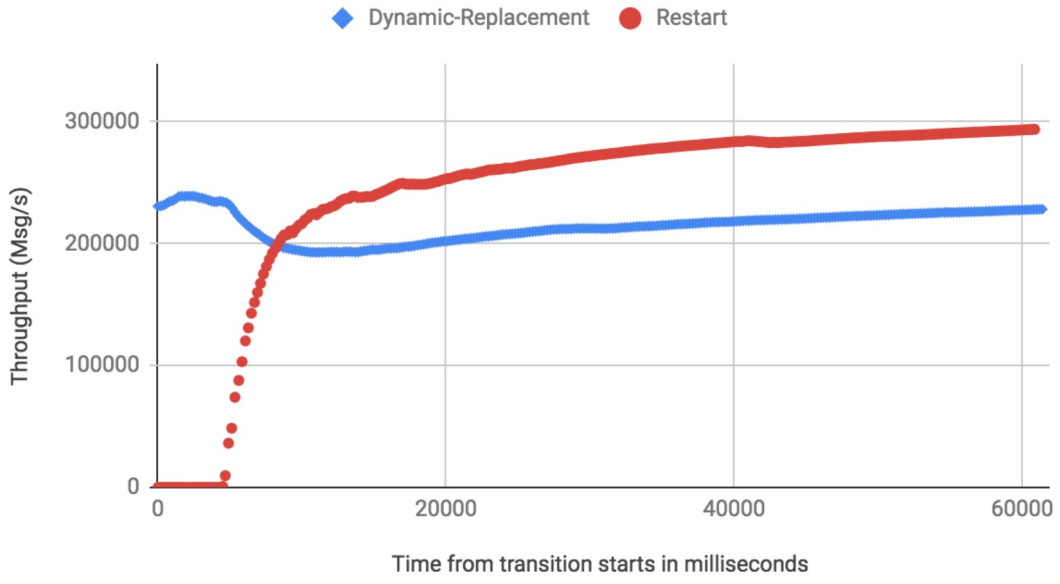


Figure 4.13: The Sink receive-throughput after transition starts in Topology 1

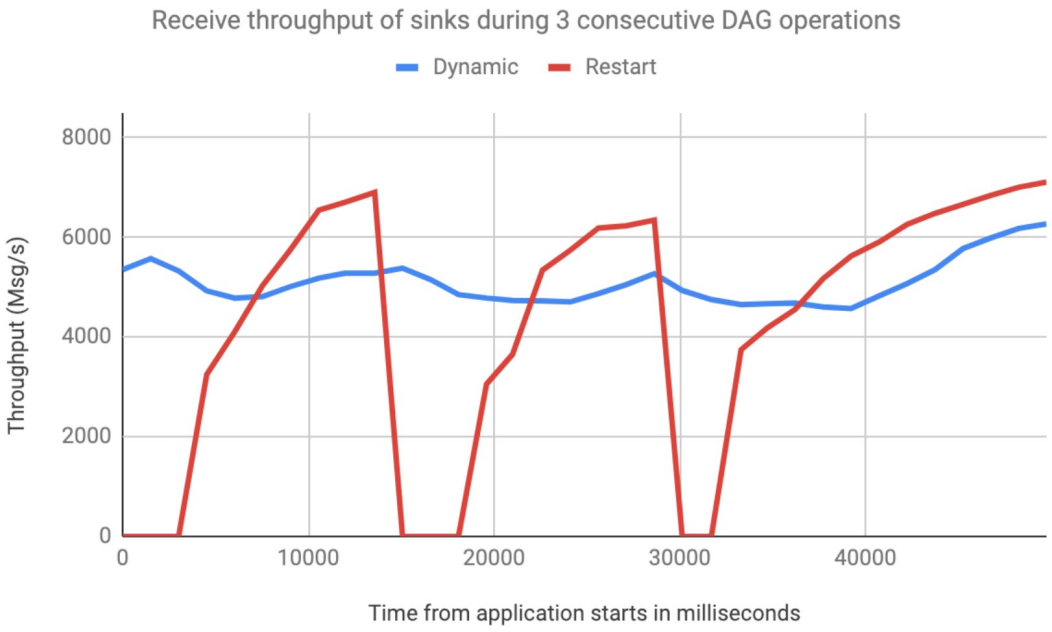


Figure 4.14: The Sink receive-throughput during 3 consecutive operations in Topology 2

are kept the same, which is increasing the parallelism of *Node_1* by 1 while at the same time decreasing the parallelism of *Node_7* by 1. The interval between 3 operations are 15 seconds.

The result of Topology 1 is visualized in Figure 4.13. As we can see, the transition time of Restart approach is about 4 seconds, which corresponds to the 0-throughput section of the red-dot line. However, after the application being recovered, the throughput increases dramatically in next 5 seconds and turns into slow arising after reaching around 23000 Msg per second. In the case of Dynamic Update approach (represented by the blue-diamond line), the throughput experiences a small decline due to the shutting-down of old source processor, and the turning point appears after around 10 seconds, the throughput eventually starts to arise. Note that after the transition process, the throughput in the Restart case always outperforms the throughput in the Dynamic Update case. The reason is that in the Restart case, all the tasks of recovered application will be launched in one executor just as before, so the communication between tasks is within the executor process which is efficient. In contrast, the application of Dynamic Update version launched new tasks of *source_0* on the new executor. So the communication between old tasks and new ones have to cross executors.

Figure 4.14 presents the sink throughput of 3 consecutive operations in Topology 2. We can see that when each DAG operation begins (corresponds to approximately 0, 15000 and 30000 in axis X), the sink throughput of Restart approach (red curve) appears a significant drop to 0, while the throughput of Dynamic approach (blue curve) only declines slowly and slightly. After the transition process finished, the throughput

of Restart rebounds quickly, while the one of Dynamic begins to increase slowly. This shares the same trend as in Figure 4.13.

We can also get some information of transition time of these two approaches. The transition time of Restart, which corresponds to time duration of 0 throughput of the red curve, is comparatively stable. Note that the transition time is defined as the time duration from when transition starts to the time that every processor works normally. Hence we cannot identify the time, from when the blue curve starts to drop to when it starts to arise, as the transition time. For example, in the topology shown in Figure 4.7, if we change the parallelism of *Node_5*, the throughput curve starts to drop because the old *Node_5* is stopped immediately as the transition starts. However, *Source_0* now gets more capability to send messages to *Node_0* before a new *Node_5* is launched. So the curve may start to arise as there would be more messages sent to *Node_0* hence more received by *Sink_0*. In one word, the arise of throughput curve does not necessarily mean the transition process ends and new processors begin to work. Despite this mismatching, we can still observe from the blue curve, that during the second and third transition, it take longer for the Dynamic application to turn its throughput from dropping to rising. This phenomenon indicates the transition time increases as the number of DAG operations goes up.

Taking all these results into account, we find that the Restart approach is superior to Dynamic Update. Firstly, the transition time of Restart is shorter when resource is limited(running at local), and stable when doing multiple DAG operations. The hierarchical supervision tree enables the efficient shutting-down and starting of AppMaster and its children, which contributes a lot to the quick recovery. Secondly, the throughput performance of Restart version application is better than the Dynamic Update one. In addition to the fast recovery, the other reason is Dynamic Update is more resource-consuming, which leads to more cross-executor communication between tasks.

This chapter answered *RQ1*, as we found that 'Dynamic DAG' is not competent enough for our runtime DAG operations, and we proposed a good solution which acts as a cornerstone of 'online' optimization. Next its time to implement our JobOptimizer based on the Restart approach.

5

Online Optimization of Job Parallelization

With the support from Restart approach, now we are able to implement an 'online' version of optimization. First we are going to shortly explain Bayesian Optimization algorithm and the implementation we select. Then we will describe the implementation details of our JobOptimizer, which will give the answer to our *RQ2*.

5.1 Introduction to Bayesian Optimization

An optimization problem is to find an input x which maximizes an objective function $f(x)$. Usually many techniques are based on a few assumptions of the objective function. $f(x)$ are assumed to have a known mathematical representation and be convex. However, many real cases do not conform to these strong assumptions. Derivatives and convexity are often unknown. Bayesian Optimization (BO) is a powerful algorithm for finding the extrema of objective functions that do not have a closed-form expression, but one can obtain observations of this function at sampled values. It is particularly useful when derivatives are not known or the function is not convex (Brochu et al., 2010). BO typically works by assuming the unknown function was sampled from a Gaussian Process, and maintains a posterior distribution for this function as observations are made.

5.1.1 Gaussian Process

Just like a Gaussian distribution is a distribution over a random variable, specified by its mean and variance, a Gaussian Process is a distribution over functions, specified by its mean function and covariance function. It can be represented as follow:

$$f(x) \sim GP(\mathrm{m(x)}, \mathrm{k(x,x')}) \tag{5.1}$$

$m(x)$ is the mean function and $k(x, x')$ is the covariance function which is also known as kernel. For easier understanding purpose, we can think of GP as a function. Common functions return a value f for an input x. Instead of giving a certain value of f, GP returns a mean and a variance of a normal distribution over possible values of f given an

x. In practice we often assume the prior mean $m(x)$ as a constant 0. A popular choice of covariance function is the squared exponential function shown in equation 5.2

$$k(x_i, x_j) = exp(-\frac{1}{2}\|x_i - x_j\|^2) \tag{5.2}$$

Note that this kernel function approaches 1 as x_i and x_j get close to each other and 0 as they go further apart. The closer two inputs get, the higher kernel function value is. To interpret what GP can be used for, let's assume we try to describe an unknown function $f(x)$ and we have already evaluated its values at $x_{1:t}$ as $f_{1:t}$, in which t is an integer represents how many xs we have been evaluated. $1 : t$ is 1 to t. By fitting $x_{1:t}$ and $f_{1:t}$ to a GP with the prior mean 0 and squared exponential kernel, we get a multi-variant normal distribution $N(0, K)$ in which K is computed as shown in equation 5.3. Obviously the diagonal values of K matrix is 1 (when in the noise-free environment) and K is symmetric.

$$K = \begin{pmatrix} k(x_1, x_1) & \dots & k(x_1, x_t) \\ \vdots & \ddots & \vdots \\ k(x_t, x_1) & \dots & k(x_t, x_t) \end{pmatrix} \tag{5.3}$$

Next with the help of this newly fitted GP, we can predict the function value f_{t+1} at any given input x_{t+1}. By the properties of GP $f_{1:t}$ and f_{t+1} are joint Gaussian distribution as described in equation 5.4. Using the Sherman-Morrison-Woodbury formula (Sherman and Morrison, 1950), one can get the expression of posterior distribution of f_{t+1} as shown in equation 5.5.

$$\begin{bmatrix} f_{1:t} \\ f_{t+1} \end{bmatrix} \sim N(0, \begin{bmatrix} K & k \\ k^T & k(x_{t+1}, x_{t+1}) \end{bmatrix}) \tag{5.4}$$

where

$$k = \begin{bmatrix} k(x_{t+1}, x_1) & k(x_{t+1}, x_2) & \dots & k(x_{t+1}, x_t) \end{bmatrix}$$

$$P(f_{t+1}|D_{1:t}, x_{t+1}) = N(\mu_t(x_{t+1}), \sigma_t^2(x_{t+1})) \tag{5.5}$$

where

$$\mu_t(x_{t+1}) = k^T K^{-1} f_{1:t}$$
$$\sigma_t^2(x_{t+1}) = k(x_{t+1}, x_{t+1}) - k^T K^{-1} k$$

5.1.2 Acquisition function and Bayesian Optimization

Bayesian Optimization is an iterative process of computing f_{t+1} and update GP given an unknown objective function $f(x)$. Each iteration determines next input x. This optimization process is guided by an acquisition function, which is typically defined such that high acquisition value corresponds to *potentially* high values of the objective function. To achieve potentially high objective value, maximizing the acquisition function should suggest the x either leads to a high predicted value of $f(x)$, or uncertainty is

great at x. In other words, the acquisition function is designed to do a trade-off between exploitation and exploration.

