12-2018

Scale-Out Algorithm For Apache Storm In SaaS Environment

Ravi Kiran Puttaswamy
University of Nebraska-Lincoln, ravikiran.mara@live.com

Scale-Out Algorithm For Apache Storm In SaaS Environment

by

Ravi Kiran Puttaswamy

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Ying Lu

Lincoln, Nebraska

December, 2018

Scale-Out Algorithm For Apache Storm In SaaS Environment

Ravi Kiran Puttaswamy, M.S.

University of Nebraska, 2018

Adviser: Ying Lu

The main appeal of the Cloud is in its cost effective and flexible access to computing power. Apache Storm is a data processing framework used to process streaming data. In our work we explore the possibility of offering Apache Storm as a software service. Further, we take advantage of the cgroups feature in Storm to divide the computing power of worker machine into smaller units to be offered to users. We predict that the compute bounds placed on the cgroups could be used to approximate the state of the workflow. We discuss the limitations of the current schedulers in facilitating this type of approximation as the resources are distributed in arbitrary ways. We implement a new custom scheduler that allows the user with more explicit control over the way resources are distributed to components in the workflow. We further build a simple model to approximate the current state and also predict the future state of the workflow due to changes in resource allocation. We propose a scale-out algorithm to increase the throughput of the workflow. We use the predictive model to measure the effects of many candidate allocations before choosing it. Our approach analyzes the strengths and drawbacks of Stela algorithm and design a complementary algorithm. We show that the combined algorithm complement each others strengths and drawbacks and provides allocations to maximize throughput for much larger set of scenarios. We implement the algorithm as a stand alone scheduler and evaluate the strategy through physical simulation on the Apache Storm Cluster and on software simulations for a set of workflows.

ACKNOWLEDGMENTS

Foremost, I would like to express my sincere gratitude to my advisor Prof. Ying Lu for the continuous support of my study and research, for her patience, motivation, enthusiasm, and immense knowledge. Her guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my Masters Thesis.

Besides my advisor, I would like to thank the rest of my thesis committee: Prof. David Swanson and Prof. Hongfeng Yu, for their encouragement and insightful comments.

I would also like to thank the CSE department and HCC for providing ample amount of support and resources to help me complete my work.

Finally I would like to my family and friends, without whom I wouldn't be here today.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cloud computing technology has received widespread acceptance in the recent years. Companies are moving their assets and workloads to the cloud through mixed or pure cloud deployments in ever increasing numbers.[2] The cloud offers a lot of flexibility by allowing the user to procure resources on demand, which reduces operational costs. Cloud computing is offered in many flavors, to better suit the user's perception of the infrastructure. Some of the popular flavors are Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (Saas). In IaaS, the computing resources and the supporting infrastructure like networking, storage etc. are owned by the cloud providers. The user can procure these resources on demand. This flavor provides the greatest freedom, as the user can customize the infrastructure and the software. PaaS offers the middleware with the infrastructure. Middleware typically includes the operating system, development tools, database management system, etc. The user typically deploys his application using the tools provided by the middleware. SaaS offers a complete software solution as a service. The user can connect to the software and use it over the internet. The applications are deployed and maintained by the cloud provider or a third party agent, and the user has no access or control over the infrastructure that hosts the service.

One of the technologies to take advantage of this form of computing is Big Data. Cloud computing along with the recent advancements to the data processing frameworks has greatly reduced the barrier of entry to many medium and small sized companies. Apache Hadoop,[3] Apache Storm,[4] Apache Spark,[5] Heron,[6] Milwheel,[7] Apache Zookeeper,[8] S4[9] are among the more popular frameworks used for big data processing. Creating and maintaining a data center is expensive and often untenable for small companies. Instead now they can take advantage of the infrastructure provided by the cloud. The cloud and the frameworks enable the user to develop their solutions quickly and test the viability and usefulness of these big data solutions before committing to it fully.

The current approaches to deploy big data applications often obtain resources in either IaaS[10] or PaaS[11] flavor. Typically machines are acquired and configured with the necessary software like data processing framework. These approaches place some burden on the user as they are responsible for deploying, configuring and maintaining the necessary software. For small companies such investments in resources may be untenable. In our work we explore the possibility of offering data processing frameworks in a SaaS flavor.

SaaS has seen steady growth in recent times.[12] When a software is offered as service, the service provider installs, configures and maintains the software. The user accesses the software through a service point. The whole infrastructure supporting the software is abstracted and hidden from the user. The user has no control over the machines on which the software is hosted, or on the software itself. The software provider defines the operations that the user can execute as well as the means to acquire or release resources.

For the SaaS flavor of data processing frameworks, the user only needs to provide the workflow to execute and the number of resources to be used. The service provider would allocate the requested resources and execute the topology on the cloud. The user does not have any control over the machines or the software or where the workflow is

deployed. It is up to the service provider to ensure that the workflow is executed in isolation and its performance is not impacted by workflows of other users. Additionally the service provider needs to ensure that the privacy and security concerns of users are addressed. This could be done externally by using containers or internally if the data processing framework supports features similar to containers.

Apache Storm is a popular framework used to process streaming data. This project is open sourced,[13] in active development and has widespread usage. The workflow for processing data is called a Topology. A topology is described in the form of an acyclic graph. Each node represents some task to be performed on data and the arcs indicate the flow of data between nodes. Compute resources are allocated to the node to execute the task. The part of the framework responsible for allocating the compute resources to the components is called a Scheduler. Resources are typically allocated at the beginning of the execution of the workflow. The user can add or remove additional resources to accommodate changes in the workload. This operation is called Scaling a topology. This becomes a very important operation in the cloud environment, as the user wants to maintain just as many resources as needed. In our work, we concentrate on adding resources to topology also called the Scale Out operation.

Deciding on which node should receive the resources can be extremely tricky in some cases. Consider the scenario where the user wants to scale the topology by allocating resources to relieve the congested components.

In the topology shown in figure 1.1, nodes B and C are congested. Allocating the resource to either B or C seems to be the obvious solution. Allocating the resource to B would decongest the component and increase the output from B. However D and F are already running close to their capacity. It is possible that one or both of them could get congested. The net effect of the allocation would have been to transfer the congestion to a downstream node, resulting in little improvement in throughput. In this case, allocating the resource to C seems like a better option. In general, topologies are complex

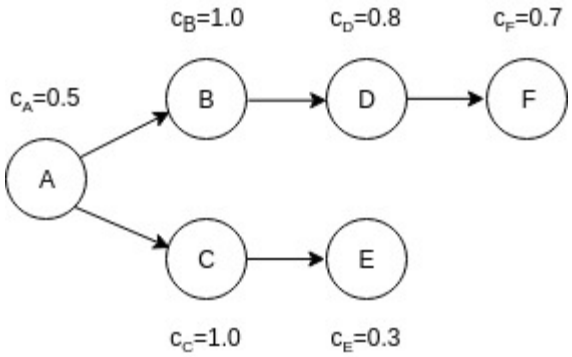

Figure 1.1: A sample topology where C_x indicates the level of congestion. It varies between 0 (no congestion) and 1.0 (fully Congested)

and the throughput could depend on many factors. Thus, generating an optimal allocation for a resource is a challenging task.

When multiple resources are provided for allocation, the scheduling problem becomes even more complicated. Consider the following example when two resources are provided for allocation.

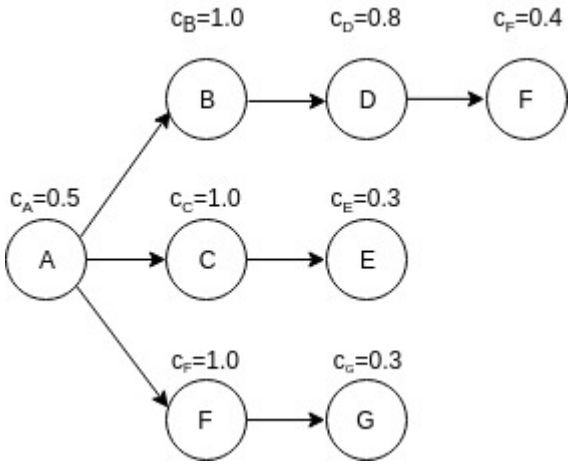

Figure 1.2: A sample topology with more components. C_x indicates the level of congestion. It varies between 0 (no congestion) and 1.0 (fully Congested)

In the sample topology shown in figure 1.2 nodes B, C and D are congested. Two resources are provided to the scheduler to allocate to the topology. These resources can be allocated to

- the most congested components in the B branch (.i.e. to B and D)
- components C and F
- component B and the remaining resource to either C or F

As the number of resources increase, the number of ways the resources can be distributed among the components also increases drastically. Suppose a topology contains 'n' number of components and 'r' resources are procured for allocation. Each of the resource can be allocated in 'n' ways. The total number of possible allocations are n^r. Even if we restrict the allocation to only congested components, during the allocation child components can potentially get congested and will need to be considered for allocation. Thus n^r gives the worst case possibilities.

Iterating and evaluating through all the possible distributions can be time consuming. Picking the right solution requires the scheduler to consider a number of factors related to the topology both in current state, and the possible state after allocation. This further illustrates the complexity of scheduling multiple resources.

There are two approaches to allocate multiple resources. One way would be to allocate them in a serial manner. Each instance of the resource is considered in isolation. At any given time, optimal allocation for only one instance is calculated and assigned, before considering the next instance. Such a scheduling strategy is called Serial allocation. The other way is to consider a set of resources at a time for allocation. The set of resources considered is called a batch. The allocation for all the resources in the batch is calculated at the same time, before considering the next batch. Typically, for small number of resources all the resources are grouped into a single batch. Such a scheduling strategy is called Batch Allocation. A scheduler typically opts for one of these approaches. The relative merits and the trade offs of each of the approaches will be discussed later.

We base our work on a scheduler called Stela.[14] Stela implements a scaleout algorithm that follows the serial allocation method. Stela increases the throughput of a

topology by allocating resources to congested nodes. In order to choose between congested nodes, Stela uses a novel method to measure the impact of the nodes on the throughput. If the throughput of the subtree under the congested component is high, then the impact is determined to be high. Stela picks the component with the highest impact, allocates a resource and updates the state of components in the topology. The steps are repeated for each resource. Once the resource is allocated, it is absolute and it cannot be reallocated.

Stela uses a relatively simple approach. Stela prioritizes paths with the highest impact on throughput to decongest. More often than not, this strategy leads to optimal results. Stela assumes correlation between the impact of a congested component to the throughput. This assumption is the main strength as well as a drawback for Stela. Apart from impact, there are other factors that contribute to the throughput like the state of the subtree. If the subtree is already running near capacity, then the allocation just moves the congestion downstream and the throughput increase may be less than expected. In our algorithm, we attempt to address such limitations.

We now define the conventions of how the term "root", "subtree", "children", "child Subtree" are used in our work. The terms are used only to our work.

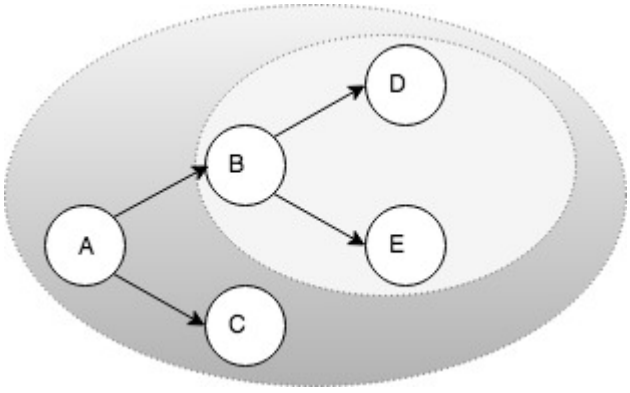

Figure 1.3: Tree structure of the topology. For the entire topology A is the rootnode. B,D,E,C are the children. B,E,D is a child subtree of A. B is the root of the subtree

In the topology shown in figure 1.3, A is the "root" node of tree {A,B,C,D,E,F}.

B,D,E,C are treated as "children" of A. {B,D,E} is a "subtree" of A. {C} is also a subtree of A. In the subtree {B,D,E}, B is the "root" and {D,E} are the "children". In the work, when B is referred to as the "root", it is done so in the context of the "subtree" with B as the root. The children of the root node refer to all the nodes under the root node, and not just the immediate child nodes unless otherwise specified.

In our work, we look at using the new features developed by Apache Storm to support it in a SaaS flavor. We mainly focus on the scale-out operation of a topology. Given additional compute resources, we provide an effective algorithm to allocate the resources to the components to maximize the throughput of the topology.

