Scheduling Decisions in Stream Processing on Heterogeneous Clusters

Technical summary

Since all processors have identical performance and capacity in the homogeneous computing environments, resources allocation and task scheduling can be evenly determined across all available processors. However, there are different types of processors in heterogeneous clusters. The performance of the processors depends on the specific input data. Hence, it is very difficult to optimally arrange and distribute different type of the input data and incoming tasks with individual types of processors.

In order to achieve optimal performance on heterogeneous cluster, the authors propose a novel scheduling advisor based on the Apache Storm distributed stream processing platform. And the authors test their implementation by scheduling of the example use case application.

Description of contributions

In the paper, the authors summarize two key requirements for a scheduler to optimally balance workload in a heterogeneous cluster. In order to meet the requirements, the authors further introduce some important scheduling decisions in the stream processing, specifically *data locality*, *offline decisions* and *online decisions*. Finally, the authors propose a novel scheduler based on the Storm default scheduler.

Major critiques

In order to know performance characteristics for individual types of nodes, the authors devise an innovative scheduler on the basis of *offline decisions* and *online decisions*.

At the *offline decisions* phase, every processor will be measured on a specific cluster and benchmarked on each class of hardware nodes. After this phase, benchmarks comprise the performance data of nodes. Meanwhile, the authors introduce an existing method to analyze the computation characteristic of incoming tasks and input data. In their approach, tasks will be tagged according to their required computation resources. At the *online decisions* phase, the scheduler can utilize the data captured by the methods above. Moreover, the new scheduler can deploy some node with yet unknown performance. And these historical performance data will be updated with the reflection of processed data.

From the definition of *data locality*, processors generally operate on data streams from remote sources at first in order to minimize the communication cost. However, the optimal placement of tasks across cluster nodes in this paper depend on the characteristics of nodes and tasks beyond

the communication costs mentioned above. Hence, it is definitely possible that the authors cannot achieve optimal processing time with their scheduling method.

Minor points

In the paper, the authors include a commonly adopted way to address the issue about how to analyze characteristics of incoming tasks. However, they don’t propose their novel method to resolve this problem.

A Predictive Scheduling Framework for Fast and Distributed Stream Data Processing

Technical summary

The tuple processing time is a crucial performance metric for a scheduler of stream processing systems. It will be efficient to assign tasks to processors based on the average tuple processing time.

The authors devise a predictive scheduling framework based on Storm, which is able to accurately predict tuple processing time by an innovative topology-aware method and efficiently arrange tasks with the prediction time. And the authors evaluate their implementation with two representative applications, word count and log stream processing.

Description of contributions

Firstly, the authors propose a method for modeling the average tuple processing time of a typical application whose topology is a general graph.

Second, the authors devise a predictive scheduling algorithm, which can compute a scheduling solution based on the collected runtime data of threads and the average tuple transfer latency.

Third, the authors extend the Storm architecture by adding several modules, such as data collector, data store, data pre-processor, and so on.

Major critiques

In authors' model, for a given scheduling solution, the average tuple processing time of an application is the sum of the average tuple processing latency at Processor Units and the average tuple transfer latency between Processor Units. Meanwhile, the average tuple processing latency at each Processor Unit can be predicted by the average tuple processing latency of threads. And the average tuple transfer latency between Processor Units can be predicted by the average tuple transfer latency between threads.

In order to collect the statistics from multiple threads, the author propose a module *Data Collector*. The data of a tuple from a thread of a PU to a thread of the next PU will be recorded by the data collector with the unique tuple ID. And the data contains the tuple's processing and transfer latencies. Then the data will be sent to the distributed *Data Store* instantly.

Based on the statistics in the data store, the modules *Data Pre-processor* and *Performance Generator* adjust the scheduling solution and try to assign threads by using the predictive scheduling algorithm.

Minor points

Although the authors introduce a module *Data Collector* to collect the tuple's processing and transfer latencies within threads, they don't explain how the data collector records these statistics. The authors just list the set of features like Machine-CPU, which can help predicting the average tuple latencies of a thread. This paper is not detailed enough here.

Additionally, it is manifest that the proposed algorithm in this paper is advanced and useful. However, it requires a large amount of historical performance data to make optimal decision.

Distributed QoS-Aware Scheduling in Storm

Technical summary

The demand for processing data has been increasing, as the traditional method of storing and collecting information cannot deal with the large amount of data in real-time. Hence, the distributed stream processing systems like Storm are more and more necessary.

Additionally, a distributed stream processing system is usually modeled using a directed graph, in which the vertices are the Processing Unit, and edges indicate data flow. In order to minimize the cost of data transmission in the network, the authors design and implement a novel distributed QoS-aware scheduler for Storm. Then, the authors also evaluate the self-adaptation capabilities of their distributed QoS-aware scheduler by two experiments.

Description of contributions

The main contribution of this paper is two-fold.

For one thing, the authors introduce a new measurement model for operators which transforms the performance metrics into distance between operators. The distance called *cost space* in this paper, consists of three QoS attributes, specifically latency, utilization, and availability. And the authors devise a distributed scheduling algorithm with *cost space* to solve the operator placement problem.

For another, the authors add three key modules to the Storm architecture, specifically *QoSMonitor, AdaptiveScheduler, and WorkerMonitor.* The modules are responsible for information (*cost space*) collection and operator placement.