One example of the acquisition function is Probability of Improvement (PI) proposed by Kushner (1964). As its name indicates, PI is the probability of improvement over the current largest objective value $f(x^+)$, where $x^+ = argmax_{x_i \in x_{1:t}} f(x_i)$. Hence the expression of PI is defined as in equation 5.7, where $\varphi(\bullet)$ is normal cumulative distribution function. However, there is a drawback of this equation, which is that it is pure exploitation. Points that have a high probability of being greater than $f(x^+)$ will be selected, while points which potentially give larger gains but with more uncertainty will not. To mitigate that a small modification is made upon the equation 5.7 by adding a trade-off parameter ξ. The PI in practice is specified as equation 5.8. This ξ encourages the algorithm to explore, and it is decided by user. Kushner (1964) suggested a schedule for ξ, which initialize ξ to be very high in the beginning of optimization, and reduce it toward zero as the optimization process goes.

$$\begin{aligned} PI &= P(f(x) \geq f(x^+)) \\ &= \varphi(\frac{\mu(x) - f(x^+)}{\sigma(x)}) \end{aligned} \tag{5.6}$$

$$\begin{aligned} PI &= P(f(x) \geq f(x^+) + \xi) \\ &= \varphi(\frac{\mu(x) - f(x^+) - \xi}{\sigma(x)}) \end{aligned} \tag{5.7}$$

Another commonly used acquisition function is Expected Improvement (EI) (Mockus et al., 1978). Suppose $x*$ is the value where the maximum of the objective function locates. Then $f(x*)$ is the maximum value. At the iteration of $t+1$, we want to find the x_{t+1}, such that the expectation of the distance from $f(x_{t+1})$ to $f(x*)$ is minimized. This x_{t+1} is shown in equation **??**. However, we do not know the true maximum objective value and where it locates. To overcome this problem, equation 5.9 is proposed, which tries to find the x that maximize the expected improvement compared to the current largest value $f(x^+)$. We can obtain a analytical expression of Expected Improvement based on Gaussian Process as in equation 5.10, from which x is able to be computed.

$$\begin{aligned} x_{t+1} &= \arg\min_x E(\|f_{t+1} - f(x*)\||D_t) \\ &= \arg\min_x \int \|f_{t+1} - f(x*)\| p(f_{t+1}|D_n) df_{t+1} \end{aligned} \tag{5.8}$$

$$x = \arg\max_x E(max\{0, f_{t+1}(x) - f(x^+)|D_t) \tag{5.9}$$

$$EI(x) = \begin{cases} (\mu(x) - \mu^+ - \xi)\Phi(Z) + \sigma(x)\phi(Z) & \sigma(x) > 0 \\ 0 & \sigma(x) = 0 \end{cases} \tag{5.10}$$

$$Z = \frac{\mu(x) - \mu^+ - \xi}{\sigma(x)}$$

There are a few other acquisition functions such as GP-UCB (Srinivas et al., 2010) and Thompson Sampling (Thompson, 1933). These acquisition functions share the same goal which is seeking a trade-off between exploitation and exploration to drive the optimization. The pseudocode of Bayesian Optimization is shown in Algorithm 1. Inside the loop first a x_t is computed which maximize or minimize the acquisition function u. Then the objective value is evaluated. This observed value is the sum of $f(x_t)$ and a noise ε_t. In the last step the knowledge base $D_{1:t}$ is added with x_t and observed value f_t, and GP is updated accordingly. Note that there is no end of this for loop which requires us to consider the ending rules in practice. In our implementation convergence criteria is defined and discussed in Section 5.2.

Algorithm 1 Bayesian Optimization

1: for t = 1,2,... do
2: determine x_t by optimization the acquisition function u over GP:
3: $x_t = \text{argmax}_x u(x, D_{1:t-1})$
4: Evaluate the objective function: $f_t = f(x_t) + \varepsilon_t$
5: Augment data $D_{1:t} = \{D_{1:t-1}, (x_t, f_t)\}$ and update GP
6: end

5.2 JobOptimizer enabled by Run-Time DAG Operation

In this section, we describe the design and implementation of our JobOptimizer. Considering integrating algorithm into our module, we will first talk about the selection of algorithm implementation. Based on that we introduce the architecture and workflow of JobOptimizer. Lastly more details such as objective value computation and convergence criteria will be discussed.

5.2.1 The selection of algorithm implementation

There are two approaches of integrating algorithm into JobOptimizer. One is the tight-embedded approach, which means embedding algorithm libraries into the Gearpump source code. With this approach we are required to find and download the algorithm libraries (and all dependencies), then include these libraries to Gearpump source code. The algorithm can be imported as regular project libraries. However, this would inevitably increase the code complexity of Gearpump. Furthermore, Gearpump is imple-

mented in Scala and Java. Since most of the algorithm implementations we found are in Python, there is compatibility problem if we want to use Python libraries in Scala code.

The other approach is the loose-coupled approach, which is setting algorithm implementation as an external service. This approach allows Gearpump system and the optimization service work independently. Instead of considering importing algorithm libraries into Gearpump source code, we take care of the communication between Gearpump and optimization service. In addition to avoiding compatibility problem, this solution brings flexibility in such a way that: any change to Gearpump system or optimization service does not affect the other.

Table 5.1: A comparison between tight-embedded and loose-coupled approach

	pros	cons
Tight-embedded	Enable importing algorithm anywhere conveniently in the project.	Compatibility issue: most implementations are in Python, Code complexity: complex dependency tree has to be included, Code redundancy: many other functions of the library are never used.
Loose-coupled	No need to increase code complexity, Independence: Gearpump and service can work independently, Flexibility: any change of Gearpump or service not affect the other.	Need to consider cooperation and communication between Gearpump and service.

The pros and cons of each solution are listed in Table 5.1. Parallelization optimization is a good-to-have feature but not a indispensable functionality. For example, if we run an application with very simple topology, performing an optimization might not be necessary. Besides, in some cases users do not want to perform optimization as they already know the best parallelization according to previous experience. So it is not a good choice to embed the algorithm implementation into the Gearpump source code, not to mention there may exist a complex dependency tree. To keep the source code as clean as possible, we decide to introduce the algorithm implementation as an external service, which cooperate with our JobOptimizer module to conduct the optimization process.

5.2.2 Advisor: REST service for black-box optimization

After investigation we decided to use a software named Advisor [1]. Advisor is a simplified and open source version of Google Vizier, which is a Google internal service for performing black-box optimization. Google Vizier is used to optimize many of our machine learning models and other systems, and also provides core capabilities to Google's

[1] Advisor, https://github.com/tobegit3hub/advisor

Cloud Machine Learning HyperTune subsystem (Golovin et al., 2017). Following advantages motivated us to select Advisor as our Bayesian Optimization implementation:

- Easy to use API and SDK: Advisor provides simple REST APIs for us to interact with the service.
- State-of-art algorithms: algorithms are well implemented.
- Easy to check optimization status: Advisor provides a Web UI to expose optimization information to users.
- Simple process thanks to abstractions: Advisor simplifies the optimization process by defining and using two main abstractions: Trial and Study. We introduce them next.

A **Trial** is a list of parameter values x which leads to a single evaluation of $f(x)$. A trial has two status: "Completed" which means x has been evaluated and the objective value $f(x)$ has been assigned to it, and "Pending" which means it has not been evaluated yet. A **Study** represents a single optimization run over a feasible space. Each Study contains a configuration describing the feasible space of x, as well as a set of Trials (Golovin et al., 2017). The work flow of Advisor (as well as Vizier) is described in Algorithm 2. *client* represents a user program which starts an optimization process. At first *client* defines and submit a study task with configurations to Advisor. At the beginning of optimization, *client* load the study from Advisor. Within the iteration, *client* ask Advisor for a suggestion which is a trial containing suggested parameters. Then *client* evaluate this trial by running the system being tuned with suggested parameters. Finally the client reports the gathered results to Advisor, who will then update its model with newly-received results.

Algorithm 2 The work flow of Advisor

```
client.LoadStudy(study config)
While (not client.StudyIsDone()):
    # get a Trial for evaluation
    trial = client.GetSuggestion()
    # Evaluate the objective function at the trial parameters
    metrics = RunTrial(trial)
    # report back the results
    client.CompleteTrial(trial, metrics)
```

In order to test if the Bayesian Optimization algorithm works correctly as well as exploring the REST APIs provides by Advisor, we conducted a simple experiment. We tries to find the maximum of a known objective function shown in Equation 5.11. The plot of this function is shown in Figure 5.1. We can compute that the maximum of this objective function occurs at x equals 2, which can also be observed in the figure.

$$f(x) = e^{-(x-2)^2} + e^{-\frac{(x-6)^2}{10}} + \frac{1}{x^2+1} \tag{5.11}$$

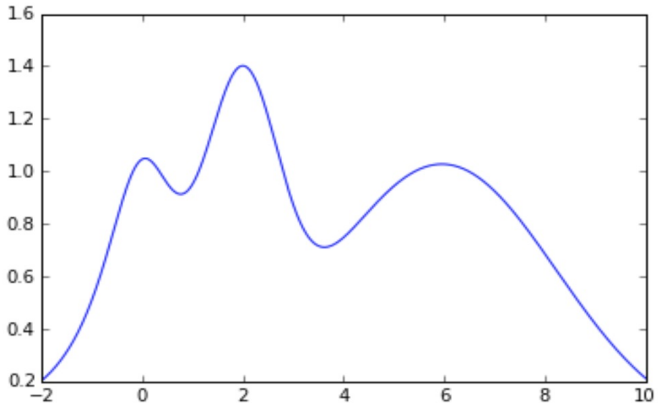


Figure 5.1: The plot of test objective function

Next we set up and start Advisor service. According to the document, we can create a study by posting HTTP request with form data. We use a tool named Advanced REST Client [2] to send HTTP request to Advisor server and check the response. We can specify attributes of the posted Form data: "algorithm" which specify the algorithm we want to use, "name" of the study, "study_configuration" which is the configuration of study. For this specific test example, the study configuration in JSON format is shown in **??**. The meaning of each attribute is explained in Table 5.2.