Chapter 2

Literature Review

For the literature review, we first examine the role and usage of clusters in the context of private in-house deployment and public clouds. We then consider some of the research efforts related to scheduling which determines allocation of tasks from a workflow to a cluster. Finally we consider related work which focuses on model based scheduling, as we use the same approach in our scaleout algorithm.

Clusters were traditionally used for high performance and high throughput computing, and were often shared among many users or teams. HTCondor[15] is one of the oldest software solutions and was initially designed to harness cpu power from idle desktop workstations. User can submit a job to HTCondor. HTCondor then chooses machines from the available pool, executes the job and returns it back to the user. The user can specify some constraints over which machines can execute the job, but it is up to HTCondor to select the right machines.

Borg[16] is a cluster management software used at Google. Borg is used to execute production jobs and non-production jobs. Production jobs are very sensitive to latency and are executed with higher priority, while the non-production jobs are treated with lower priority. The cluster is divided into cells, each cell containing thousands of machines running a mix of production and non-production jobs. Jobs from different

users are executed within the same cell and sometimes on same machine. Each job contains a set of tasks. The user can apply some additional constraints to the tasks. When a user submits a job to Borg, Borg looks for machines that match the user defined constraints to schedule the job. In order to improve machine utilization, Borg uses containers to execute the jobs. Every job has a priority assigned to it. Borg allows jobs of lower priority to be preempted by higher priority jobs to ensure that production jobs are not starved of resources. Borg presents a typical view of a cluster management software used in companies. Other companies have similar software used to manage their private clusters[17][18][19][20] and provide similar functionality as Borg.

The service provided by the in-house cluster are very similar to those provided by the cloud. The user can procure resources on the fly to execute his tasks. But there are also some differences between the services provided by the in-house cluster and the cloud. In the case of in-house clusters, users typically belong to the same company. The requirement for security and privacy is much lower than that of the cloud, as the users can be trusted. Also in-house clusters are designed to maximize resource utilization of the cluster. It expects the user to be more accommodating and willing to release some of his resources for higher priority jobs of other users. But in the cloud, the user expects that his jobs execute in strict isolation from other users. The cloud should not revoke resources from users without his permission. Finally, the cost of over allocating resources is more costly to a user in the cloud than in an in-house cluster. Thus the principles used to design cluster management software and user solutions need to be reexamined in the cloud context. Typically a user acquires virtual machines from the cloud to execute his tasks. Virtual machines offer good security, privacy and isolation guarantees. But the cost of procuring a virtual machine to execute small jobs can be expensive. A resource allocation strategy with finer granularity is favored.[21]

Schedulers play an important role in determining how the tasks within a job are distributed among the machines and consequently determine how well the resources are

used. An efficient distribution can increase the overall utility of the machines, thus requiring fewer machines to execute the job. This is important not only in the context of acquiring resources for initial deployment, but also when scaling the workflow to keep up with the changes in the volume of input. Over allocation of resources could have adverse implications on the cost for the user. We now examine some popular strategies used to distribute the tasks efficiently.

The following set of schedulers belong to a class of communication oriented schedulers. The main goal of these schedulers is to minimize the communication cost incurred on the network and the cpu consumption of the worker hosts, and thus improving the overall utilization of the machines to do useful work. T-Storm[22] aims to reassign executors with the highest inter-node traffic to the same node while making sure that the nodes are not overloaded by the reassigned executors. It is to be noted that an executor typically communicates with many other executors. T-Storm monitors each worker to measure the input and output traffic of each executor. It then sorts the executor by highest inter-node traffic. T-Storm recognizes that moving an executor to a different node to reduce the inter-node traffic with one of its communicating executor could adversely increase inter-node traffic with its other communicating executors. Thus the algorithm calculates the net effect of the reassignment on the overall inter-node traffic, and only reassigns the executor if the overall inter-node traffic of the workflow is reduced. T-Storm assumes that when the executor is moved to a different node, it is added to the same worker process as its communicating partner to reduce inter-process traffic.

Traffic Aware Two-Level Scheduler for Stream Processing Systems in a Heterogeneous Cluster[23] is an improvement over T-Storm as it explicitly addresses the problem of reducing inter-process traffic. The scheduler thus has two goals. It first aims to minimize the inter-node traffic, and second to minimize the intra-node traffic between the worker processes. To reduce the inter-node traffic, the algorithm in the

paper locates a pair of components with the highest traffic between them. It then allocates this subgraph of two components on to a node with the highest available capacity. If the node can accommodate more components, the subgraph is expanded to include a neighboring component. Among all the neighbouring components of a subgraph, the neighbour with the highest traffic with the subgraph is chosen. The subgraph is expanded until the node cannot accommodate any more components. The same process is repeated until all the components are assigned to nodes. The second goal to minimize the intra-process communication within the node is achieved by using a graph partitioning tool called METIS.[24] The algorithm sets a threshold for maximum number of the tasks that can be allocated to the worker. The algorithm then uses METIS to partition the tasks of the component assigned to the node into groups with roughly T tasks each, while minimizing inter group traffic. Tasks in a group are allocated to a worker process.

Poster:Iterative Scheduling for Distributed Stream Processing Systems[25] also uses a graph partitioning technique. Poster uses k-way clustering to form groups of tasks. Poster partitions tasks with high communication among them into the same group, thus minimizing the traffic between the tasks of the groups. Poster determines a threshold T on the number of tasks in the graph that the optimization software can solve in a given time to obtain task allocation. If the threshold T is set high, then the software is allowed more time to process the graph. The threshold is determined by how long the user can wait to obtain an allocation. If the number of tasks in the graph is greater than T, then the algorithm uses a heuristic approach to reduce the size of the graph to T tasks in iterative steps. The central idea is to partition the graph into groups by K-way clustering, and then combine one of the group to a single virtual task, thereby reducing the size of the graph. The capacity of the node is determined by the number of tasks it can host. For each iteration Poster checks if the size of the graph t, is less than or equal to the Threshold T. If t is greater than T, the graph is partitioned into k groups such that the

number of tasks in each group is roughly equal to the number of tasks the node with the highest available capacity can accommodate. The algorithm heuristically assumes that the task group will be allocated to the node with highest capacity. The combined virtual task is reintroduced to the graph and t is updated. The process is repeated, and for the next iteration the machine with the next largest available capacity is considered and the number of partitions is adjusted so that the tasks in each group is roughly equal to the capacity of the considered node. The process stops when $t \leq T$. As an optimization, Poster tries to combine more than one group in each iteration. If more than one machine of the same capacity is available, then it continues to combine groups until no more machines of the same capacity is available or until the threshold T is reached.

An alternative approach to scheduling is to use Load balancing techniques. Load Adaptive Distributed Stream Processing System for Explosive Stream Data[26] focuses on redistribution of the tasks to share the load across all nodes. The paper uses cpu load as the primary factor to determine if a node is overloaded. The algorithm continually monitors the nodes and executors at runtime. Based on the cpu load a machine can be classified as 'overloaded' or 'normal'. The executors are similarly profiled as 'red', 'yellow' or 'green'. The tasks in the 'red' executors are migrated to 'green' executors in 'normal' node to reduce the load on the overloaded machines.

We can benefit by incorporating these approaches to extend our work. From the service provider's perspective, when user makes a requests for a set of resource units, it is up to the provider to determine from which machines these units are allocated. Based on the traffic generated by the users workflow, the provider can apply traffic aware techniques to move the resource units with high traffic among them to the same machine. Alternatively, use Load aware techniques to balance the workload on the nodes.

We now consider some previous research done on Model based scheduling, and other research that uses containers.

Model-driven Scheduling for Distributed Stream Processing Systems[27] seeks to use

a model driven predictive approach to do resource allocation. For a workflow with a given input rate, the algorithm provides the number of resources (VM) required to sustain the input rate and an allocation to map the tasks to the resources. The intuition is that the number of threads required to sustain a stable input rate for each of the tasks in turn determines the number of resources required for the workflow. The algorithm depends on a priori performance modeling of each of its tasks. The paper distinguishes between Linear Scaling Approach (LSA) and Model Based Approach (MBA) for modeling the input rate of the task. LSA assumes that the input rate sustained by a single thread scales linearly as we increase the number of threads. MBA on the other hand explicitly models the effect of number of threads on the input rate. The rest of the discussion uses MBA modeling, as MBA is shown to be more efficient to LSA in experiments. Each task is profiled in a single resource slot. Each resource slot is equivalent to a worker process and each machine has a set number of slots. For each task, the peak input rate attained and the corresponding memory and cpu usage is measured for a given number of threads of the task. Thus this task model table provides a range of input rates achieved corresponding to thread count, memory and cpu usage. This is later used as a lookup table to determine the number of threads, memory and CPU resources required to sustain an input rate.

For scheduling, the input rate for each of the tasks is calculated based on the input rate given to the workflow. The resources required (number of threads, memory and cpu) corresponding to the input rate is looked up in the task modeled table. For mapping these resources to a VM, the paper uses a novel technique called Slot Aware Mapping. Essentially the resources in each VM is divided into slots with a set amount of CPU and Memory. Tasks are selected from the workflow in the order of Breadth First Search (BFS) traversal. A slot from the VM is procured and a set of threads belonging to a task is allocated to it. The number of threads selected depends on the maximum input rate achievable with the cpu and memory resources provided by the slot. This can be looked

up from the task model table. The process is repeated until all the threads of the tasks are allocated to slots. This mapping approach provides an efficient allocation with minimal resources used.

For our work, we use the LSA approach. We profile the peak performance for a slot instead of number of threads and assume that the input rate of the slots scale linearly. The definition of Slot in the paper is similar to our definition of a 'resource unit'. Our algorithm focuses on scaling a topology to increase throughput as opposed to the algorithm described by model based approach that focuses on sustaining an input. Other efforts to model the behavior of the tasks and framework are summarized in paper.[28]

In the next work we examine the role of containers in scheduling. 'Cost-Efficient Enactment of Stream Processing Topologies'[29] uses containers on top of VMs to allow for fine-grained elastic provisioning. The paper extends the VISP runtime[30] and optimizes the execution of CERMA project. CERMA has three sources of input with varying input load, and uses 9 operators to process the input. Each of the operators defines a different functionality. Accordingly each of the operators have different cpu and memory requirements. In addition to this, each of the operators have SLA defined like the maximum processing time for each data item. The operators are deployed on virtual machines obtained from the cloud. The paper aims to minimize the virtual machines acquired and used, while maintaining the SLA for each component. The approach used by the paper is to deploy operators in containers. As the memory and cpu requirements of each of the operators are different, the containers are defined individually with just enough resources. This approach also ensures that resources allocated to a container is not shared by operators, thus an operator can perform at its capacity at all time. The use of containers are also useful for scaling the operators. If the SLA of an operator is not met because of the increase in input rate, a new instance of operator is created. As the memory and cpu requirement of the container is known, the algorithm can easily check for an existing virtual machine to see if it has enough capacity to host the operator. As

every operator executes in its own container, addition or removal of an operator from the virtual machine does not affect the performance of other operators. We advocate for a similar model to allocate resource units from the cloud for Storm.

Efficient Bottleneck Detection in Stream Process System Using Fuzzy Logic Model[31] addresses one more important aspect of predictive allocation. It is important to determine the state of the components when an allocation is made. Typically multiple operators are deployed on the same machine sharing the cpu and memory. Increasing input to one operator could adversely increase cpu consumption of other operators on the same or different machine. Due to the interrelated effects, predicting congestions is a hard problem. The paper uses fuzzy logic control theory to detect congestion. It assumes that each worker process hosts operators of the same component thus processing input stream of a single component, and defines a set of rules to determine if a component gets congested.

The complexity of congestion detection can be attributed to components on a machine sharing a common pool of cpu and memory resources. In our model, we define cpu limits on each resource unit and each resource unit contains operators of a single type. Thus each component has its own pool of memory and cpu. This approach simplifies the task of congestion detection. The user only needs to compute the maximum capacity of the component based on allocated resources and compare it with the input rate.

The final work describes a simulation framework. Modeling and Simulation of Spark Streaming[32] describes the implementation of a framework to simulate Spark Streaming. The framework allows the user to configure a number of parameters mimicing the execution cost of each stage, number of worker nodes, resource specification of worker nodes, data arrival pattern etc. The framework allows the user to experiment with various configurations and measure the processing time, scheduling delay and other parameters.

Chapter 3

Background

Apache Storm is a distributed real time streaming data processing framework. It was developed by Nathan Marz in Backtype. The project was later open sourced after being acquired by Twitter.