Major critiques

In order to efficiently solve the operator placement problem, the authors implement a distributed scheduling algorithm with the new metric *cost space*. In their operator placement algorithm, the distance of two operators varies with the amount of data transmission among them. This means that the *cost space* between two nodes should be smaller if there are larger amount of data transmission within them. Hence, the *cost space* will dynamically adapt to changing network and node conditions.

Additionally, it is very difficult to gauge *cost space* directly, which comprises latency, utilization, and availability. In this paper, the authors adopt a four-dimension space. Each node will have a coordinate associated to latency, utilization, and availability. Therefore, calculating the *cost space* between two nodes is same as calculating Euclidean distance.

Further, in order to avoid the exhaustive probing, the authors devise *QoSMonitor* and *WorkerMonitor*, which capture intra-node and inter-node information in real time. With the information collected by *QoSMonitor* and *WorkerMonitor*, the *AdaptiveScheduler* executes the distributed scheduling policy on every worker node periodically (every 30s).

From my perspective, there is also a little flaw in their implementation. Since the iteration of reassigning operators is executed periodically (every 30s), it is definitely possible that the data rate between two neighbor operators becomes smaller in next period. It turns out that the authors' method based on past statistics is optimal enough, but it will not work well at some times.

Minor points

In the layer of distributed schedulers, the authors propose an innovative method to achieve optimal operator scheduling. However, the authors retain the original centralized scheduler in storm, called *BootstrapScheduler*. I think it is also necessary to design a QoS-Aware centralized scheduler coordinated to the *AdaptiveScheduler*.

Efficient Queue Management for Cluster Scheduling

Technical summary

At the most of cases, the modern cluster schedulers cannot assign tasks to the best possible machine. Meanwhile, since default schedulers lack cluster-wide information, the globally optimal allocations across various queue managers are not possible.

In the paper, the authors extend a centralized (YARN) and a distributed (MECURY) cluster scheduling. And then, the authors offer new queue techniques at worker nodes with different task placement strategies. At the end of paper, the authors compare the extension with the default schedulers by using workloads derived from Microsoft clusters.

Description of contributions

The main contribution of this paper is as below.

Firstly, in order to address the utilization problem, the authors extend two cluster scheduler YARN and MECURY by adding tasks queue at worker nodes.

Second, the authors propose a guideline on how to bound the length of the queues at worker nodes. Meanwhile, they introduce task prioritization techniques like queue sizing and queue reordering, which are able to decrease the job completion times.

Third, the authors implement a novel algorithm to estimate queue wait time at worker nodes.

Finally, the authors investigate a novel strategy that takes account for job structure which can be used for prioritizing task execution.

Major critiques

Compared with existing systems that also enable queues at worker nodes, the authors propose a method to bound the queue size in order to optimally improve cluster utilization and decrease queuing delays. Moreover, in order to reduce head-of-line blocking, they propose another technique to reorder tasks in the queue. In addition, the authors initiate a default task duration at the beginning, and gradually refine it as the execution of the job proceeds. Because bounding queue lengths and prioritizing task execution rely on estimates of task durations.

For queue sizing, the authors introduce two approaches to bound the length of queues, length-based queue bounding and delay-based queue bounding. In the first approach, all nodes have the same fixed queue length. In opposite, delay-based strategy relies on the estimated queue wait

time to dynamically determine the queue length for each node. Hence, it will work well with heterogeneous tasks.

In order to reorder queue tasks, the authors use several reordering strategies, specifically Shortest Remaining Job First (SRJF), Least Remaining Tasks First (LRTF), Shortest Task First (STF), Earliest Job First (EJF) and the novel Starvation-aware Queue Reordering.

Minor points

Based on the statements in the ‘major critiques’ section, the task duration estimate plays an important role in bounding queue lengths and queue reordering. However, the method the authors used looks like naïve and easy.

Model-based Scheduling for Stream Processing Systems

Technical summary

Due to the huge volume of data flows generated from devices, the velocity of data ingesting and processing is one of big challenges for stream processing systems. Meanwhile, with the limitation of memory-storage and the extreme volume of data requests, throughput and latency have been regarded as the key concern of an efficient stream processing system.

The authors devise a model-based scheduling scheme, which dynamically react to the input data and optimally allocate resources. And the authors evaluate their model with throughput and latency by comparing with the generic scheduling scheme of Apache Storm.

Description of contributions

The main contribution of this paper is two-fold.

For one thing, the authors design two models, *throughput model* and *latency model*. The two models serve as the guideline for the schedule algorithm to allocate resources to individual component(operator).

For another, the authors inform an adaptive scheduling algorithm based on two system model introduced above.

Major critiques

The latency for a data tuple is the sum of the execution time and the waiting time in the queues. In order to easily estimate latency, the authors assume that the execution time for a data tuple will not change if the provisioned resource is not altered. For the waiting time, the authors use the Allen-Cunneen (A-C) formula to estimate the mean waiting time. Based on the previous statements, the authors create a model that can calculate the time of each tuple in a stream.

In *throughput model*, the authors collect the data about the number of tuples processed as the output at any time interval. With these data, the throughput model is able to monitor real-time workload and perform throughput estimation for each stream.

In addition, the authors classify the hosts into different groups depending on the provisioned resource of CPU and memory. With the classification, the adaptive scheduling algorithm is able to select the most appropriate hosting resources for components. Further, the algorithm can optimally allocate components over a distributed platform and keep the throughput consistent with the stream’s incoming traffic.

Conclusion

The authors devise a model-based scheduling to minimize the gap between the resource demanding and the provisioned resources by efficiently determining the ideal host for components. With the novel allocation optimization, it is definite that the waiting time of a data tuple will decrease.