Listing 5.1: The study configuration of test example

```
{
  "goal": "MAXIMIZE",
  "randomInitTrials": 3,
  "maxTrials": 50,
  "params": [
    {
      "parameterName": "x",
      "type": "DOUBLE",
      "minValue": -2,
      "maxValue": 10,
      "scalingType": "LINEAR"
    }
  ]
}
\label{list:test_config}
```

[2] Advanced REST Client, https://install.advancedrestclient.com/install

Table 5.2: Attributes of study configuration and corresponding explanation

Attributes	Comments
goal	goal of the optimization, "MAXIMIZE" or "MINIMIZE"
randomInitTrials	how many random tries before suggesting based on the selected algorithm
maxTrials	the maximum number of iteration
parameterName	name of the parameters
type	type of the parameters, can be "DOUBLE", "INTEGER", "CATEGORICAL", " DISCRETE"
minValue	minimum value of the parameter to try, lower boundary of the feasible space
maxValue	maximum value of the parameter to try, upper boundary of the feasible space
scalingType	scaling type suggested by user, indicates how the objective value changes as the parameter changes, can be "LINEAR" or "LOG"

After posting the request of creating study, we interact with the Advisor service directly by sending HTTP request and receiving HTTP response. First we send a request asking a trial suggestion of x from Advisor. Then an objective value at x is computed by our Python program which implements the objective function shown in Equation 5.11. Next we send a request with computed objective value to update the trial. These steps are repeated until convergence. In this test case, convergence means Advisor always suggests x which are close enough to the targeted optimal value 2.

Advisor provides a Web UI for us to monitor the optimization process. The screenshot of trial list of this test optimization problem is shown in Figure 5.2. This table shows the attributes of trials: study name, trial name (which algorithm it uses), parameter values, objective values and status. The first 3 steps are random trials, gathering initial information about the objective function. We can observe the algorithm converges after 13 iterations, when Advisor always suggests x which are very close to optimal value 2 (within the red square in Figure 5.2). This phenomenon reminds us that: the algorithm does not know how and when to converge and stop, we have to define the convergence criteria by ourselves. The convergence criteria of our JobOptimizer is discussed in next section. From this test example, we also figure out what REST APIs to use in order to conduct an optimization task. We list the APIs which we have to use in our JobOptimizer in Table 5.3.

5.2.3 The implementation of JobOptimizer

For the purpose of answering *RQ2*, we are going to discuss the design and implementation details of JobOptimzier in this subsection. As we have mentioned in Section 4.2.1, JobOptimizer should be a child of AppMaster in the actor supervision tree. On one hand, JobOptimizer serves each application as a functionality, just like DAGManager

Id	Study Name	Name	Parameter Values	Objective Value	Status
6	REST_test01	RandomSearchTrial	{"x": 7.757878870568867}	0.750515	Completed
7	REST_test01	RandomSearchTrial	{"x": -0.7148751277367107}	0.673432	Completed
8	REST_test01	RandomSearchTrial	{"x": 5.978599364600422}	1.02717	Completed
9	REST_test01	BayesianOptimizationTrial	{"x": 2.6104019591701295}	1.1339	Completed
10	REST_test01	BayesianOptimizationTrial	{"x": 4.060909011054318}	0.75807	Completed
11	REST_test01	BayesianOptimizationTrial	{"x": 9.999939956457233}	0.2118	Completed
12	REST_test01	BayesianOptimizationTrial	{"x": 0.9830272469456749}	0.94475	Completed
13	REST_test01	BayesianOptimizationTrial	{"x": -1.9997996656102908}	0.20167	Completed
14	REST_test01	BayesianOptimizationTrial	{"x": 1.9103730806912003}	1.3948	Completed
15	REST_test01	BayesianOptimizationTrial	{"x": 6.823445760281274}	0.95547	Completed
16	REST_test01	BayesianOptimizationTrial	{"x": 5.065193841799951}	0.95392	Completed
17	REST_test01	BayesianOptimizationTrial	{"x": 8.879618123431577}	0.44892	Completed
18	REST_test01	BayesianOptimizationTrial	{"x": 1.6460064263163958}	1.30202	Completed
19	REST_test01	BayesianOptimizationTrial	{"x": 2.134910713991519}	1.38638	Completed
20	REST_test01	BayesianOptimizationTrial	{"x": 2.0126667551511037}	1.40177	Completed
21	REST_test01	BayesianOptimizationTrial	{"x": 1.9889270134772201}	1.4017728	Completed
22	REST_test01	BayesianOptimizationTrial	{"x": 2.004696986219149}	1.40188	Completed
23	REST_test01	BayesianOptimizationTrial	{"x": 2.001138248544999}	1.401897	Completed

Figure 5.2: The screenshot of trial list of test example

Table 5.3: REST APIs of Advisor used in JobOptimizer

URL	Method	Description
/v1/studies	POST	create a study with given configuration, which is in JSON format
/v1/studies/< *name* >	DELETE	delete a study whose name is < *name* >
/v1/studies/< *name* >/suggestions	POST	ask for a suggestion from a study
/v1/studies/< *name* >/trials/< *id* >	PUT	update a trial whose id number is < *id* >
/v1/studies/< *name* >/trials/< *id* >	GET	get the information of a trial, including parameters, status and objective value

and other functional actors. On the other, JobOptimizer need to communicate and cooperate with other functional actors to achieve its goal. Hence it should be created and managed by AppMaster.

JobOptimizer has three main tasks. With Advisor as an external service to give parameter suggestion, JobOptimizer's first task is to communicate and coordinate with Advisor. After getting a suggestion from Advisor, its second task would be conducting runtime DAG operation according to the suggested parameters. This is supported by Restart approach we discussed in Chapter 4. The last task is computing the objective value from gathered metrics, and updating the Trial in Advisor server. We design our JobOptimizer as shown in Figure 5.3. Advisor is set up as an external web service as shown on the left. JobOptimizer module consists of three components, which correspond to three tasks we defined above.

JobOptimizer Client, as the name indicates, is responsible for communicating with Advisor using REST APIs. As the data format of communication is JSON, it has to construct JSON string for sending HTTP request, and convert received JSON objects into Scala objects. It implements functions responsible for creating studies, asking for suggestions and updating trials, supported by REST APIs in Table 5.3. **DAG Operations** component implements the Restart approach for doing runtime DAG operation. It receives the suggested parameters from Client and transform DAG according to new parameters. Implementation details are described in Section 4.2.1. **Metrics Manipulation** component gathers metrics from Gearpump system and compute the objective value from them. AppMaster has a child actor named MetricsService which contains and reports metrics. Available metric types are receive-throughput, send-throughput, receive-latency and process time. To fetch metrics, JobOptimizer need to send a message named $QueryHistoryMetrics$ to AppMaster. Then AppMaster forwards the message to

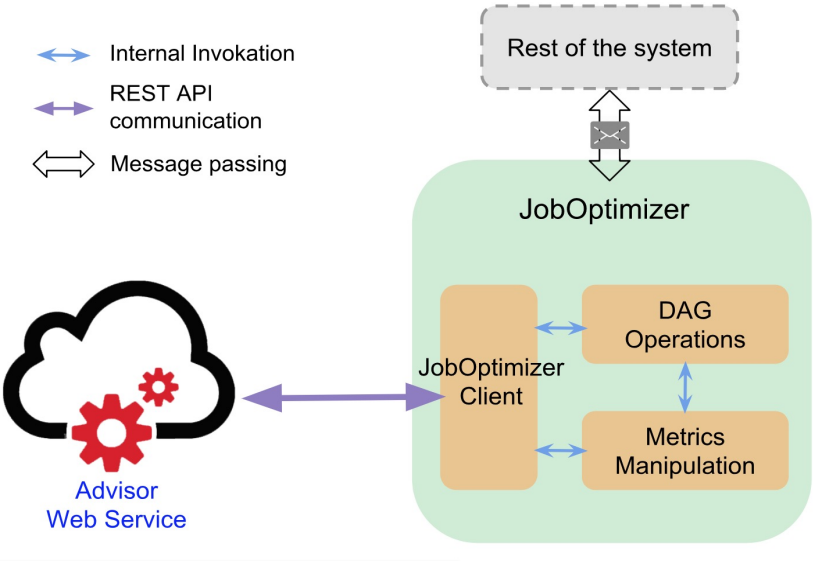


Figure 5.3: The architecture of JobOptimizer

MetricsService, who sends metrics as a response directly to JobOptimizer. This fetching process is explained in Figure 5.4. We do not send *QueryHistoryMetrics* directly to MetricsService, because in this way there is no need for JobOptimizer to hold an actor reference of MetricsService. AppMaster creates and knows every functional actors, so it is the clear and right way to send request to AppMaster.

Knowing functionalities of each component, we are going to introduce how they work together. The workflow of JobOptimizer is draw in Figure 5.5. The "Start" marks the successful submission of Gearpump application. We provide configuration options for the user to enable or disable JobOptimizer. If the JobOptimizer is enabled, AppMaster creates an instance of JobOptimizer during initialization. The prerequisite of normal op-

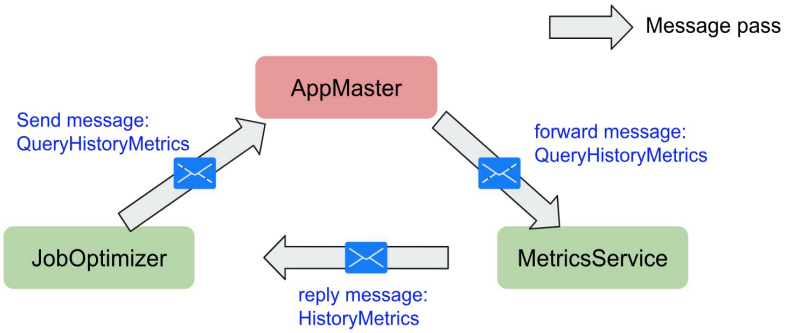


Figure 5.4: How JobOptimizer fetches metrics

eration of JobOptimizer is that the Advisor service is set-up. If Advisor is not available, JobOptimizer does nothing, without affecting the execution of application.

We let AppMaster send a *ApplicationReady* message to JobOptimizer when the application is ready to run. The Client component of JobOptimizer then starts doing its job by creating a study in the Advisor service. For creating the study, JobOptimizer analyses the submitted DAG and extracts a parameter set. One parameter for each processor of the DAG, representing the degree of parallelization of this processor. After the creation of study, JobOptimizer start the optimization iterations. Each iteration consists of following five steps:

1. Ask for a new Trial: Client ask Advisor for a new Trail, containing suggested parameter values.

2. Make runtime DAG operations: runtime DAG operation is conducted with Restart approach, a new DAG is created and made available according to the suggested parallelization.