The data stream is made of Tuples. A tuple is a named list of values, where each value can be of any type. The workflow submitted to storm is called a Topology. As mentioned before, topology is represented as an acyclic graph. The topology is a map of computations that need to be performed on the tuple, and the subsequent destination for the output.

The nodes in the topology are called Components. The components can be either a Spout or a Bolt shown in figure 3.1. A spout reads data from an external source and emits it into the topology. Bolts receive data from spout or other bolts. Each bolt is made of a number of tasks, each performing the same operation. The tuple is processed by one of the tasks. The output is emitted back to the topology to be consumed by the next bolt. If the bolt does not have any children, then it is called a Sink. In order to distribute the tuples among the tasks of the bolt, storm supports Stream Groupings. There are eight built-in stream groupings. Storm also provides an interface to define custom groupings.

Storm supports two processing semantics, at-most once and at-least once. In case

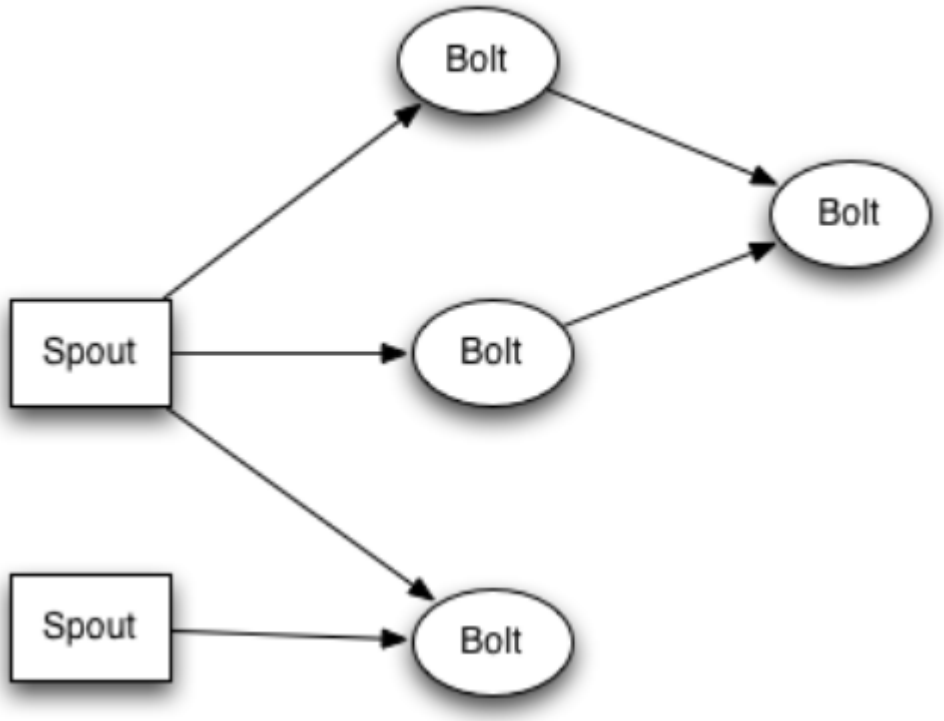

Figure 3.1: Topology structure indicating Spout and Bolt

of at-most once semantics, Storm guarantees that a tuple is delivered to a component for processing, at most once. If the tuple is lost, say due to machine failure or network latency, Storm does not resend the tuple. These tuples are lost. In case of at-least once semantics, storm makes sure that the tuple is processed at least once. Tuples are replayed when there are failures, and replayed till they are processed. In certain scenarios, it is possible for the same tuple to be processed more than once. Using Trident, a higher level abstraction over Storm's basic abstractions, exactly-once processing semantics can be achieved. The type of processing semantics used is very much dependent on the type of problem the application is trying to solve.

3.1 Architecture of Apache Storm

A Storm cluster is a collection of machines, also called as Nodes. It consists of two types of nodes, Master Node and Worker Node. At any point of time, there can be only one active master node in the cluster. It can however have as many workers as needed.

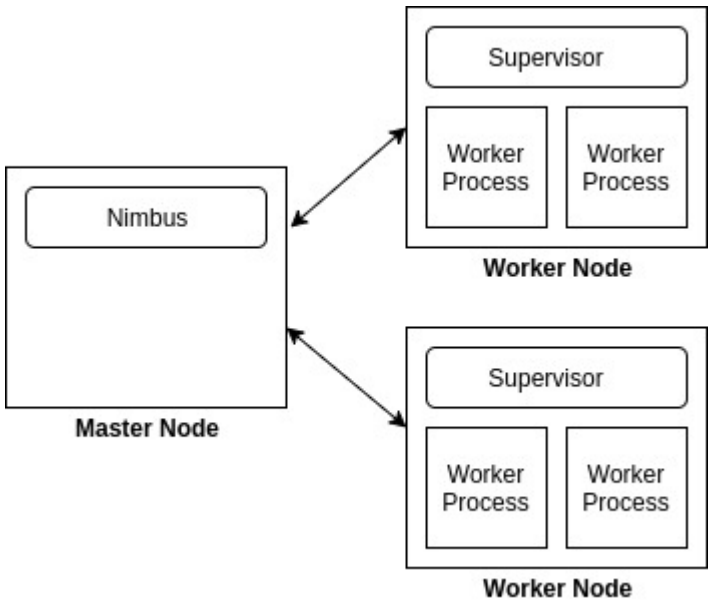

Figure 3.2: High Level Architecture of Apache Storm

Figure 3.2 provides an overview of the Apache Storm architecture. Master node hosts the Nimbus service. A topology submitted to the cluster for execution is accepted by Nimbus service. It is also responsible for distributing code around the cluster, assigning tasks to the worker nodes and monitoring the nodes for failures.

The scheduling decisions are made by a component called Scheduler. These decisions include allocating the compute resources to the components of the topology during the initialization, scale-out, scale-in operations, and reclaiming the resources after the topology is killed.

Storm provides 4 built-in schedulers, namely Even Scheduler, Isolation Scheduler, Resource Aware Scheduler[33] and Default Scheduler. The Even scheduler distributes the tasks evenly among the worker nodes. The aim is to spread out the workload evenly

among the worker nodes, so that none of the worker nodes sits idle. The Default Scheduler uses the same strategy of distributing resources evenly. It in fact calls the Even Scheduler internally. The Isolation scheduler lets the user specify which topologies should be “isolated", meaning that they run on a dedicated set of machines within the cluster. Any other topologies executed on those machines will be evicted, before the isolated topologies are scheduled on them. Resource Aware Scheduling allows the user to specify the resources required for the topology. The user can specify the amount of cpu and memory required for each of the component in the topology. If the scheduler is not able to allocate the requested amount of resources, the topology is not scheduled.

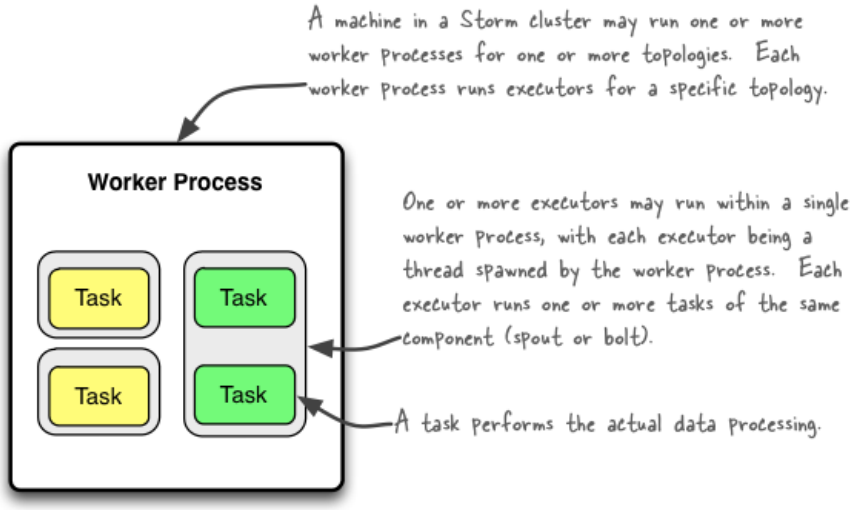

Figure 3.3: Architecture of Worker Process[1]

The worker node hosts a daemon called Supervisor. The Supervisor is responsible for receiving assignments from Nimbus and execute it on the worker machine. Additionally, it communicates the heartbeat status to the Nimbus. The Node may host one more worker process. Each worker process is a JVM process. The worker process may in turn contain one or more Executors. Each Executor is a thread in the worker process, and executes tasks from the same component. The executors may in turn execute one or more tasks. Since each Executor is a single thread, the tasks within the executors are run serially.

The initial number of workers, executors and tasks to be created for the topology, can be specified in the topology description. These initial numbers are called “parallelism_hint". It basically indicates how many of the tasks are being executed in parallel. Suppose if the component has 8 tasks. If it requests for only one worker and executor, then all the 8 tasks are executed serially on a single thread (Executor). If the component requests two executors, then the tasks are split between two threads, and hence more tasks are executed in parallel.

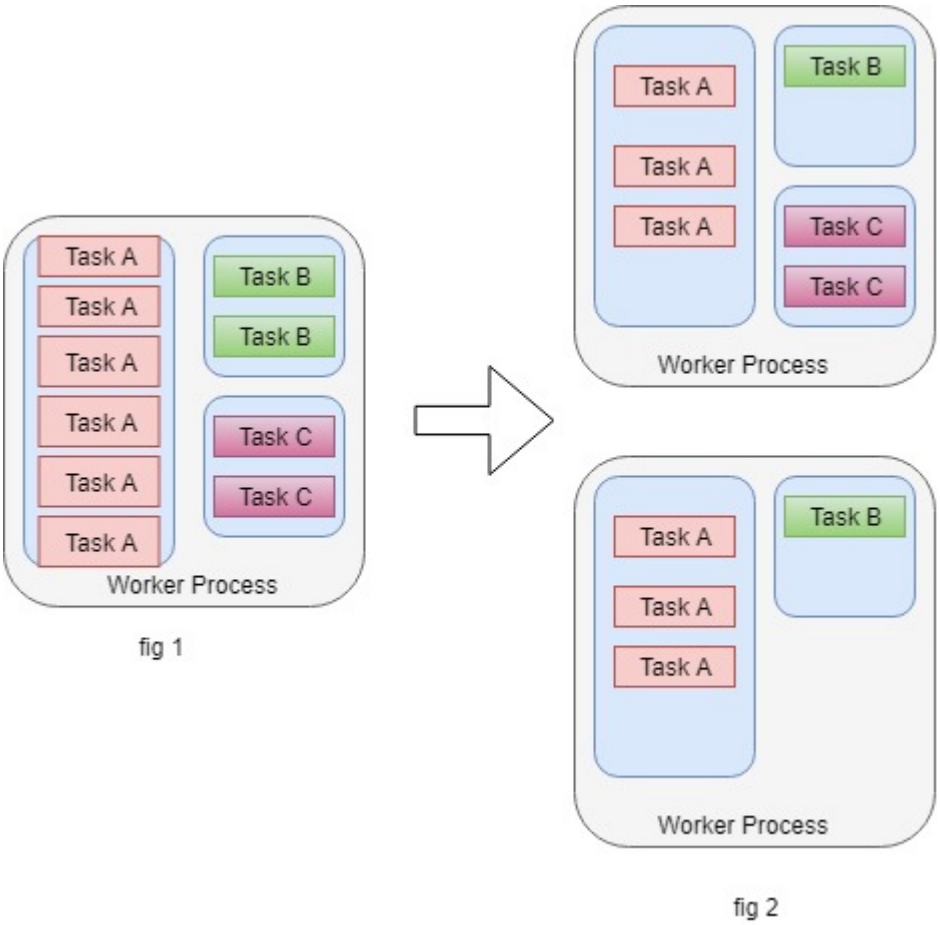

Figure 3.4: Task distribution in a Topology

Figure 3.4 shows the distribution of tasks in a topology. The user scales out the topology by adding more resources. Even scheduler distributes the tasks evenly across the resources. The 6 tasks of A share a single executor as shown in in the 3.4. When the

number of executors are increased to 2, the tasks of A are distributed evenly among the two executors with 3 tasks each. If the number of executor is increased to 3, each executor will execute 2 tasks of A. No changes are seen for Task C, as the number of executors remain same. The same principle holds true when the number of worker processes is increased. The executors are redistributed so that each of the worker process hosts equal number of executors (except in the case when the number of executors is odd).