3. Fetch Metrics: JobOptimizer fetches metrics in the way shown in Figure 5.4.

4. Compute objective value: Metrics Manipulation component receives metrics and compute objective value from metrics. As to what kinds of metrics is used and how to compute the objective value, we will discuss below soon.

5. Update the Trial with objective value: Client send a request to Advisor service to update the corresponding Trial. Advisor service updates its fitted Gaussian Process automatically.

At the end of each iteration, JobOptimizer judges whether the optimization task is finished. The stop criteria is discussed below. If JobOptimizer determine the optimization should be stopped, either because the optimal parameter set has been found, or the algorithm failed to converge within predefined maximum steps, JobOptimizer stands by and lets the new DAG run. Otherwise the above-mentioned five steps of iteration is repeated.

Objective value computation

To measure the performance of the application, we use the receive-throughput as the indicator. Send-throughput, receive-latency and process time also indicates the performance, but high receive-throughput at sink processors is a more apparent and straightforward goal for the application. Receive-throughput of one single sink processor or a subset of sink processors is not enough to measure the overall performance of the DAG.

We compute the summation of receive-throughput of all sink processors at a certain timestamp as the objective value. The metrics provided by Gearpump actually depends on a package named *codahale.metrics* [3]. *codahale.metrics* is able to compute

[3]codahale.metrics, https://github.com/codahale/metrics

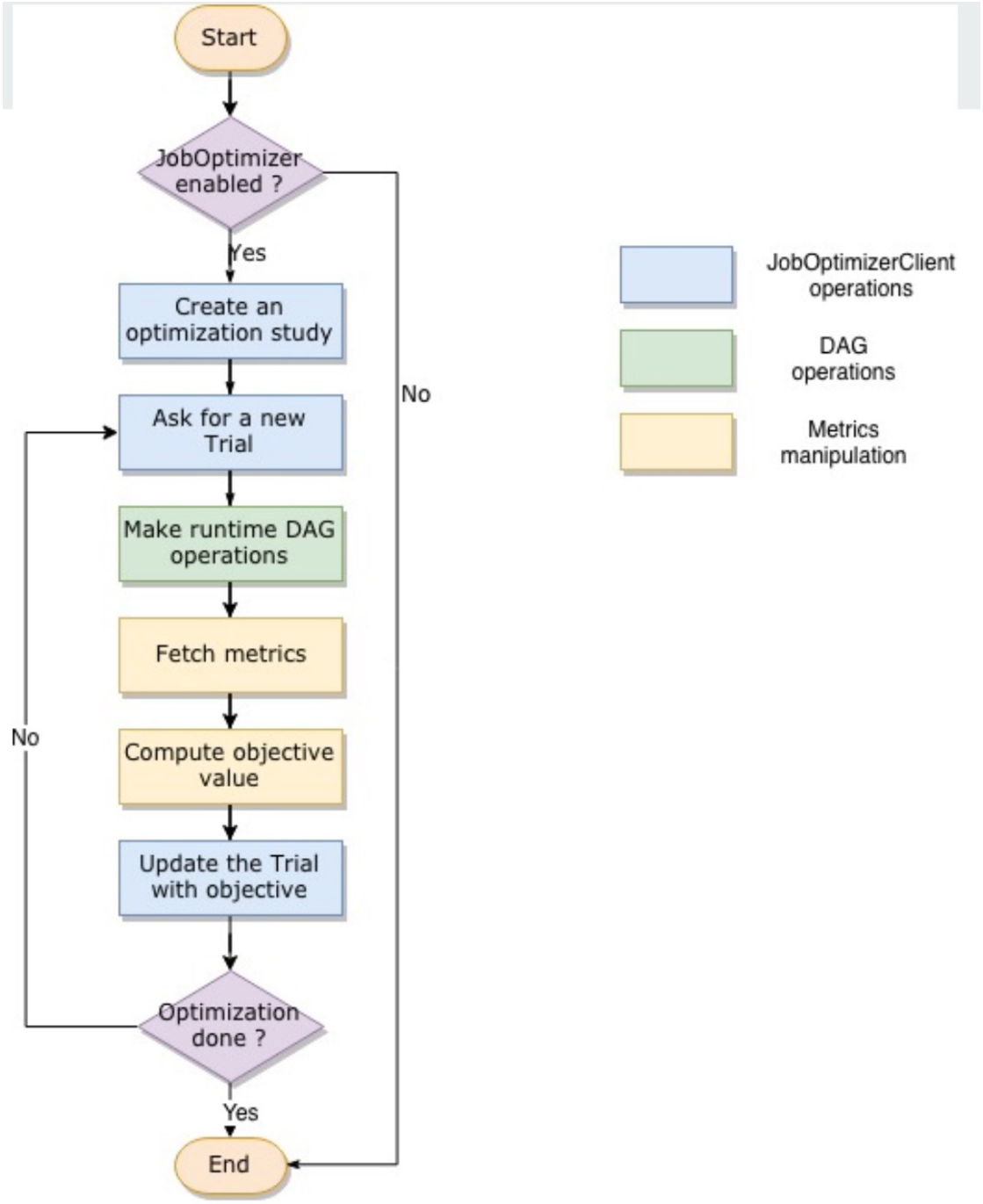


Figure 5.5: The workflow of JobOptimizer

the moving-average of input values, which is used by Gearpump MetricsService to compute send- and receive throughput. The time window of the moving-average used by Gearpump is 1 minute. Therefore the receive-throughput we fetch is not instantaneous but an computed average. In the environment of distributed computing, it is difficult to get real-time instantaneous metrics of each component of the system. In addition, the instantaneous values are often with great noise because of random errors or unstable network conditions. Therefore computing moving average is a appropriate way to increase accuracy.

Although the moving-average is used, in practice the variance of metrics cannot be removed. For example, there is difference (sometimes very large) among the receive-throughput measured in the same condition (same DAG and same time window). Since in Advisor's documents we do not find the relevant information about mitigating the influence of noise, we try to improve the measurement accuracy by extending the running time of the application. We fetch the metrics after the application running for 1 minute.

Stop criteria

As we introduced in 5.1.2, BO has no internal stop criteria and repeats the for loop in Algorithm 1 forever. We have to define our own stop criteria according to our own requirements. Optimization process costs computational resources and we want the application to run normally as soon as possible.

JobOptimizer is designed to stop its work in two cases. One is the optimal parallelization has been found and the performance of application is maximized. The other is the maximum number of predefined tries is reached but no optimal parallelization being found. The first case is called convergence, which is JobOptimizer finds the optimal parameter set. As shown in the Advisor example described in Subsection 5.2.2, BO does not know when convergence occurs (when BO always suggest the points close to x equals 2), we have to determine by ourselves. In the second case, we choose the parameter set which has been evaluated with the highest objective value to run the application.

Theoretically speaking, in our case the convergence occurs when Advisor always suggest the same set of parameters. However, taking variance into consideration, some small changes to one parameter set usually does not affect the performance greatly. Hence we cannot assume the optimal parameter set is one specific combination of parameters without tolerance of small changes. A set of combinations of parallelization, who is 'close' to each other are acceptable.

To measure how 'close' of two numerical parameter sets, we can convert two parameter sets into two vectors and compute their Euclidean Distance. However, suggesting two similar parameter sets in a row might be random tries and is not persuasive enough to conclude that the optimal parameter set is around these two. Therefore we consider a sequence of recent consecutive suggestions, computing their relative standard deviation (RSD). RSD is the division between standard deviation and mean of a sequence. If the RSD is below some threshold, we think BO is suggesting points close to each other. In terms of implementation, we use a queue to store recent trials. The length of queue can be configured.

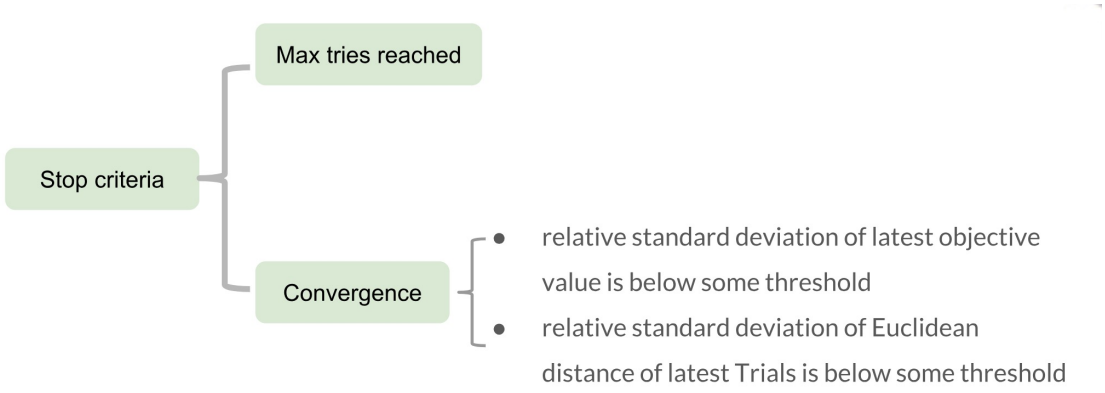


Figure 5.6: The stop criteria of JobOptimizer

When the suggested points are close to each other, we can not consider it as convergence if the metrics measured at these points vary significantly. Therefore another relative standard deviation of receive-throughput has to be computed and compared to a threshold, in order to determine whether the performance varies a lot. The sequence of objective values can also be obtained from above-mentioned queue. The stop criteria of JobOptimizer is summarized as in Figure 5.6.

Next we are going to evaluate our JobOptimizer by optimizing some example DAGs.

5.3 Evaluation

To examine whether our JobOptimizer works, how good or bad it performs as well as answering *RQ3*, we conduct a series of experiments. The first set of experiments are targeted at analysing JobOptimizer's behavior, and investigating its capability of optimization. To figure out how good or bad JobOptimizer behaves, in the second set of experiments, we do comparisons between JobOptimizer and other two baseline solutions (linear ascent and random selection), by observing their performance within a fixed number of iterations.

We use three groups of random-generated DAGs with different size as our evaluation applications. These three groups are named as small-, medium- and large-size group respectively. Again these DAGs are generated from package 'random-dag' as the same in Subsection 4.2.3. Each group consists of three DAGs which share the same parameters used for their creation in 'random-dag'. The parameters used for DAG creation are specified in Table 5.4. The meaning of these parameters have already been introduced in Subsection 4.2.3. Note that we use source processors to generate streaming data without data source from outside. Hence we exclude source processors when optimizing parallelization, as we assume the input streaming is stable.