These numbers can be adjusted during runtime. The name of the command used to adjust the parallelism hint is called “rebalance”. It allows the user to modify the number of workers and executors assigned to a component. Note that the number of tasks in a component cannot be modified by the rebalance command.

Storm version 1.0 and above implement automatic Backpressure. This feature essentially slows down or throttles the process of a parent component whenever a component is congested in the topology. In Storm 1.x the feature can be disabled by configuration options. In Storm 2.0, the feature is redesigned and it is no longer possible to disable it using configuration options. The new implementation uses the state of the internal message buffers to determine if the worker is saturated with messages. If it is, then it sends a backpressure signal to the worker process of parent component requesting it to stop sending tuples until further notice.

Figure 3.5 shows the internal message buffers[1] used in a worker process. The internal message buffers play a major role in the operation of Storm. The tuples are transferred between workers in batches, essentially making the input pattern bursty. The message buffers essentially insulates the rest of the components from the input pattern. The message buffers are essentially implemented as queues. The worker process uses three queues. The worker receive thread adds the incoming tuples to the Worker Receive queue. If the receive queue is full, the receive thread redirects the incoming tuples to a different queue called Overflow queue. This queue is used by the Backpressure mechanism to send backpressue signal to the parent component. The third queue is the

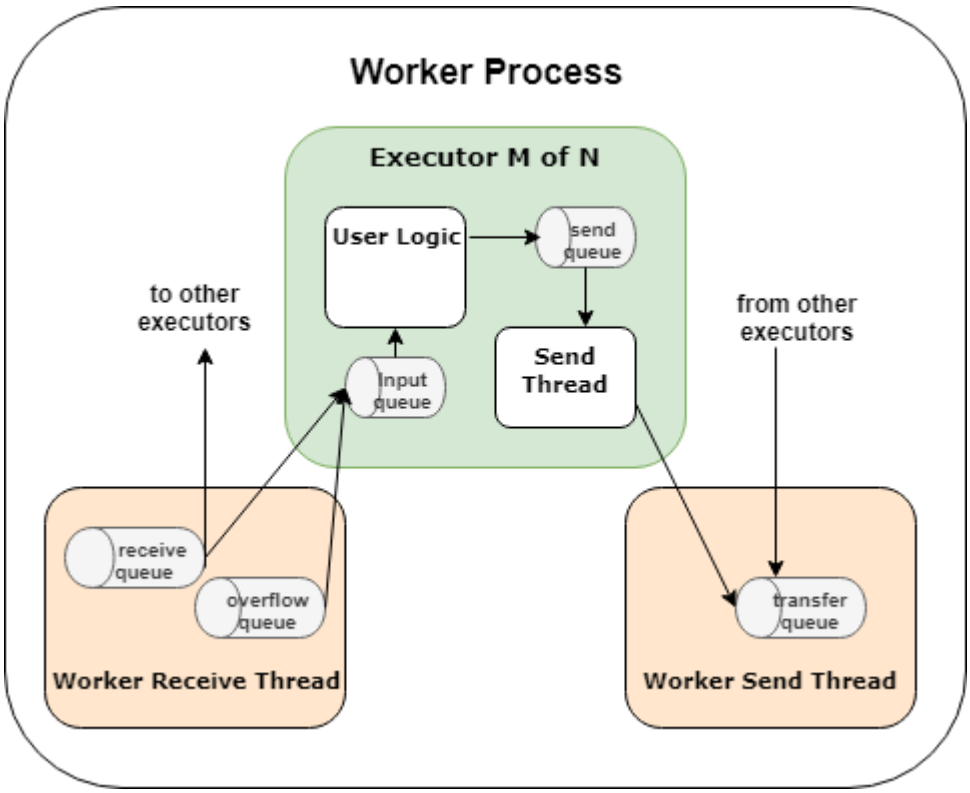

Figure 3.5: Internal message buffer of worker process

Transfer queue, which acts as a common output buffer for a worker. The tuples emitted by the executors are added to the output queue. The transfer thread then sorts the tuples by destination, collects tuples into batches, and sends them to the destination. Each executor maintains two queues, the Executor Input queue and the Executor Send queue. The tuples from the input queue are sorted and added to the respective executor's input queue. After operating on the tuple, executor sends the emitted output tuple to the send queue. A send thread then collects the tuples from send queue of executor and adds it to the Output queue of worker.

Storm version 2.0 implements support for cgroup Enforcement. This feature is critical to support Storm in a SaaS flavor and essentially provides the same features as a container. cgroups[34] (abbreviated from control groups) is a Linux kernel feature that limits, accounts for, and isolates the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes. cgroups allows the provider to enforce resource limits

on the worker process. This can ensure that each worker processes executes in isolation and is not affected by other workers.

We define Input rate for a component as the number of tuples arriving to the component in unit time. The Output rate similarly is the number of tuples emitted by component in unit time. Processing rate is defined as the number of tuples processed by the executors of the component in unit time.

Apache Storm 1.2 introduced a new metric system, to allow users to collect and report the internal state of the cluster. The metric system is based on Dropwizard Metrics. The new system defines a set of API's that can collect different kind of information. A Timer metric aggregates timing duration and provides duration statistics. A Meter metric measures events and provides an exponentially weighted moving average for one-, five- and fifteen-minutes. A Counter metric simply counts the number of events. In our implementation we extensively use the Meter metric.

Our approach will be using the cgroups feature implemented in Apache 2.0 to create worker process under cgroups, and the metric system to measure the number of tuples moving through each of the internal queues. More details are provided in the implementation chapter.

Chapter 4

Stela

Stela is a scale-out algorithm designed to increase the throughput of the topology. Stela achieves this by allocating resources to congested components. The idea is that the increased processing power due to newly allocated resources increases the output from the congested component. This output then propagates downstream and leads to overall increase in throughput.

Stela follows the serial allocation approach. When multiple resources are made available to allocate, Stela decides on each resource in isolation. Once the decision is made, it is not reconsidered or reverted when deciding the allocation for the subsequent resources.

The Stela algorithm for allocating a single resource can be summarized into three tasks as listed below.

1. Identify congested components
2. Measure desirability of each congested component
3. Select a congested component and allocate a resource

If the user provides more than one resource, the above tasks are repeated in order for each resource.

4.1 Identify Congested Components

A component is determined to be congested by the Stela algorithm if the processing rate of the component is less than or equal to its input rate. Essentially, the number of tuples input to the component in a unit time is greater than the number of tuples that can be processed by the same component. An excess of input leads to tuples being dropped. The processing power of the component is proportional to the amount of resources allocated.

The algorithm measures the input, output and processing rates of each component in real time at regular intervals. If the input rate for the component is greater than the processing rate, the component is classified as congested. All the congested components of the topology are stored in a list called “congestedMap”

4.2 Measure Desirability of Congested Component

Once the list of congested components are identified, these components or the allocations derived for these components need to be ranked.

In order to measure the desirability of each component, the algorithm uses a novel metric called “Expected Throughput Percentage” or ETP. ETP essentially gauges the importance of each node to the overall throughput of the topology. More specifically, the percentage of the total throughput contributed by the subtree of the congested node. To measure the output from a subtree, ETP considers only those sinks that do not contain a congested component in the path as shown in Algorithm 1

The ETP for a component is calculated as below. ETP for each congested component is calculated and stored in a map called “ETPMap".

$$ETP(M) = \frac{Throughput_{EffectiveReachableSinks}}{Throughput_{Workflow}}$$

Algorithm 1 ETP Calculation Algorithm

```
1:  procedure FINDETP(Topology topology, Component component)
2:      if component.hasChild == false then
3:          return component.outputRate/throughput(topology)
4:      end if
5:      subtreeSum = 0
6:      queue.add(component.children)
7:      while queue.isEmpty() == false do
8:          Component child = queue.pop()
9:          if child.isCongested == true then
10:             continue
11:         end if
12:         if child.hasChild == true then
13:             queue.add(child.children)
14:         else
15:             subtreeSum += child.outputRate
16:         end if
17:     end while
18:     return subtreeSum/throughput(topology)
19: end procedure
```

4.3 Choosing The Component

Stela employs a greedy approach to choose among the congested components. It uses the prepared ETPMap to pick the component with the highest ETP. Once the resource is allocated to the component, the input and output for the component and the downstream components are updated. If k resources are allocated to component M, the new processing rate for the component after allocating one additional resource is calculated as

$$\frac{k+1}{k} * CurrentProcessingRate(M)$$

The Stela scaleout algorithm is presented at Algorithm 2 .

Algorithm 2 Stela Scaleout Algorithm

```
1:  procedure STELA(Topology topology, int numRes)
2:      Map<Component, int> allocationMap = null
3:      while numRes > 0 do
4:          List<Component> congestedMap =
5:                getCongestedComponents(topology);
6:          for Component comp : congestedMap do
7:              etp = FindEtp (topology, comp)
8:              ETPMap.add(comp, etp)
9:          end for
10:         Component target = ETPMap.getMaxEtp()
11:         currentAlloc = allocationMap.get(target)
12:         allocationMap.add (target, currentAlloc+1)
13:         numRes = numRes-1
14:         update input, output and processing rate of all components
15:     end while
16: end procedure
```

4.4 Drawbacks of Stela

In its outlook, the algorithm is fairly optimistic. The algorithm does not consider the after effects of allocation. Consider fig 1 for example.

If the ETP of B is greater than C, the resource is allocated to component B. However, the increase in the throughput is not realized as D gets congested. Stela does not check if the downstream components are able to handle the additional output from component B.

Stela does not examine or measure the consequences of an allocation before making the allocation. So it is unable to detect scenario mentioned above. As a result, Stela produces less than optimal allocations in the following two cases. First, downstream nodes do not have the capacity to handle the increased output. Second, the degree of congestion of a component is not taken into account. That is, a lightly congested component is treated the same as a heavily congested component. It is possible that the increase in the output on a lightly congested component might be too little to improve the throughput significantly. Thus being able to predict the after effects, and estimate the increase in throughput could help overcome these limitations, which motivates our

work in this thesis.

Chapter 5

Scale-Out Algorithm

5.1 Goals and Assumptions

We recognize scaling to be an important operation in the cloud, and is critical to the success of such a service. Our work proposes an algorithm to a support scale-out operation. More specifically, given a set of resources for scheduling our algorithm provides an allocation to decongest components and maximize the throughput increase. Additionally it attempts to predict the state of the topology if the suggested allocation is indeed committed.

The scale out algorithm focuses on maximizing the throughput of the topology by allocating resources to congested components.

In order to narrow the scope and simplify the problem, our work makes the following assumptions about the topology and the cluster.

- The resources are made available in the form of resource units. Each resource unit provides a fixed amount of processing power, that is the cpu power on a machine of a cluster. The processing power provided by each resource unit is equal. This can be enforced by using cgroups or containers.
- We assume that the congestions are caused by the lack of compute power allocated

to components. It is assumed that network capacity is infinite, and the effects of network latency can be ignored.

- We assume that when a topology reaches a steady state, the input, processing and output rates of the components remain nearly constant.
- It is assumed that the cluster is configured to disable backpressure enforcement.

When backpressure is enforced, Storm throttles the input whenever a component gets congested. Thus a single component can become the bottleneck for the entire topology. This not only limits user's ability to control execution, but can also adversely affect resource usage. It prohibits components from fully utilizing the resources allocated to it, as the input can get throttled due to congestion in a different part of the topology.

The flip side of the argument is that the tuples can get dropped when backpressure is disabled. Our algorithm caters to the same class of topologies as Stela, where loss of tuples is tolerable. In the future work section, we propose an approach to further improve backpressure system, where the loss of tuples can be avoided.

The above assumptions are consistent with those made by Stela. In our approach, each component is free to scale as required. In a cloud environment this allows for maximum flexibility and control over how the resources are allocated and used by the topology. The user can prioritize certain components over others, sometimes at the expense of congesting components of lower priority.

5.2 Approach and Design

The general approach taken by our algorithm is very much contrary to Stela. Our algorithm chooses to use Batch allocation and measures the effects of an allocation before it is made. Our algorithm is conservative and considers allocations that do not

worsen or introduce any new congestion in the downstream components. We propose a simple model to predict the input and output rates of the components after an allocation is made. Finally, we recognize the limitations of our algorithm and provide an approach to incorporate Stela as a co-scheduler. Combining both algorithms provides better allocations to a larger set of scenarios than each of the individual algorithms.

5.2.1 Resource Units

The resources are provisioned in terms of resource units. Each resource unit is guaranteed a certain processing power by using cgroups. All the resource units have equal processing power allocated to them. The user can request resource units to be allocated to a component. The component uses these resources provided to execute its tasks. For the rest of the discussion the term resource is used to indicate resource unit.

5.2.2 Batch Allocation

Batch allocation strategy is favored by our algorithm. Serial allocation strategy does not take the the number of resources provided into consideration before making scheduling decisions. When a resource is allocated to a component, the output propagates through its subtree. If any of the components in the subtree are congested, then additional resources need to be allocated. Knowledge of the number of resources needed by a subtree to achieve required throughput and the number of resources provided to a user could be helpful in deciding allocations.

5.2.3 Compute Model

In order to estimate the state of the components in the topology we build a simple model. The model serves the following purposes

1. Represents the state of the components in the topology

2. Allows the algorithm to predict if the allocation causes a congestion in the subtree
3. Allows the algorithm to estimate the throughput for an allocation

The processing power for components are supplied in resource units. So logically, we can view components as made of resource units. Components are profiled to measure the maximum input, output and processing rates when executed in a single resource unit. The total rates for a component is obtained by multiplying the number of resources allocated to the component with their profiled rates. The components are organized in a structure similar to the topology complying to their parent-child relationships. The network capacity is assumed to be unlimited. We define the following properties for the component.

unitCount is the total number of resource units allocated to a component.

congestion is a flag set to a component to indicate if it is congested. A component is determined to be congested if the inputRate exceeds processingRate.

inputRate is the total number of tuples flowing into a component in unit time. It can be calculated as the aggregate of the inputs coming into all the resource units that are executing the tasks of the given component M. Note that this accounts for both processed tuples and dropped tuples.

$$inputRate(M) = \Sigma inputRate(resource\ units), \forall\ resource\ unit \in M$$

droppedRate is the number of tuples dropped by the component if the processing power is not sufficient. This is measured from the component.

$$droppedRate(M) = inputRate(M) - processingRate(M)$$

processingRate is the input that is processed by the component in unit time. This

could be less than or equal to the inputRate.

$$inputRate(M) = processingRate(M) + droppedRate(M)$$

outputRate is the total number of tuples flowing out of a component in unit time. This is calculated as the aggregate output of all the resource units that are executing the tasks of the given component.

$$outputRate(M) = processingRate(M) * outInRatio(M)$$

outInRatio is a simple ratio of the output to processing rate of a component.

$$outInRatio(M) = \frac{outputRate(M)}{processingRate(M)}$$

maxProcessingRate is the maximum input the component can process. This is determined by the amount of resources allocated to component. Max input for a component is profiled for a resource unit beforehand. Given 'n' is the number of resources allocated to component M,

$$maxProcessingRate(M) = maxProcessingRate(resourceunit) * n$$

childOutputRatio(i) is the ratio of tuples destined for the i^th child component of component M. This is profiled beforehand.

$$childOutputRatio(i) = \frac{rate of tuples\ assigned\ to\ child_i}{outputRate(M)}$$

throughput of a topology is calculated as the sum of all the totalOutput of leaf nodes.

$$throughput = \Sigma totalOutputRate(M),\ \forall\ M\ is\ \ a\ leaf\ node\ in\ topology\ T$$

throughputResourceRatio is calculated as the ratio of increase in throughput to the number of resources allocated. Given 'n' is the number of resources allocated to a subtree rooted at M,

$$throughputResourceRatio(M) = \frac{increaseInThroughput}{n}$$

5.2.4 Calculating Throughput for Allocation

The current throughput can be directly obtained by measuring the output of leaf components. This section describes an approach to calculate the throughput when a new resource is allocated.

As per the assumptions made, the processing rate is entirely dependent on the cpu allocated. If 'n' is the current number of resources allocated to a component M. When a new resources is added, the maximum processing rate and thus the maximum output rate of the component changes.

- The new processing rate is calculated as shown below.

$$newMaxProcessingRate(M) = \frac{(n+1)}{n} * maxProcessingRate(M)$$

- The new output rate is accordingly calculated as below.

$$newOutputRate(M) = newProcessingRate(M) * outInRatio(M)$$

The children use the same procedure to update their processing, input and output rates. These changes are propagated further till all the components in the subtree are updated. We update the values for each component in the subtree by using a modified BFS traversal. The new throughput of the topology is then calculated as the sum of the output of all the leaf components with its updated output rates. It is to be noted that the

model is simple and makes assumptions about the topology. In our testing we show that the model predicts the rates of the components with reasonable accuracy.

5.2.5 Strategy

This section describes the high level strategy used by our algorithm. The user provides the topology and the number of resources to be allocated. The algorithm works on the principle that the throughput of the topology is constrained due to congestions in components and the output rate can be increased by decongesting components.

An allocation is defined as a map of components associated with the number of additional resources to be allocated. The allocation map is initialized to a list of all the components mapped to the value 0 indicating no additional resources are currently allocated to any component. When the algorithm decides to allocate a resource to the component, the value corresponding to the component in the map is incremented.

Similar to Stela, the algorithm starts by identifying all the congested components in the topology. Components are profiled for their maxProcessingRate. A component is determined to be congested if the maxProcessingRate for the component is less than the inputRate. This can also be determined by measuring the droppedRate. If a component has a droppedRate above a threshold, then the component is congested. In our work we use the maxProcessingRate.

The strategy is to allocate resources to a congested component to increase its output. If any of the components in the subtree gets congested additional resources can be allocated till it is no longer congested. It is important to note that we do always decongest a root component completely every time it is considered. The extent to which the root component is decongested depends on the output the subtree can support with the remaining resources. Thus the resources allocated can be split into two parts, the ones allocated to congested root component, and the ones allocated to subtree to sustain the output. If the total number of resources required is more than the resources available

then this introduces or worsens congestion of at least one component in the subtree.

For each congested component, the algorithm allocates resources to the root congested component incrementally. For each iteration one additional resource is allocated until the root component is completely decongested. After each allocation to the root component, the algorithm propagates the increased output through its subtree. If any of the components in the subtree gets congested, the number of additional resources is calculated for the component. The new output is then propagated through its child components. At the end of the subtree, the algorithm obtains the total number of additional resources needed by the subtree, as well as the throughput increase caused by the additional resources. If the total number of resources exceeds the number of resources provided, then the allocation is rejected.

Each valid allocation is stored in a map keyed by the root component and the number of resources allocated to the root. The map also stores some useful information like the throughput increase, number of resources needed and the throughputResourceRatio. The list of valid AllocationMap are maintained in a list called AllocationMapList.

The AllocationMapList is sorted in decreasing order of throughputResourceRatio. The algorithm tries to select as many allocations as possible in order, as long as the total number of resources required for the allocations is less than or equal to the resources provided by the user. These allocations are combined together to obtain optimal allocation. Though we are selecting allocations in a greedy serial order, the number of resources allocated at each step may be greater than 1. So essentially we are allocating resources in batches. A further improvement to this method is to use an algorithm similar to the one that solves knapsack problem to ensure the selection of allocations is optimal as well.

Next, we describe how to combine allocations. Though each of the allocations are calculated separately, it is possible that some of the components in these allocation

overlap with each other. Consider two allocations A and B. If the root of allocation B is a descendent of the root of allocation A then allocation A is a superset of B. This follows from the rule that the algorithm considers only the allocations that completely decongest all of the descendents of a subtree rooted at a congested node. Since allocation A is a superset of B, if A is chosen then B is discarded from our solution.

5.2.6 Residual Resources

Our algorithm uses a greedy approach similar to Stela to select allocations from the AllocationMapList. This number of resources allocated in each greedy selection is dictated by the allocation considered in the allocationMap. It is possible that in some cases not all the provided resources can be allocated. There are cases where the remaining resources are not sufficient to satisfy any allocation from the AllocationMapList. When this happens, these resources are called Residual resources.

Many approaches can be taken to allocate the residual resources. One approach would be to choose an unused allocation from the sorted allocationMapList with the highest throughputResourceRatio and allocate resources to components encountered in a Deapth First Search (DFS) or Breadth First Search (BFS) traversal. Since the number of resources are not sufficient the subtree becomes partially congested. The outcome of this approach can vary greatly depending on the number of remaining resources, the selected allocation and the topology itself.

We select an unused allocation with the lowest number of resources required instead of one with the highest throughputResourceRatio. The idea is that even if we worsen the state of congestion with the residual resources, fewer resources would be needed to decongest these newly created congestions. Resources could be allocated to decongest these components the next time user wants to scale out the topoplogy.

Once the allocation is selected, we allocate one residual resources to the root component. The increased output now causes new components in the subtree to

congest. Thus we have a problem of allocating the remaining residual resources to the subtree to decongest it. This is exactly the same problem as the scale out problem, except for in this case we are trying to scale out a subtree instead of the entire topology. This gives us a good premise to use recursion to solve the problem.

A new topology is created which mirrors the subtree. One resource is allocated to the root of the subtree and our algorithm could then be called with the remaining number of residual resources to obtain an optimal allocation for the subtree of newly congested nodes. If all of the resources are still not allocated to the subtree because they are not sufficient to decongest any node and its descendents, the allocation with the lowest resource requirement for the subtree can again be selected and a new topology can be created. This process can be repeated until all the resources are allocated. A residual allocation map is created by combining the optimal allocation obtained from each of the steps to the optimal allocation for the main topology. The algorithm section provides more details about this process.

Thus the output from our algorithm to the user would be two allocation maps

- optimalAllocationMap - does not worsen the congestion in the topology. This does not contain allocations for residual resources
- residualAllocationMap - is the combined map obtained by merging the optimalAllocationMap with the residual map obtained by the aforementioned recursive algorithm.

5.2.7 Algorithm

The algorithm expands upon the strategy outlined in the previous section. We first introduce the two main data structures used in the algorithm as shown in Algorithm 5.2.7. Allocation is a simple map structure, that maps a component to the number or resources allocated to it. Each component is initialized to 0 resources and can be

incremented or decremented as needed. The second structure AllocationMap is used to associate an allocation with a congested component. For each congested component in the topology, the algorithm tries to decongest it and its subtree by allocating resources. The root of the subtree is stored in headComp. Our algorithm explores options to partially decongest the root node, so headRes stores the number of resources allocated to just the root component of a subtree.

Algorithm 3 Data Structures

Allocation $\{comp_1 : nRes_1, comp_2 : nRes_2, ...\}$;
▷ foreach component $comp_i$ in the topology, $nRes_i$ is number of resources allocated to ith component.

AllocationMap {
$headComp$,
$headRes$, ▷ resources allocated to headComp of a subtree
$Allocation$, ▷ for the subtree
$Throughput$, ▷ throughput increase due to the Allocation
$totRes$, ▷ sum of $nRes_i$ in the Allocation
$throughputResRatio$
}

The scale out algorithm as shown in Algorithm 4 accepts the topology and number of resources to be allocated from the user. The topology contains the runtime information like the input rate, processing rate, output rate etc of each component. The algorithm can be divided into four parts.

- Get allocation for the topology using the approach described in previous section (line : 2). For the rest of the discussion we refer to it a predictive approach.
- Allocate the residual resources to the sub topology (lines 6-14).
- Get allocation using Stela approach (line : 15).
- Pick the allocation that is predicted to give the higher throughput (lines 16-18).

Line 2: gets the allocation using our predictive approach as described in Algorithm 5. The predictiveAlloc returns two values. The first is an allocation that does not worsen

congestion of any component stored in optimalAlloc variable. The second is allocaTable. It is essentially the list of all congested components and respective allocations to decongest their subtree. These will be used to allocate residual resources.

Lines 6-13 allocate any resources not allocated by predictiveAlloc. Line 7 sorts unused AllocationMaps in allocTable, picks the allocation that needs the least number of resources say c. We use our predictiveAlloc algorithm to allocate the residual resources. We creates a new topology by copying the components of subtree under the headComp of allocation c. This new topology is used as input in Line 11. Line 10 allocates a resource to the head node of the new topology. This causes some of the components of the subtree to get congested. Line 11 tries to allocate the remaining resources to the new topology. The optimal allocation returned from the step is used to update residualAlloc. The remaining unallocated resources are updated, and the loop is executed with the new AllocationMapList till all the resources are allocated. Once all the resources are allocated residualAlloc from our algorithm is obtained. Line 14 obtains allocation using stela algorithm. We then pick the allocation with the higher throughput and apply it for the user.

The merge operation in line 12 is performed by adding the resources allocated to each component in both allocation maps. For example if map 1 allocates 1 resource to component A 2 resources to component B, and map 2 allocates 0 resources to component A and 2 resource to component B, then the merged map allocates 1 resource to component A and 4 resources to component B.