Note in those applications, we build sources which produce messages by themselves. These artificial sources are not able to reply messages. In these applications the at-least-

once message delivery is not guaranteed, since missing messages can not be resent from sources. However, the mechanism of Clock Service gives us a way to manually check if there exists message missing. The global clock will not advance if there occurs message missing in the system, which can be seen from log files. We paid attention to the log files of these applications, and found that the clock service works smoothly, hence we know that there is no message loss in our examples. And furthermore, as we discussed in Restart approach, the processors of these applications are stateless.

Table 5.4: Parameters used for DAG creation in random-dag package

DAG group	max per rank	min per rank	max_ranks	min_ranks	probability
Small-size group	4	1	3	2	0.2
Medium-size group	7	2	5	3	0.15
Large-size group	10	3	7	4	0.10

Table 5.5: Configuration of JobOptimizer used in evaluation applications (RSD=relative standard deviation)

Parameter name	Value
number of random trials	3
maximum number of trials	100
threshold for RSD of trials	0.05
threshold for RSD of objective value	0.05
length of queue storing recent trials	5

The configuration of JobOptimizer used in our evaluation applications is shown in Table 5.5. All applications run in the same cluster environment as described in Subsection 4.2.3. As optimization is a probabilistic process, we run each application 3 times and select the best results to report.

5.3.1 Optimization Capability and Behavior Analysis

In Chapter 2 we mentioned that Fischer et al. (2015) run BO with a fixed number of iterations and select the best performance among those iterations. There are two drawbacks about this approach: first we do not know if the selected parameter set is globally optimal, second we always have to run the defined number of iterations, even though we may already find the optimal value in previous iterations. Our convergence criteria mitigate these two drawbacks. In theory the black-box function is unknown and we will never know if a specific value is globally optimal. However, in our case we define that some combinations of parameters which are close to each other and generate close performance are the globally optimal values we are looking for. That is reasonable in a sense that if BO always gives similar suggestions, there is no better possible choice in our optimization process. When our defined convergence is achieved, JobOptimizer

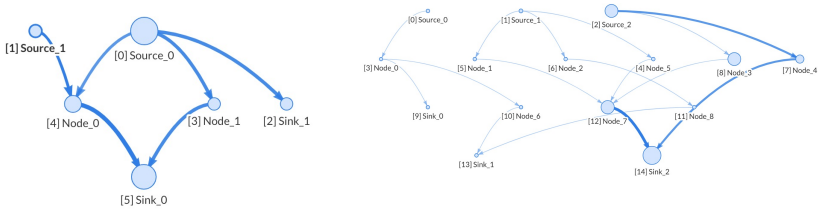
can certainly stop the optimization process before maximum number of iterations are reached.

In this first set of experiments, our goal is to analyse JobOptimizer's behavior, and test whether JobOptimizer can find the globally optimal value under the constraint of maximum number of trial. Each iteration of the optimization consists of the DAG transition stage (about 5 seconds) and the metrics gathering stage (1 minute), which takes around 1 minutes 5 seconds. As we treat the degree of parallelization of each processor as one parameter, the parameter space of optimization arises exponentially as the DAG size increases. In order to make it possible to find the optimal parameter set in reasonable time and also within the maximum number of trials (100), we limit the maximum degree of parrallelization to be 5 in this set of experiments. Note that even with this limitation, the parameter space is still large. For example, the parameter space for a very small DAG with 4 processors is $5^4 = 625$. So it is still a challenging job to find an optimal parameter set within 100 iterations.

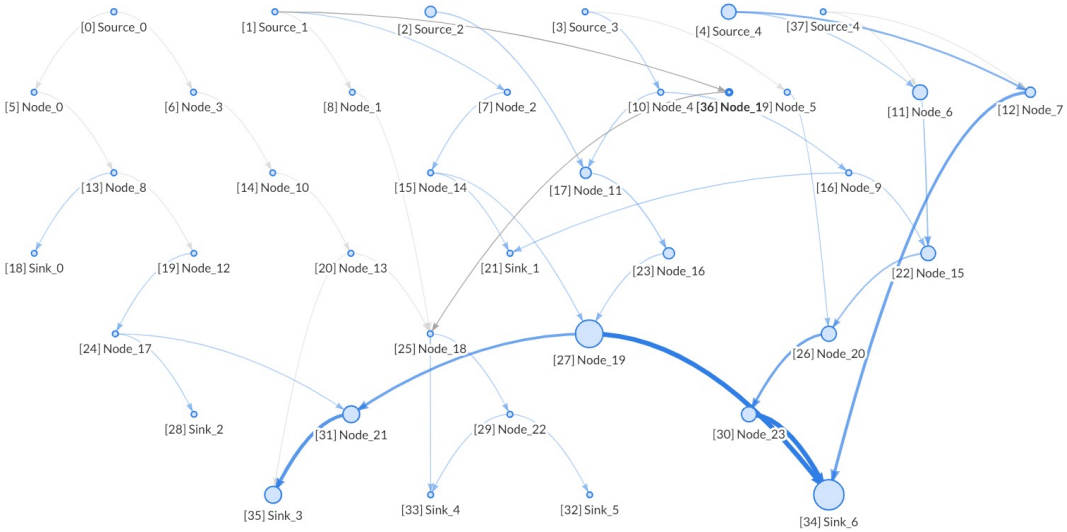
Table 5.6: Number of iterations needed for convergence in 9 test applications

DAG group	DAG Nr.	DAG Size	Iterations needed	Average by group
Small-size	1	7	18	19.0
	2	6	12	
	3	7	27	
Medium-size	4	15	65	74.5
	5	17	>100	
	6	17	84	
Large-size	7	36	>100	/
	8	40	>100	
	9	35	>100	

We run 9 applications (3 for each group, assigned with number from 1 to 9) and record the number of iterations they need to converge. This number is the measurement of convergence speed. The bigger the number is, the slower convergence speed is. The result is shown in Table 5.6. As we can see, all applications belonging to Small-size group are able to converge within the defined maximum iterations which is 100, with average number of iterations of 19.0. Two out of three applications in Medium-size group are able to converge, with on average 74.5 iterations needed. The numbers of needed iterations have large variance, indicating the optimization process is highly probabilistic and depends heavily on specific applications. Furthermore, comparing Small group and Medium Group, the number of iterations grows quickly as the DAG size increases, since the dimension of parameter space arises exponentially. No application in the Large-size group is able to converge within 100 iterations. Indeed Bayesian Optimization in high dimensional space is identified by the research community as one of the most urgent and challenging problem to be solved (Wang et al., 2013; Djolonga et al., 2013). To ensure that a global optimum is found, we require good coverage of the parameter space, but as the dimension increases, the number of evaluations needed to cover the parameter space



(a) Topologies of DAG Nr.2 (left) and Nr.4 (right)



(b) Topology of DAG Nr.7

Figure 5.7: Topologies of DAG Nr.2(Small), Nr.4(Medium) and Nr.7(Large)

increases exponentially.

We select one application from each group to inspect their behaviors. For Small and Medium group, we choose Nr.2 and Nr.4 as they converge most quickly within their respective group. For Large group, we randomly select Nr.7 as a representative. The DAG topologies of these three application are depicted in Figure 5.7. The small DAG only has 6 processors, in which 2 are sources. So the degrees of parallelization of other 4 processors (3 on the second layer and 1 sink on the third layer) is optimized. Similarly, the medium and large DAG have 15 and 36 processors, with 3 and 6 sources respectively. Hence 12 processors of the medium DAG, and 30 processors of the large DAG, are tuned.

The Web UI screenshot of trials information for tuning the small DAG is given in Figure 5.8. The algorithm first conducts 3 random tries and converges within only 12 iterations. From the 8th iteration, Advisor always suggest the same combination of parallelization. As we define the length of queue for storing recent trials as 5 (Table 5.5), JobOptimizer compute the relative standard deviation (RSD) of both 5 recently suggested parameter sets and 5 recent objective values, respectively. Both computed

Id	Study Name	Name	Parameter Values	Objective Value	Status
1	Gearpump_study	RandomSearchTrial	{"processor4": 5, "processor5": 4, "processor2": 2, "processor3": 2}	396910.638669	Completed
2	Gearpump_study	RandomSearchTrial	{"processor4": 1, "processor5": 3, "processor2": 2, "processor3": 2}	260502.080873	Completed
3	Gearpump_study	RandomSearchTrial	{"processor4": 3, "processor5": 2, "processor2": 3, "processor3": 3}	409599.932538	Completed
4	Gearpump_study	BayesianOptimizationTrial	{"processor4": 4, "processor5": 2, "processor2": 2, "processor3": 2}	413162.134752	Completed
5	Gearpump_study	BayesianOptimizationTrial	{"processor4": 4, "processor5": 2, "processor2": 2, "processor3": 2}	413253.319275	Completed
6	Gearpump_study	BayesianOptimizationTrial	{"processor4": 4, "processor5": 2, "processor2": 2, "processor3": 2}	411004.86425	Completed
7	Gearpump_study	BayesianOptimizationTrial	{"processor4": 4, "processor5": 2, "processor2": 2, "processor3": 2}	417978.675699	Completed
8	Gearpump_study	BayesianOptimizationTrial	{"processor4": 3, "processor5": 2, "processor2": 2, "processor3": 2}	423291.459123	Completed
9	Gearpump_study	BayesianOptimizationTrial	{"processor4": 3, "processor5": 2, "processor2": 2, "processor3": 2}	426657.056552	Completed
10	Gearpump_study	BayesianOptimizationTrial	{"processor4": 3, "processor5": 2, "processor2": 2, "processor3": 2}	418534.978002	Completed
11	Gearpump_study	BayesianOptimizationTrial	{"processor4": 3, "processor5": 2, "processor2": 2, "processor3": 2}	424903.531209	Completed
12	Gearpump_study	BayesianOptimizationTrial	{"processor4": 3, "processor5": 2, "processor2": 2, "processor3": 2}	418837.248308	Completed

Figure 5.8: The screenshot of trial list for small DAG example Nr.2

results are lower than predefined thresholds (0.05 as in Table 5.5). Hence JobOptimizer determines the convergence occurs and terminates the optimization process. The optimized application then runs with the globally optimal degree of parallelization, which is $\{processor2 : 2, processor3 : 3, processor4 : 3, processor5 : 2\}$ in this case.

To interpret how algorithm behaves in medium and large DAG case, we extract trials information from Advisor Service and depict the evolution of both objective values, and the RSD of parameter sets over number of iterations. The objective values reflect the throughput performance of Gearpump applications, and the RSD of parameter sets indicates how close the suggested points are to each other. These plottings shown in Figure 5.9 tells us how these two measurements change over time. In both plottings, the red lines represent the evolution of objective value measured by the left y axis, while the blue lines represent the change of RSD measured by the right y axis.

The optimization process starts with 3 random trials to get some initial information about the black-box function, represented by peaks of the RSD (blue) lines at the beginning. As the parameter spaces are large (5^{12} for medium DAG and 5^{30} for large DAG), 3 randomly tried parameter sets are far away from each other with high probability, which leads to the peaks of RSD lines in both depicts. Then the RSD lines sharply drop down as the algorithm tries to pay attention to exploitation, which means the algorithm will not

randomly guess any more and tries to guess on the basis of current information instead. Both RSD lines fluctuate dramatically. The local small peaks reflect the algorithm tries to explore the points a bit far away from current trials in order to explore. However, there is a decreasing tendency of the RSD line in the medium DAG example case, which indicates the algorithm lowers down the scope where the globally optimal value locates. At 65th iteration the line drops below 0.05 which is our defined threshold. In the large DAG case, the RSD does not experience a decreasing trend, since the algorithm needs more iterations to explore and get a better understanding of the parameter space.

The fact that RSD drops under 0.05 is not enough to trigger the detection of convergence. According to our rules shown in Figure 5.6, the RSD of objective values should also be under the threshold. In the medium DAG example, the objective value line (red) arises quickly as the number of iterations goes up, indicating the algorithm fits a high quality Gaussian Process which is close to the real black-box function we are exploring. The arise slows down after around 30 iterations, and the variance of objective values also decrease. Finally at 65th iteration, the computed RSD of objective value drops down under 0.05. In combination with that the RSD of suggested points is also under the threshold, JobOptimizer determines convergence occurs, and the application runs on the best parameter set chosen from latest 5 trials. Opposed to the objective value of medium DAG application which experiences a quick arise, the objective value of large DAG drops down after 40th iteration. In addition, the variance of objective value line even becomes larger after 100 iterations. Same as what RSD line tells us, the algorithm is still in the process of heavy exploration of the parameter space in the large DAG case.

5.3.2 Comparison Experiments

In this second set of experiments, we are going to construct two baselines for the purpose of evaluating how good or bad JobOptimizer behaves. We run a fixed number of iterations of 100 as specified in Table 5.5. Two metrics are extracted from the trials information: the highest objective value which indicates the best throughput performance enabled by the corresponding optimizer, and the number of iterations the optimizer needs to achieve the highest objective value, which demonstrates how fast the optimizer can find the best solution.

The first baseline optimizer is named as Linear Ascent Optimizer (LAO) which increases the degree of parallelization of each processor synchronously. At the beginning the degree of parallelization of all processors will be initialized to 1, which then is increased by 1 in every iteration. Therefore the dimension of parameter space is only 1, as all processors share the same degree of parallelization. Obviously size of the parameter space is the number of iterations we run, which is 100. The second baseline optimizer is called Random Exploration Optimizer (REO), whose strategy is exploring the parameter space by determining every trial randomly without any guidance from algorithm. Different from LAO, REO share the same dimension and size of parameter space with JobOptimizer.

We are going to use the same set of topologies as in last section, which consists of Small, Medium and Large-size groups with 3 applications in each group. In order

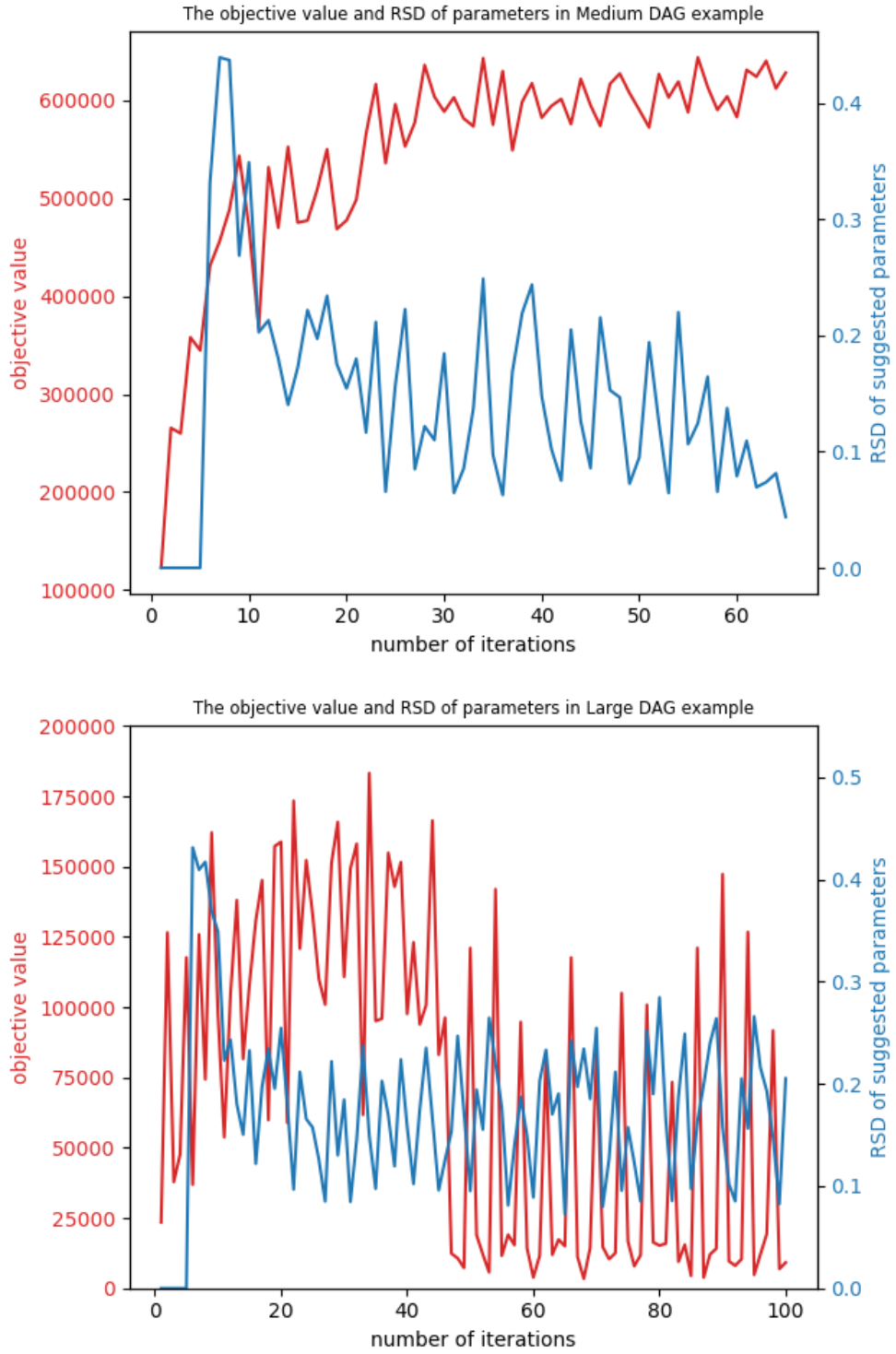


Figure 5.9: The objective value and RSD of parameters for DAG Nr.4(Medium) and Nr.7(Large)

to observe convergence within reasonable number of iterations, we limit the maximum degree of parallelization for each process to be 5 in last section. In addition to this set of experiments, we conduct another set of experiments with JobOptimizer by extending the limitation of parallelization to 10. This increases the parameter significantly. The limitation for REO is also set to 10. Each application run 3 times and the best result is taken.

In summary, we have four sets of experiments for comparison: JobOptimizer with 5 as the limitation of parallelization, JobOptimizer with 10 as the limitation of parallelization, Linear Ascent Optimizer (LAO) and Random Exploration Optimizer (REO). These optimizers are all applied to optimize 9 applications described in last section and run for 100 iterations. The highest objective value and number of iterations needed to achieve this value averaged by group are depicted as bar charts shown in Figure 5.10.

As we can observe from the upper chart in Figure 5.10, REO behaves the worst among 4 optimizers, with lower objective value and much larger variance than any other optimizers, which indicates that the random determination of parameters is not a good optimization strategy. When applied to small DAG applications, JobOptimizers with both 5 and 10 as the limitation of parallelization are not able to outperform LAO. While in the Medium and Large group, both JobOptimizers can find better solutions than LAO which lead to higher objective values (higher throughput). LAO applies an 'one-fits-all' strategy, which sets the same degree of parallelization for every processor of the DAG. According to the chart, this strategy works well for applications in the Small group, since LAO gives similar performance to JobOptimizers do. However, when the topology gets more complex, setting homogeneous degrees of parallelization is no longer preferred. JobOptimizers which tun the parallelism of each processor individually bring higher throughput than LAO in both Medium and Large groups. Furthermore, even though increasing the limitation of parallelization from 5 to 10 enables JobOptimizer to explore in a much larger parameter space, JobOptimizer_5 perform a bit better than JobOptimizer_10 in all three groups. This fact demonstrates that, with the constraint of fixed number of iterations, enlarging the exploration space does not necessarily help us get better solutions.

The second chart reflects the speed of optimizers to find the best parameter sets. For simplicity purpose, We simplify the term 'the number of iterations needed to achieve the best objective value' as 'number of iterations'. We can see that enlarging the exploration space increases the number of iterations significantly. JobOptimizer_10 and REO have the same size of parameter space, which is 10 powered by the number of processors to be tuned. Therefore both of them take much more iterations than JobOptimizer_5 and LAO to find the best parameter set. REO takes less iterations than JobOptimizer_10 on average, while with much larger variance. LAO is the fastest Optimizer in terms of determining the best parameter set. With the limitation of computational resources, it is unlikely that increasing the degree of parallelization of every processor to a large number (for example 50) generates better results than comparatively small number. This is proved by the fact that the average number of iterations needed by LAO in all three groups are under 20, which means the best homogeneous degrees of parallelization are all below 20. JobOptimizer_5 needs less number of iterations than REO but more than LAO.