Algorithm 5 gives the pseudocode of the predictive strategy we discussed in the previous section and can be summarized into th following steps. Line 2 gets a list of congested components in the topology. Lines 5-25 obtain a list of candidate allocations. Essentially for each component in congested list, one or more resources are allocated to head component. By doing a bfs traversal through its subtree the number of resources needed to decongest the whole subtree is calculated as shown in Lines 12-16. If the total

Algorithm 4 Scale out Algorithm

```
1:  procedure ScaleOut(Topology topology, int numRes)
2:      {optAlloc, allocTable} =
3:              PredictiveAlloc (topology, numRes);
4:      Allocation residualAlloc = optimalAlloc
5:      excessRes = numRes - countRes(optAlloc)
6:      while excessRes > 0 do
7:          sortByResRequired(allocTable)                    ▷ non decreasing
8:          c = allocTable[0];
9:          newTopo = makeTopologyOfSubtree(c.headComp)
10:         allocate 1 resource to c.headComp in newTopo
11:         {tempAlloc, allocTable} = PredictiveAlloc (newTopo, excessRes-1)
12:         residualAlloc.merge(tempAlloc)
13:         excessRes = numRes - countRes(residualAlloc)
14:     end while
15:     stelaAlloc = GetStelaAlloc(topology, numRes)
16:     if throughputIncrese(stelaAlloc) > throughputIncrease(residualAlloc) then
17:         residualAlloc = stelaAlloc
18:     end if
19:     apply residualAlloc
20: end procedure
```

number of resources required for allocation is less than or equal to the number of resources provided to the user, the allocation is stored in allocMapList in Line 22.

The allocMapList is sorted by throughputResourceRatio. The strategy is to select as many allocations as possible that have high throughputResourceRatio as long as we have enough resources. The final optimal allocation is calculated, by walking through the list in order as shown in Lines 28-40. For each allocation considered, if it requires less than or equal to the remaining resources, we update the optimal allocation and remaining resources in Line 36-38. The step exits when all the resources are allocated (Line 30) or we reach the end of the allocMapList.

Consider the graph in figure 1.3. If both components A and its child B are congested, then our predictive algorithm calculates an allocation to decongest both of the subtrees of A and B separately. As we follow the rule to decongest all of the components in subtree, the allocation for A is a superset of the allocation for B.

Now consider the case where the throughputResourceRatio of the subtree of A is greater than B. When we sort the allocations in line 26 in Algorithm 5, allocation A is placed above B. As a result, when calculating the optimal allocation allocation of A is considered first. If we do have enough resources, allocation of A is added to optimal allocation snapshot (Line 36). When we consider allocation of B we need to be careful when adding all of the allocation to optAlloc since the resources for subtree B will be allocated twice.

We use the following rules when considering overlapping allocations. If optAlloc already has an allocation that is a superset of the allocation being considered, say allocation of A is already added to optAlloc and allocation of B is considered, then this allocation of B is ignored.

If the allocation being considered is a superset of any of the allocation already part of optAlloc, say allocation of B is already added to optAlloc and allocation of A is being considered, then the allocation of B is removed from optAlloc and allocation of A is added in its place.

We check if an allocation overlaps with the current snapshot optimal allocation optAlloc. If any of the components in optAlloc are descendents of the considered allocation, the number of overlapping resources are returned.

5.2.8 Comparision to Stela

The approach taken by our algorithm contrasts with that of Stela. Stela is good at aggressively seeking out high reward paths. Though the actual rewards may vary depending on the state of the topology. Our algorithm in contrast is good at considering different subtrees based on the number of resources provided, e.g. if decongesting a high reward path takes 4 resources, and we have only 3, our algorithm looks at other subtree that can be decongested with three resources. If one or more of such subtrees are found, our algorithm selects the one that leads to the highest increase in throughput.

Algorithm 5 Decongesting allocation

```
1: procedure PREDICTIVEALLOC(Topology topology, int numRes)
2:     List<Component> congestedList = getCongestedComponents(topology);
3:     currThroughput = GetThroughput(topology)
4:     List<AllocationMap> allocMapList = null
5:     for Component congComp : congestedList do
6:         Allocation tempAlloc; Queue queue;
7:         for int res = 0; res < numRes; res++ do
8:             tempAlloc.initialize()                    ▷ initialize nRes_i to 0
9:             throughput = 0; tempAlloc[congComp] += res
10:            queue.erase(); queue.add(congComp.children)
11:            while queue.empty() == false do
12:                qc = queue.pop(); queue.add(qc.children)
13:                qc.NewIn = qc.inputRate;
14:                qc.newOut = qc.newIn * qc.outInRatio
15:                qc.newRes = ceil(qc.newIn/qc.maxProcessingRate)
16:                tempAlloc[qc] = qc.newRes - qc.oldRes
17:            end while
18:            allocRes = CountRes(tempAlloc); newThroughput = GetThroughput(topology)
19:            if allocRes <= numRes then
20:                thrInc = newThroughput - currThroughput
21:                resThrRatio = thrInc/allocRes
22:                allocMapList.add(congComp,res,tempAlloc,thrInc,allocRes,resThrRatio)
23:            end if
24:        end for
25:    end for
26:    sortBythroughputResourceRatio(allocMapList)
27:    AllocationMap optAlloc = null; remRes = numRes
28:    for tempAlloc : allocMaplist do
29:        if remRes <= 0 then
30:            break;
31:        end if
32:        tempRemRes = remRes
33:        overlapRes = CheckOverlap(optAlloc.Allocation, tempAlloc.Allocation)
34:        temp = remRes+overlapRes
35:        if countRes(tempAlloc.Allocation) <= temp then
36:            optAlloc.merge(tempAlloc.Allocation)
37:            allocMapList.remove(tempAlloc)
38:            remRes -= (countRes(tempAlloc.Allocation) - overlapRes)
39:        end if
40:    end for
41:    return optAlloc, allocMapList
42: end procedure
```

When constrained by limited resources we can make allocations in one of two ways. That is, to either partially decongest the high throughput subtree or fully decongest a another subtree. Which of the option is better in terms of throughput increase depends entirely on the characteristics and state of a topology.

It is also easy to see that because of the contrasting approaches between our algorithm and Stela, they work better in different scenarios. Stela tends to do better when the outputs are mainly produced by a few leaf nodes, and our algorithm tends to be better when the outputs are more evenly produced by the leaf nodes. This follows from the above reasoning that there may be multiple subtrees that produce similar throughput with lesser number of resources required. However if the outputs are concentrated along a few paths then the other subtrees explored by our approach may not produce as much output as these partially decongested high throughput paths.

Our algorithm and Stela together cover a greater number of scenarios than each of them alone. We thus decide to combine Stela approach with our algorithm. That is, we compare the residualAllocationMap from our algorithm with the allocationMap produced by Stela. We pick the better allocation based on the predicted resultant throughput and apply the allocation for the user.

Chapter 6

Implementation and Evaluation

The algorithm is a standalone component and is not integrated with Storm. The program takes the metrics as input and outputs two allocations optimalAllocation and residualAllocation as described in Algorithm 5. For the case of testing, we modify the algorithm to present the allocations by Stela and our approach separately. The algorithm is a standalone program written in Java. It does not aim to be an autoscaling program, although it can be accomplished with a bit of external scripting work. The advantage of being a standalone program is that it does not interfere with the performance of Storm. It could be hosted on different machines, and can execute as many instances as needed. The user can experiment with different allocations without affecting the topology executing in Storm and apply the changes only when he is comfortable.

The input contains the list of components in the topology and the number of free resources to be allocated. Each component contains topology related information the like id of its parent and children components. It also contains the runtime state of the component like input, output, processing and dropped rates. If the component has multiple children, it maps the ratio of the total output going to each children. For example, if a component has 2 children A and B, and 100 tuples are emitted, where 60 are sent A and 40 to B, then the output ratio for A is 0.6 and B is 0.4. When an allocation is

made to the parent components, the input to the child is obtained by accumulating the input from each of its parent components and the output ratio to the child component.

The input uses JSON format to describe the component information. We use this format because current webapi for storm returns output in JSON format. It would be possible to implement or extend the existing webapi to return the component runtime information, and use it as input to the program.

6.1 Changes to Storm Framework

Apache Storm 2.0 implements cgroup support for its worker process. When enabled, the Apache Storm creates its process in a control group. cgroups provide compute guarantee and isolate the cgroup processes from each other. However, in its current state the second requirement of Isolation, that resource units are not to be shared between components is not met. This is because the user does not have explicit control over how the executors are distributed among worker processes.

In Storm, a worker may contain more than one executors. Executors are essentially threads of execution in a worker process. The work of the component are executed by tasks in executors. An executor may contain execute many tasks, but all the tasks within an executor belong to the same component.

In order to facilitate the Isolation requirement, we implement a custom scheduler. The scheduler allows the user to specify the number of workers to be allocated to each component. Additionally, the user can also provide the number of executors and tasks for each component. The scheduler also ensures that each worker process contains executors belonging to a single component. The scheduler operates similarly to even scheduler. The workers are evenly distributed among the worker machines. The executors of a component are evenly distributed among the number of component workers. This fulfills the isolation requirements.

To measure the current state of the topology, the algorithm uses Storm metrics. For the algorithm, we need to collect the input rate, processing rate, dropped rate and output rate. Metrics are collected at the key points of the Storm processing pipeline. Input rate is collected by monitoring worker Receive Thread. The worker receive queue size set to a constant. Once the number of tuples in the receive queue exceed the size of the queue, the tuples are dropped. The dropped metrics counts the number of tuples being dropped. The processed rate is collected at the executor input queue. For the output, we collect two metrics. The emitted rate is the number of tuples directly emitted by the executors and placed in executor send queue. The tuples are then transmitted to the transfer queue, where the worker send thread transmits the tuples to destination workers. This could either be on the same machine or might travel over the network to a different machine. We collect transfer rate as the tuples exiting from the transfer queue.

The reason to collect two different metrics, emitted rate and transferred rate is to ensure that the tuples are not dropped after processing due to network constraints. In our experiments, no tuples were dropped between the emitted queue and transfer queue.

Typically metrics are stored in databases. To simplify our implementation, we use "org.apache.storm.metrics2.reporters.ConsoleStormReporter" metrics reporter. The output is emitted to the worker log. Since each worker is dedicated to a single component, we map the worker to component and accumulate the individual rates of each worker to obtain the metrics for a component. This is a bit tedious; we used scripts to map and collect the metrics numbers.

6.2 Evaluation

In this section, we evaluate our algorithm. We first describe the experimental setup used to execute our tests. We then outline the methodology used to design, execute and collect the results. We then present the results generated by the tests, analyze and contrast

those results with that of Stela.

The experiments are designed to evaluate two objectives. First we analyze the allocations suggested by both the approaches to understand why one approach works better than the other in that particular scenario. We analyze drawbacks of Stela and explain how our approach compensates for those drawbacks by suggesting alternative allocations. And conversely, we also discuss how Stela compensates for the drawbacks of our approach. We ultimately show that in a typical topology our combined algorithm helps in improving the allocations suggested by Stela.

Secondly we evaluate the accuracy of the predictive model used by our approach. We rely heavily on the ability of the model to predict the input and output rates of the components after an allocation. So it is important to establish some confidence in our model.

In line with the objectives experiments for evaluation are divided into two parts. The first set of experiments are conducted by software simulation. We design topologies of varying shapes and sizes. For larger topologies, we vary the input, processing and output rates of each of the components to generate a new set of test topologies. We then execute our algorithm and Stela to obtain allocations for the topology and measure the throughput increase. The number of free resources provided to the algorithm is increased for each iteration.

6.2.1 Software Simulation

For simulation we use the JSON input files as described before. We design topologies with varying shapes, input and processing rate and child ratio, and execute our algorithm. We obtain the allocations returned by Stela and our approach and compare them.

Figure 6.1 shows the structure of a sample topology. The shape of the topology is linear. Output from each component flows to the next and there are no branches. We

present the input, output and max processing rates for each of the components in Table 6.1. Followed by the allocations returned by our approach and Stela.