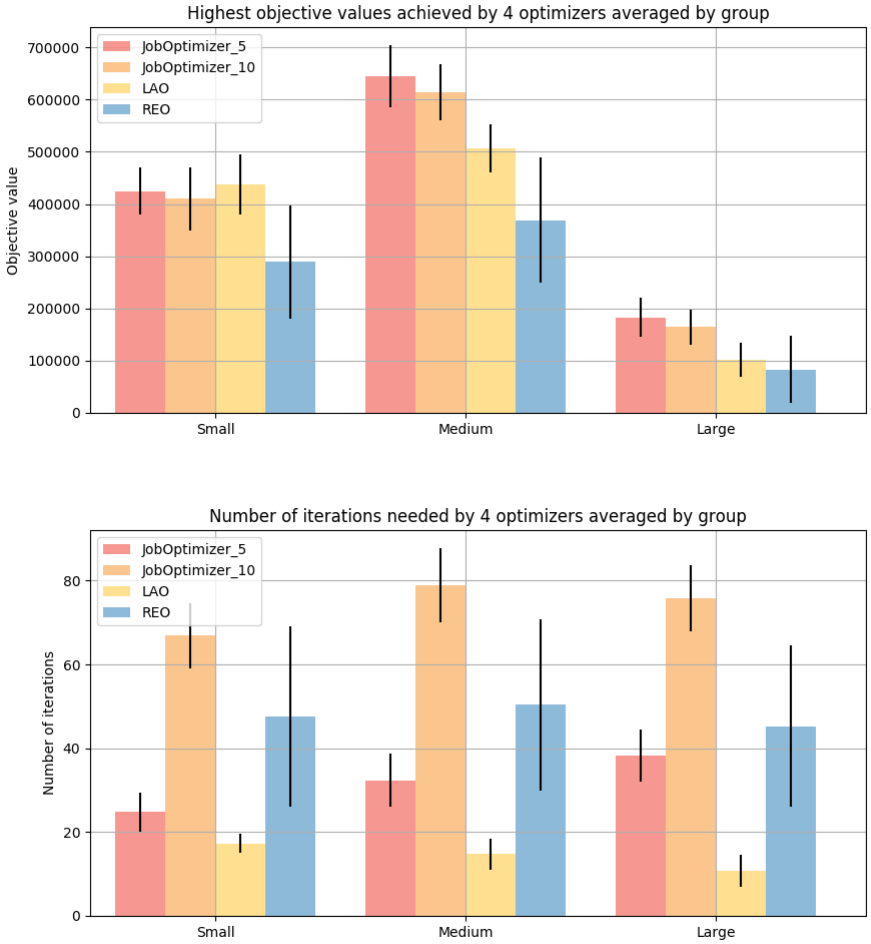


Figure 5.10: The comparison of highest objective values and number of iterations needed among 4 optimizers averaged by group (LAO = Linear Ascent Optimizer, REO = Random Exploration Optimizer)

In summary, as the answer to *RQ3*, with our designed convergence criteria JobOptimizer is able to determine the globally optimal parameter set for small and a proportion of medium-sized DAG applications. For larger DAG applications, convergence is not guaranteed within the predefined maximum number of trials, because Bayesian Optimization in high-dimensional space is not effective. Despite of that, JobOptimizer is capable of finding better parameter sets within less number of iterations than Random Exploration Optimizer (REO). Although Linear Ascent Optimizer (LAO) takes less iterations to determine the best parameter set, JobOptimizer can find better solutions in the case of Medium and Large DAG applications. Furthermore, limiting the maximum degree of parallelization can help find better solution more efficiently (less iterations) as it decreasing the exploration space greatly.

6

Limitations

We identify a few limitations in our work. Firstly, although Bayesian Optimization does not require the objective function to be smooth, the assumption of applying BO is that the targeted black-box function is continuous. If the objective function is not continuous, BO cannot work properly to deduce the objective value. But continuity may not always be the case when configuring the parallelism of processors in streaming applications. To what extent this adversely influenced our experiment results needs to be further studied.

Secondly, Bayesian Optimization is proved to be not efficient in high dimensional space (Wang et al., 2013; Djolonga et al., 2013). This is because BO requires a good coverage of the parameter space to ensure that a global optimum is found. However as the dimension increases, the number of evaluations needed to cover the parameter space increases exponentially. As we can see from Section 5.3, JobOptimizer is not able to converge when applying to comparatively large DAG applications. Within the constraint of time and number of iterations, JobOptimizer cannot explore the parameter space adequately.

Lastly, our focus is on tuning the parameters of streaming applications at runtime with the help of Bayesian Optimization. However, BO algorithm itself has hyperparameters, including the choice of Gaussian Process kernels. We just use the default squared-exponential kernel. Our experiment performance may vary with different BO hyperparameter settings. Despite of this, the conclusion we draw about the capability of BO at runtime remains the same, which is BO is able to find better parameter sets within less iterations than random exploration, and can find better parameter sets than Linear Ascent Optimizer in the case of large DAG applications.

7

Future Work

Given the problem of inefficiency of BO working in high dimensional space, our first interest in future work is investigating BO in high dimension. A few research teams are trying to improve the performance of BO in high dimension. For example, Wang et al. (2013) proposed an algorithm named Random Embedding Bayesian Optimization (REMBO), which tries to lower down the dimension of exploration space with the help of a random matrix. If BO in high dimension can be made more efficient, the application of BO to large DAG applications will not be limited. We will inspect currently proposed solutions and figure out if they can be helpful in our runtime Optimization case.

Secondly, Restart approach stops all old processors completely, the intermediate processor states will lose. It works well in the case of applications with stateless processors. However, many real life applications are with stateful processors. In future work we intend to find out the way to save and recover processor states. In that way can we make Restart approach suitable for stateful processors.

Thirdly, we focus on tuning the parallelism of Gearpump application processors in this work, in future work we would like to generalize this approach to tuning other configuration options, both global system configuration and application-specific configuration. The modification of other parameters will also be supported by runtime operation capability. In addition, we will explore the possibility of implementing runtime operation in other streaming engines so runtime optimization may be supported in these systems in the future.

Lastly, JobOptimizer will only optimize the parameter when an application is submitted, under the assumption that the streaming input is static. However, in many use cases the input of the streaming application changes over time. Hence we would like to enable dynamic adjustment of parameters during runtime, by monitoring the input workload and launch the optimization process repeatedly and dynamically.

8

Conclusions

The problem of parameter configuration in the field of distributed (streaming) systems is a popular research area and has been studied over years. Many solutions have been proposed by the research and industry community, among which Bayesian Optimization (BO) is proved to be powerful. While the existing way to conduct the BO process is 'offline' and involves shutting down the system as well as many inefficient manual steps, we design and implement an optimizer for Apache Gearpump which is able to do 'online' optimization.

The DAG operation at runtime is the prerequisite for doing 'online' optimization. In Chapter 4, after inspection into 'Dynamic DAG' feature of Gearpump, we found it is not competent enough to support our optimization process, as it is resource-consuming and only able to modify a single processor each time. Inspired by 'Dynamic DAG', we developed Restart approach which overcomes the drawbacks of 'Dynamic DAG' (*RQ1* is therefore answered). The strategy of Restart is stopping the application and recovering it with the new DAG and old application states. By stopping the old application and deploying a new one, we reclaim resources and allocate again which avoids launching new executors. Restart approach is specifically designed for applications with stateless processors. In the case of stateful processors, we need to figure out how to save and recover processor states in future work. Furthermore, Restart enables the recovery of arbitrarily-modified DAG by storing it to Master cluster. The comparison experiment indicates that Restart is superior to the baseline approach Dynamic Update in terms of transition time and throughput performance.

In Chapter 5, we describe the design and implementation of our JobOptimizer, which is the answer to *RQ2*. In order to avoid introducing complex dependency tree and keep the source code clean, we decide to adopt a loose-coupling architecture which makes the implementation of BO as an external service. JobOptimzier consists of three modules: Client which is responsible for communicating with external BO service, DAG Operations which implements the Restart approach for doing runtime DAG operation, and Metrics Manipulation which gathers metrics from Gearpump system and compute the objective value. Stop criteria is also designed for JobOptimizer to determine when to stop the optimization process.

To evaluate JobOptimizer as well as answering *RQ3*, we conduct a series of experiments in Section 5.3. We try to analyse the optimization capability and the behavior of JobOptimizer in the first set of experiments. We found that JobOptimizer is able to

find the global optimum for small and a proportion of medium-sized DAG applications. Since Bayesian Optimization in high-dimensional space is not effective, convergence is not guaranteed within a defined number of trials in the case of large DAG applications. From the comparison experiments, we noticed that JobOptimizer is capable of finding better parameter sets within less number of iterations than Random Exploration Optimizer (REO). In addition, JobOptimizer can also find better solutions than Linear Ascent Optimizer (LAO) in the case of Medium and Large DAG applications.

Lastly, a few limitations have been mentioned in Chapter 6. For instance the effect of the continuity assumption of BO needs to be further studied. Our potential future work is briefly discussed in Chapter 7. How to improve the performance of BO in high dimensional space is one of our next research problem.

References

Akidau, T. (2015). Streaming 101: The world beyond batch. https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101.

Babu, S. (2010). Towards automatic optimization of mapreduce programs. *Proceedings of the 1st ACM symposium on Cloud computing*, pages 137–142.

Bergstra, J., Yamins, D., and Cox, D. D. (2013). Making a science of model search: hyperparameter optimization in hundreds of dimensions for vision architectures. *Proceedings of the 30th International Conference on International Conference on Machine Learning*, 28:115–123.

Brochu, E., Cora, V. M., and de Freitas, N. (2010). A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *Cornell University, arXiv:1012.2599*.

Cammert, M., Kramer, J., Seeger, B., and Vaupel, S. (2008). A cost-based approach to adaptive resource management in data stream systems. *IEEE Transactions on Knowledge and Data Engineering*, 20:230–245.

Cox, M. and Ellsworth, D. (1997). Application-controlled demand paging for out-of-core visualization. *VIS '97 Proceedings of the 8th conference on Visualization'*, pages 235–ff.

Daum, M., Lauterwald, F., Baumgaertel, P., Pollner, N., and Meyer-Wegener, K. (2011). Black-box determination of cost models' parameters for federated stream-processing systems. *Proceedings of the 15th Symposium on International Database Engineering & Applications*, pages 226–232.

Dean, J. and Ghemawat, S. (2004). Mapreduce: Simplified data processing on large clusters. *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150.

Djolonga, J., Krause, A., and Cevher, V. (2013). High-dimensional gaussian process bandits. *NIPS'13 Proceedings of the 26th International Conference on Neural Information Processing Systems*, pages 1025–1033.

Fischer, L., Gao, S., and Bernstein, A. (2015). Machines tuning machines: Configuring distributed stream processors with bayesian optimization. *IEEE International Conference on Cluster Computing*, pages 22–31.

Ghemawat, S., Gobioff, H., and Leung, S.-T. (2003). The google file system. *SOSP '03 Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43.

Golovin, D., Solnik, B., Moitra, S., Kochanski, G., Karro, J., and Sculley, D. (2017). Gaussian process optimization in the bandit setting: No regret and experimental design. *23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1487–1495.

Gordon, M. I., Thies, W., and Amarasinghe, S. (2006). Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 151–162.

Heinze, T., Jerzak, Z., Hackenbroich, G., and Fetzer, C. (2014). Latency-aware elastic scaling for distributed data stream processing systems. *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, pages 13–22.

Jamshidi, P. and Casale, G. (2016). An uncertainty-aware approach to optimal configuration of stream processing systems. *IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 39–48.

Kushner, H. J. (1964). A new method of locating the maximum of an arbitrary multipeak curve in the presence of noise. *Basic Engineering*, 86:97–106.

Laney, D. (2001). 3d data management: Controlling data volume, velocity, and variety. http://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf.

Marr, B. (2015). *Big Data: Using SMART Big Data, Analytics and Metrics To Make Better Decisions and Improve Performance*. Wiley.

Maxim, B., Heisel, M., Ali, N., Bahsoon, R., and Mistrik, I. (2017). *Software Architecture for Big Data and the Cloud*. Morgan Kaufmann.

Mockus, J. (1974). On bayesian methods for seeking the extremum. *Optimization Techniques IFIP Technical Conference Novosibirsk*, 27:400–404.