The table provides the initial configuration of the topology. We execute our algorithm on topology in this state for each iteration. It is to be noted that each iteration is executed in isolation and none of the effects of previous iteration is carried over to next iteration. Column 1 of the table contains the name of the component. In our tests we use a numbers as names . The second column contains the number of resources allocated to the component allocated in the initial configuration. The allocation suggested by Stela or our approach is applied on top of the initial configuration. Column 3 shows the input rate as described in the approach section. The spout does not have have input rate or max processing rate. We only define the output rate for the spout for each resource allocated to it. In our algorithm we consider spout to be congested component and the resources can be used to scale the spout if no other congested components exist. If the input rate is greater than the maximum processing rate (column 5), then only part of the input rate is processed and the rest of the tuples are dropped. The dropped rate can be obtained by inputRate - MaxProcRate. The output rate is the output from the component. The child ratio indicates the children of the compnent and the ratio of the output rate each child component receives. The format used to represent child ratio is {component name : ratio}. In the above case, because this is a linear topology all of the output from component 1 goes to component 2. Hence the ratio of 1.

Table 6.2 column 1 shows the number of resources offered to the algorithm to allocate. Column 2 shows the allocation returned by Stela algorithm and column 4 shows the allocation returned by our approach. The format to indicate the allocation is {component:number of resources allocated}.Column 3 and 5 shows the increase in throughput of topology because of the allocation.

In this experiment, we see both Stela and our approach return identitcal allocations. However the rationale for choice changes are quite different. At the start, both

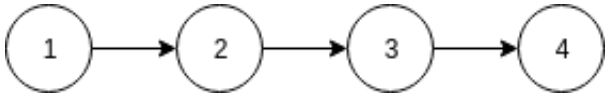

Figure 6.1: Test topology : linear

Table 6.1: Input, output and max processing rates for topology : linear

Name	Res	inputRate	OutputRate	MaxProcRate	childRatio
1	1	-	500	-	{2:1}
2	1	500	400	400	{3:1}
3	1	400	250	250	{4:1}
4	1	250	250	900	

Table 6.2: Allocations and throughput increase for topology : linear

Resources	Stela	Stela Throughput Inc	Our Approach	Our Throughput Inc
1	{3:1}	150	{3:1}	150
2	{2:1, 3:1}	250	{2:1, 3:1}	250
3	{2:1, 3:2}	350	{2:1, 3:2}	350

component 2 and 3 are congested. Stela chooses to decongest 3 as it has the highest ETP, our approach chooses the component 3 as it is the best choice among the congested components that do not cause any congestion downstream. We see how this rationale affects the allocations in the following test cases.

We now present the next topology. Figure 6.2 shows a simple diamond circuit. This operation is common among other topologies as they constitute aggregate step. The input, processing and output rates of the topology is presented in table 6.3. The allocations and the throughput are presented in table 6.4. Similar to the previous topology, both approaches provide similar allocations. Next we would like to discuss the allocation for 1 resource.

Stela calculates the ETP for both the congested components of 2 and 3. The ETP for both the components are the same as they lead to the same sink which is component 4. In this case the selection between the components is arbitrary. In this example we assume Stela makes the better choice and allocate the resource to component 3. Our approach

however calculates the actual throughput increase due to decongestion, and allocates the resource to component 3.

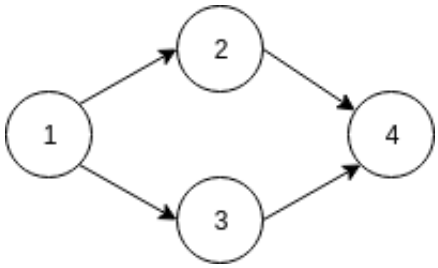

Figure 6.2: Test topology : Diamond

Table 6.3: Input, output and max processing rates for topology : diamond

Name	Res	inputRate	OutputRate	MaxProcRate	childRatio
1	1	-	1000	-	{2:0.5, 3:0.5}
2	1	500	400	400	{4:1}
3	1	500	200	200	{4:1}
4	1	600	600	900	

Table 6.4: Allocations and throughput increase for topology : diamond

Resources	Stela	Stela Throughput Inc	Our Approach	Our Throughput Inc
1	{2:1} or {3:1}	150 or 200	{3:1}	200
2	{2:1, 3:1}	250	{2:1, 3:1}	250
3	{2:1, 3:2}	350	{2:1, 3:2}	350
4	{2:1, 3:2, 4:1}	400	{2:1, 3:2, 4:1}	400

Figure 6.3 presents the simple tree topology. The input, output and processing rates are presented in table 6.5. We use this topology to highlight more of the differences between the allocation strategy of Stela and our approach.

The topology in its initial state has three congested components. Stela calculates ETP for each of the congested components. Component 4 has the highest ETP as the throughput of the subtree of component 4 is 500. However Stela does not recognize that this allocation only increases the throughput only by 100 as shown in table 6.6. Our approach calculates the throughput for each of the congested component after allocation

and chooses to decongest component 3 instead of 4 as it leads to higher throughput increase.

When two resources are provided for allocation, our approach provides a different allocation than when one resource was provided. This is different from Stela as it employs serial allocation technique. The initial state of the topology is the same for each iteration, the order in which the components are decongested also remains same. Stela allocates a resource to component 2 for the same reason described for previous iteration. It then allocates the second resource to 4 based on the new state. The allocations in rest of the iterations are straight forward as both the approaches allocate resources to the remaining congested component 3 till it gets completely decongested.

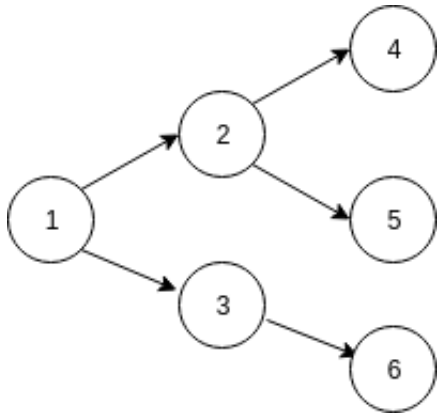

Figure 6.3: Test topology : simple Tree

Table 6.5: Input, output and max processing rates for topology : simple tree

Name	Res	procRate	OutputRate	MaxProcRate	childRatio
1	1	-	2000	-	{2:0.6, 3:0.4}
2	1	1200	800	800	{4:0.75, 5:0.25}
3	1	800	300	300	{6:1}
4	1	600	500	500	
5	1	200	200	900	
6	1	300	300	600	

The topologies designed until now are simple and the allocations show the basic difference in the strategy used by Stela and our approach. It also highlights the scenarios

Table 6.6: Allocations and throughput increase for topology : simple tree

Resources	Stela	Stela Throughput Inc	Our Approach	Our Throughput Inc
1	{4:1}	100	{3:1}	200
2	{2:1, 4:1}	500	{2:1, 4:1}	500
3	{2:1, 3:1, 4:1}	700	{2:1, 3:1, 4:1}	700
4	{2:1, 3:2, 4:1}	900	{2:1, 3:2, 4:1}	900

where Stela or our approach does better. We now design topologies of larger size. Further we manipulate its characteristics like the child ratio and max processing rates of the components while maintaining the same structure of topology. We analyze to see the effects of the allocations mainly to ascertain if a single strategy works well for a topology in all cases.

Figure 6.4 shows the structure of a tree topology. The topology has a total of 17 components and branches of the topology are somewhat symmetrical. The input, output and max processing rates are presented in table 6.7. The throughput increases corresponding to resource allocations are shown in figure 6.5

Table 6.7: Input, output and max processing rates for topology : topology 17

Name	Res	procRate	OutputRate	MaxProcRate	childRatio
1	2	-	2000	-	{2:0.4, 3:0.35, 4:0.25}
2	1	800	1600	900	{5:0.55, 6:0.45}
3	1	700	1400	1100	{7:0.6, 8:0.4}
4	1	500	1200	400	{9:1}
5	1	880	1200	800	{10:0.5, 11:0.5}
6	1	720	1400	700	{12:0.5, 13:0.5}
7	1	840	840	900	{14:1}
8	1	560	500	500	{15:1}
9	1	1200	1000	1000	{16:0.6, 17:0.4}
10	1	600	400	400	
11	1	600	300	300	
12	1	700	700	900	
13	1	700	700	800	
14	1	840	840	900	
15	1	500	500	700	
16	1	600	500	500	
17	1	800	800	900	

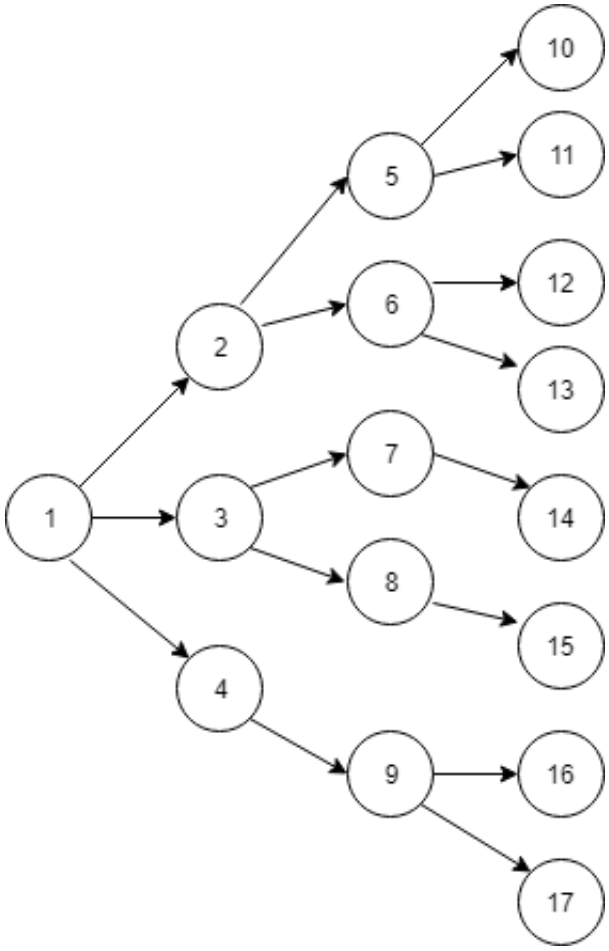

Figure 6.4: Test topology : topology 17

Table 6.8: Allocations and throughput increase for topology : topology 17

Resources	Stela	Stela Throughput Inc	Our Approach	Our Throughput Inc
1	{6:1}	40	{16:1}	500
2	{6:1, 9:1}	140	{16:2}	700
3	{6:1, 9:2, 17:1}	200	{11:1, 16:2}	1000

In table 6.8 we provide allocations returned by our approach and Stela for the first three iterations. In the first iteration, we see component 6 has the highest ETP score. Stela accordingly allocates the resource to component 6. However the children of component 6 are already processing input close to the max processing rate. Allocating resource to 6 congests both its children, thus the throughput increase is not achieved.

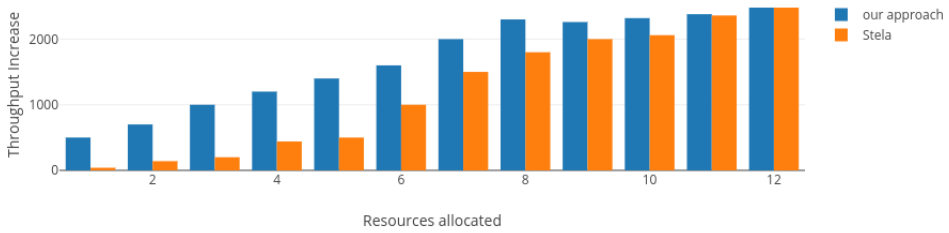

Figure 6.5: Throughput increase comparison for Test topology : topology 17

In the next iteration the algorithm is provided with two resources. Because of the serial allocation strategy, Stela allocates the first resource to component 6 as it is operating in the same premise as the first iteration. For allocating the second resource ETP for each component is recalculated. In this case, component 9 is found to have a higher ETP than any of the children of component 6. Unfortunately in this case component 9 suffers from similar problem as component 6. This behavior is similar to the earlier case with the simple tree.

If any early choice made by Stela is bad, then the same choice is made for every iteration. These resources allocated are a liability until later when its children are decongested. In the third iteration, effectively Stela is working with just one resource. This is a major drawback of Stela and our approach addresses it by searching for other allocations that can be made unless we have enough resources to decongest the subtree of component 6.

For the next experiment we use the same topology as shown in figure 6.4. We swap the child rates for the topology presented in table 6.7 essentially redirecting output of child 1 to child 2, and vice versa. As an example for component 1, the child rate after the swap will be {2:0.25, 3:0.35, 4:0.4} and for component 2 will be {5:0.45, 6:0.55}. The maximum processing rates and the input to output ratio is retained for the component. We present the full table 6.9. The input rate can be calculated by multiplying the output

rate of the parent component with the child ratio. The input rate for component 2 is calculated as output rate of its parent component 1 and the child ratio for 2, i.e. 2000 * 0.25 = 500. Rest of the table is updated similarly. We present throughput increases for each iteration in figure 6.6.