Mockus, J., Tiesis, V., and Zilinskas, A. (1978). The application of bayesian methods for seeking the extremum. *Toward Global Optimization*, 2:117–129.

Schneider, S., Andrade, H., Gedik, B., Biem, A., and Wu, K.-L. (2009). Elastic scaling of data parallel operators in stream processing. *IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12.

Schneider, S., Hirzel, M., Gedik, B., and Wu, K.-L. (2012). Auto-parallelizing stateful distributed streaming applications. *21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 19–23.

Sherman, J. and Morrison, W. J. (1950). Adjustment of an inverse matrix corresponding to a change in one element of a given matrix. *The Annals of Mathematical Statistics*, 21:124–127.

Shvachko, K., Kuang, H., Radia, S., and Chansler, R. (2010). The hadoop distributed file system. *IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10.

Snoek, J., Larochelle, H., and Adams, R. P. (2012). Practical bayesian optimization of machine learning algorithms. *Proceedings of the 25th International Conference on Neural Information Processing Systems*, 2:2951–2959.

Srinivas, N., Krause, A., Kakade, S. M., and Seeger, M. (2010). Gaussian process optimization in the bandit setting: No regret and experimental design. *27th International Conference on Machine Learning*, pages 1015–1022.

Thompson, W. R. (1933). On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25:285–294.

Wang, Z., Zoghi, M., Hutter, F., Matheson, D., and Freitas, N. D. (2013). Bayesian optimization in high dimensions via random embeddings. *IJCAI '13 Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, pages 1778–1784.

Wu, D. and Gokhale, A. (2013). A self-tuning system based on application profiling and performance analysis for optimizing hadoop mapreduce cluster configuration. *20th Annual International Conference on High Performance Computing*, pages 89–98.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: cluster computing with working sets. *2nd USENIX conference on Hot topics in cloud computing*, pages 10–10.

Zhong, S. (2014). Gearpump: Real-time streaming engine using akka. http://downloads.typesafe.com/website/presentations/GearPump_Final.pdf.

A

Appendix

In the Appendix we report some bugs we found in Apache Gearpump during our work.

A.1 The misuse of metrics in Web UI

When we were exploring the Dynamic DAG feature of Gearpump, we run an example application and in the Web UI we changed the parallelism of two processors, which are processor0 and processor8. Then we monitored the behavior of the application after we changed the parallelism. However, we found from the topology dashboard that: the old processors which are supposed to be replaced and stopped were still working. The screenshot of this bug is shown in Figure A.1. We changed the parallelism of processor0, the system start a new executor and launch a processor (processor12) with the updated parallelism. Then processor0 should stop working and completely replaced by processor12. However processor0 is still working as shown in Figure A.1.

The reason for this bug is the misuse of metrics in Web UI. We checked the metrics data in JSON format fetched by Gearpump Web UI as shown in Figure A.2. Gearpump Web UI use *m1* which is one-minute exponentially-weighted moving average of throughput instead of the instantaneous value to report metrics in the dashboard. Hence the reported throughput is smooth, even when processor0 has been stopped, its throughput will not drop down to 0 sharply, as shown in Figure A.3. That is the reason why we observe processor0 still working after it should be stopped and replaced.

A.2 Clock Service bug

The Clock Service tracks the minimum timestamp of all pending messages in the system and maintains a global view of minimum clock. Every task updates its local minimum clock to the Clock Service. The Clock Service is very important to Gearpump system as it helps detect and recover missing messages and ensure at-least-once message delivery. The global clock maintained by Clock Service is determined by the minimum clock of pending messages, outgoing messages and the minimum clock of each task.

During the Dynamic DAG experiment we found the Clock Service stop moving forward and got stuck at the infinite negative value (-9223372036854775808). In order to find

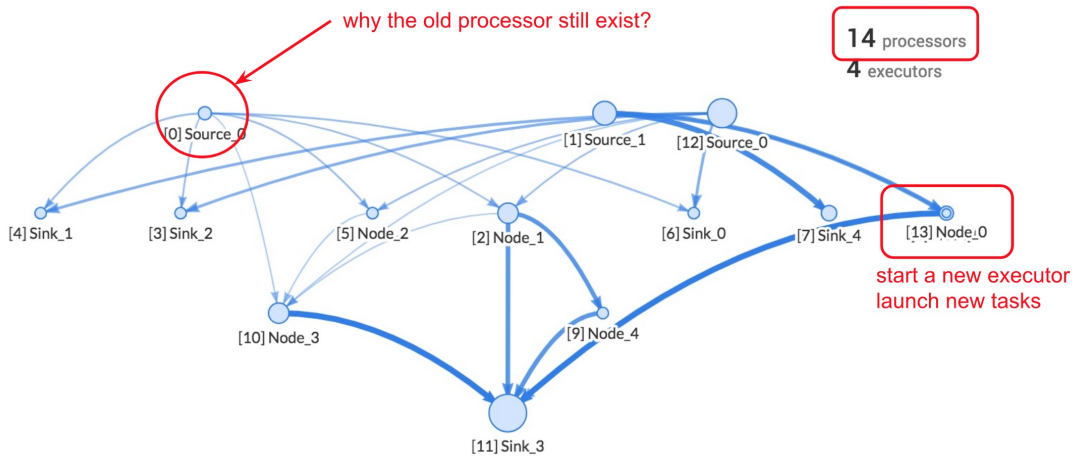


Figure A.1: The screenshot of bug within topology dashboard


```
time: "1539810772933"
value: {$type: "org.apache.gearpump.metrics.Metrics.Meter", name: "app1.processor0:sendThroughput",…}
  $type: "org.apache.gearpump.metrics.Metrics.Meter"
  count: "6912210"
  m1: 51583.04818421861
  meanRate: 63701.64695515128
  name: "app1.processor0:sendThroughput"
  rateUnit: "events/s"
```


package com.codahale.metrics

m1: One-minute exponentially-weighted moving average

meanRate: mean rate of throughput all time(from beginning till now)

Figure A.2: The metrics data in JSON format fetched by Gearpump Web UI

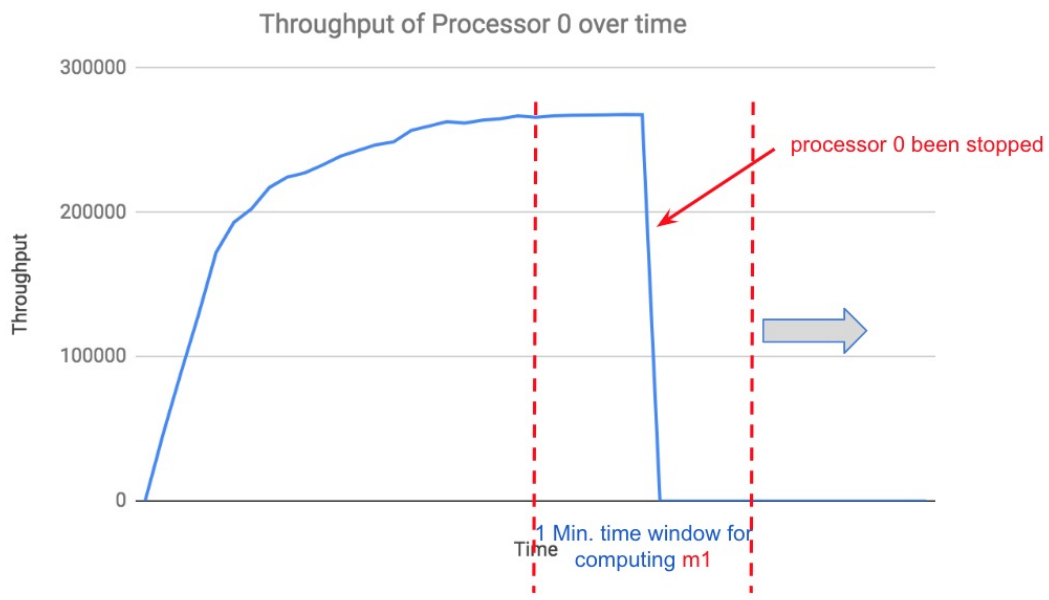


Figure A.3: The throughput of processor0 and the compute window of moving average

out the problem, we inspected the process of how Clock Service updates its watermark (global minimum clock), as shown in Figure A.4. Each upstream task actor who wants to send a message out will send it through an object named Subscription. A Subscription connects the upstream (publisher) and downstream task actor (subscriber) together. After Subscription passing a message with a watermark to subscriber, it will send a message requesting an acknowledgement from subscriber. The subscriber then replies with an acknowledgement message which confirming receiving the message with a specific watermark. The Subscription will update its own minimum clock by computing the minimum clock of all watermarks of acknowledged messages. Consequently the publisher will update its local minimum clock with the minimum clock of subscription. Finally the Clock Service will update the global clock with the minimum clock of publisher.

The problem occurs when we conduct a Dynamic DAG operation. When the replaced processor is stopped, the related subscription is not removed. When the processor which a subscription referred to is not available, a default minimum clock will be assigned to this missing processor which is negative infinity. The consequence is the local clock of this subscription is updated to negative infinity, which will then leads to the global clock being updated to negative infinity. Finally the Global Clock will stop moving forward. The key to solve this bug is simple: removing the subscription which involves replaced processors.

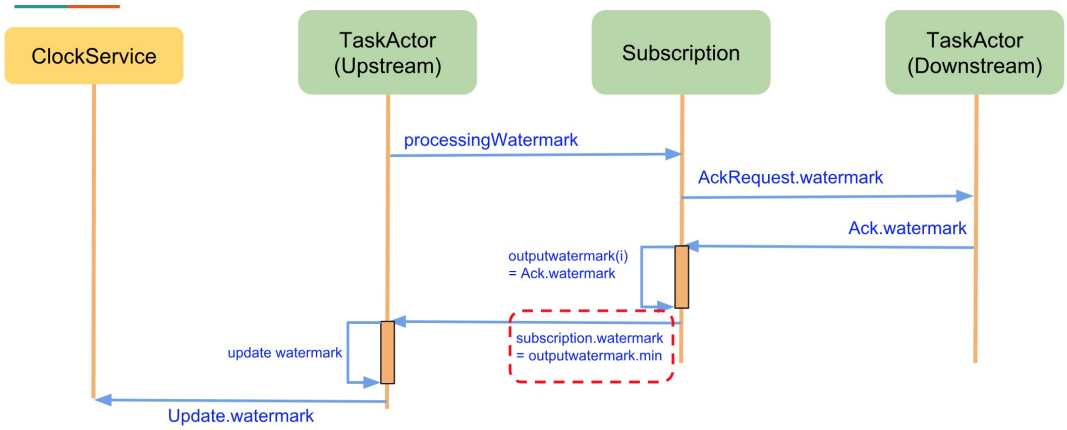


Figure A.4: The process of updating watermark for Clock Service

List of Figures

List of Tables