Table 6.9: Input, output and max processing rates for topology : topology 17

Name	Res	procRate	OutputRate	MaxProcRate	childRatio
1	2	-	2000	-	{2:0.25, 3:0.35, 4:0.4}
2	1	500	1000	900	{5:0.45, 6:0.55}
3	1	700	1400	1100	{7:0.4, 8:0.6}
4	1	800	1200	400	{9:1}
5	1	450	675	800	{10:0.5, 11:0.5}
6	1	550	1100	700	{12:0.5, 13:0.5}
7	1	560	560	900	{14:1}
8	1	840	500	500	{15:1}
9	1	1200	1000	1000	{16:0.4, 17:0.6}
10	1	337	337	400	
11	1	337	300	300	
12	1	550	550	900	
13	1	550	550	800	
14	1	560	560	900	
15	1	500	500	700	
16	1	600	500	500	
17	1	800	800	900	

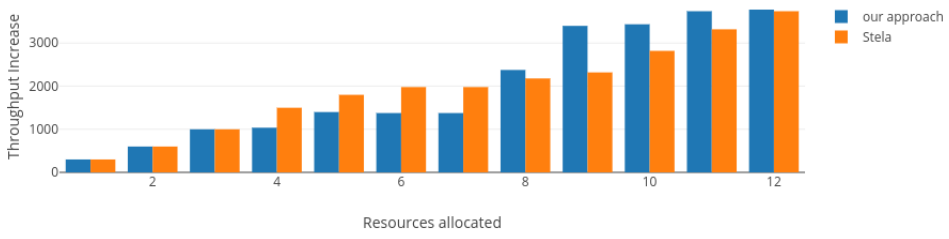

Figure 6.6: Throughput increase comparison for topology 17 with child ratios swapped

In this case, we see the performance of Stela is much better than the previous case shown in figure 6.5. The changes in the child ratio affects the input rate of each

component. It also affects the throughput under each component which is essential to calculate ETP. As a result Stela picks a different set of components to decongest. It is important to see that the changes in the input rate on the components can have a significant impact on the performance of the approaches.

For the next experiment we use the topology shown in figure 6.4. In this case we retain the child ratio as presented in table 6.7 and increase the capacity of some of the components. As the topology resembles the tree structure, we define the level of a component as 1 + the number of connections between the component and the root. We follow the rule to increment maximum processing rate by steps of 200 for each level after 2. Thus for the components in level 2 (2,3,4) are maximum processing rate is incremented by 0, level 3 components (5,6,7,8,9) by 200, level 4 components by 400 and so on. We present the results in figure 6.7

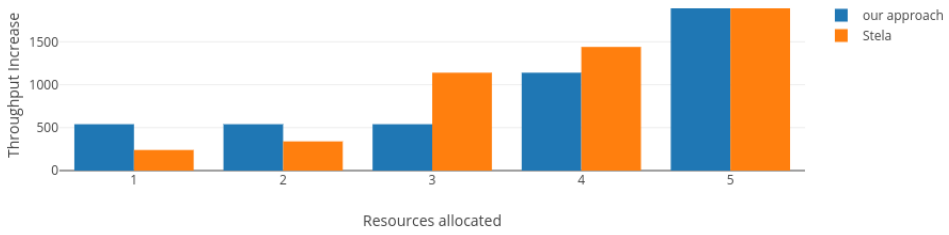

Figure 6.7: Throughput increase comparison for topology 17 step capacity increased

The changes in capacity affects the number of congested components and also makes it easier for the algorithm to allocate resources to lower level components with more confidence. The results and the rationale are similar to the cases discussed in prior experiments.

We make use of the topology shown in figure 6.4 one last time for our experiment. Here we retain the modifications made for previous case in figure 6.7 and swap the input ratios as we did for the case figure 6.6. We present the results in figure 6.8. The test again

shows that the changes in the processing rates of the component and the input rate can have significant impact on the output.

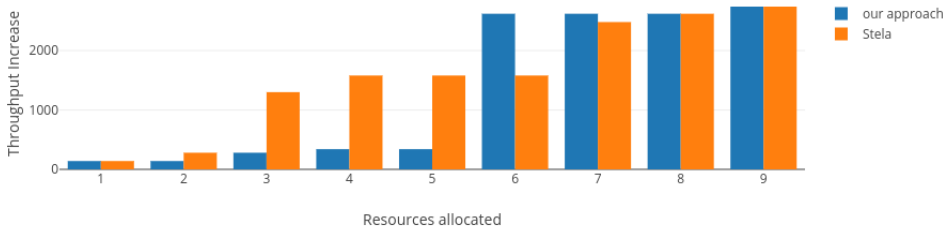

Figure 6.8: Throughput increase comparison for topology 17 step capacity increased child ratio swapped

For the next set of experiments we will use a different topology. The structure of the topology is presented in figure 6.9. The topology is similar to the tree topology. The top part of the topology resembles a skewed tree. Table 6.10 shows the initial state of the topology. We will modify the processing rate and child ratios of the components like in the case of the previous topology. We present the results and summarize our findings later.

In the first case, we get the allocations for the topology for the state presented in table 6.10. Figure 6.10 shows the throughput increase for the allocations. We see in this case Stela provides better allocations for the topology. We will now swap the child ratios as described earlier. The results are presented in figure 6.11. In this case we see our approach doing better than Stela. Further we modify the maximum processing rate of components by incrementing them in steps of 100 for every level beyond 2. Processing rates of level 2 components (4,5,6,7) are incremented by 100, level 3 by 200 and so on. We obtain the allocations for the modified topology and present the results in figure 6.12. We then swap the child ratios of the topology modified for figure 6.12 and present the resulting throughput increase for allocations in 6.13. The results for this topology is similar to those obtained from topology 17. We present our analysis of the results in the

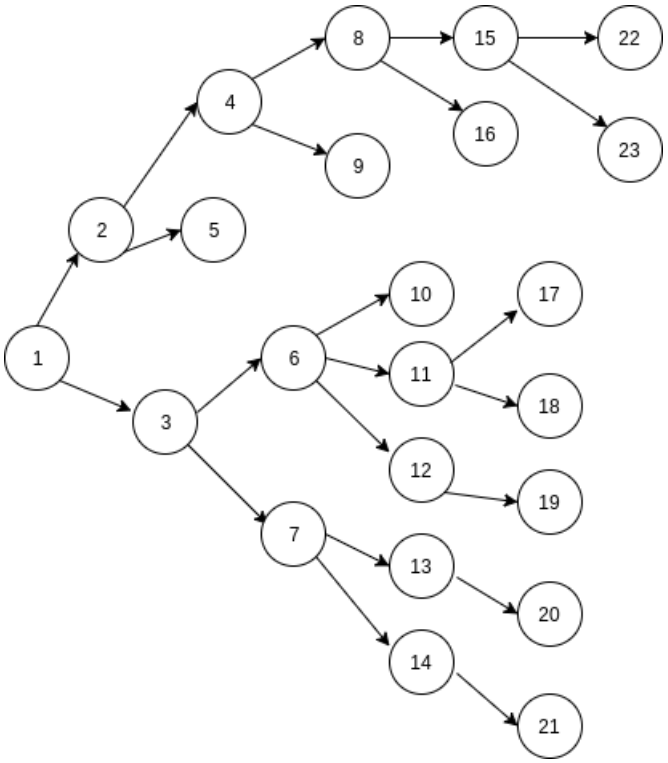

Figure 6.9: Test topology : testMix topology

next section.

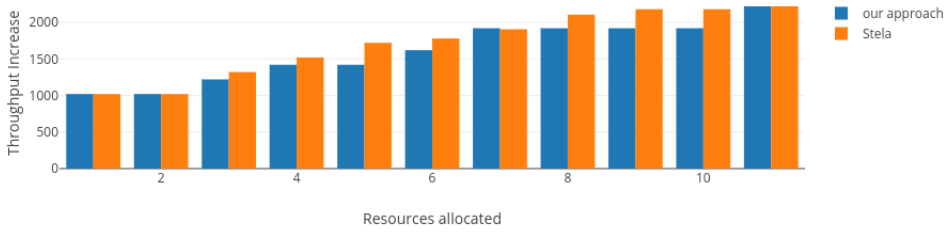

Figure 6.10: Throughput increase comparison for testMix topology

6.2.2 Analysis

In this section we summarize the observations that can be made from the results of the experiments. In our experiments, we show the conditions under which one approach

Table 6.10: Input, output and max processing rates for topology : mix topology

Name	Res	procRate	OutputRate	MaxProcRate	childRatio
1	2	2	2000	1	{2:0.4, 3:0.6}
2	2	1000	2000	1500	{4:0.6,5:0.4}
3	1	1200	2400	1500	{6:0.6, 4:0.4}
4	1	800	800	800	{8:0.75,9:0.25}
5	1	600	600	600	
6	1	600	600	600	{10:0.4,11:0.35,12:0.25}
7	1	500	1000	500	{13:0.6,14:0.4}
8	1	600	600	700	{15:0.75,16:0.25}
9	1	200	200	400	
10	1	240	240	600	
11	1	210	420	800	{17:0.5,18:0.5}
12	1	150	150	600	{19:1}
13	1	600	600	700	{20:1}
14	1	400	400	600	{21:1}
15	1	450	450	60	{22:0.75,23:0.25}
16	1	150	150	400	
17	1	210	420	500	
18	1	210	210	500	
19	1	150	150	500	
20	1	600	600	900	
21	1	400	400	600	
22	1	337	337	700	
23	1	112	112	500	

provides better allocations than the other. Stela allocates resources in a serial manner. When Stela decongests a component and if the components in the subtree is already processing close to their maximum processing rate then the subtree gets partially congested. The throughput increase is determined by the throughput produced by this partially decongested subtree. If most of the output is skewed towards fewer paths in the subtree and if these high yielding paths are not congested, then the throughput increase is significant. In these cases Stela performs better. If enough resources are not available to decongest the subtree, then the overall allocation will have subpar increase in throughput. Because the choices made by Stela are greedy, the allocations have high risk with high rewards in favorable circumstances.

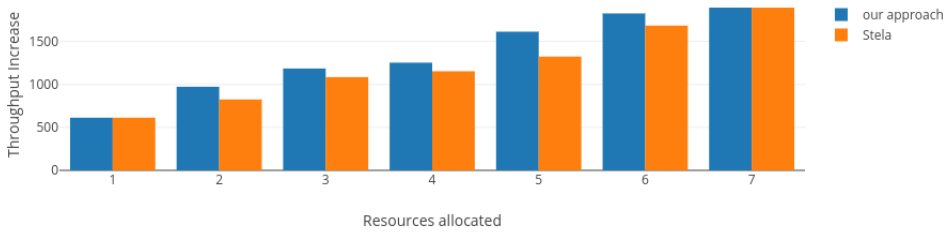

Figure 6.11: Throughput increase comparison for testMix with child ratio swapped : testMixRev topology

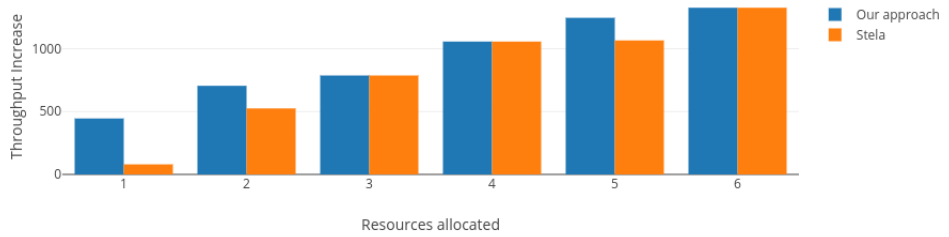

Figure 6.12: Throughput increase comparison for testMix with step capacity increase : testMixStep topology

Our algorithm avoids selecting a components for decongestion if not enough resources are available to decongest the subtree. The degree to which our algorithm does better in these cases depends on existence of alternate congested components whose subtree that can be fully decongested. The throughput increase from these fully decongested subtrees is usually higher than the partially decongested subtree of a high ETP component.

The conditions that favor one approach over the other is dependent on the shape of the topology, the processing rates and output rates of the component, the child ratio and the number of resources provided. We see in case of topology testMixRev using base configuration (table 6.10) the conditions favor Stela allocation (figure 6.10). However

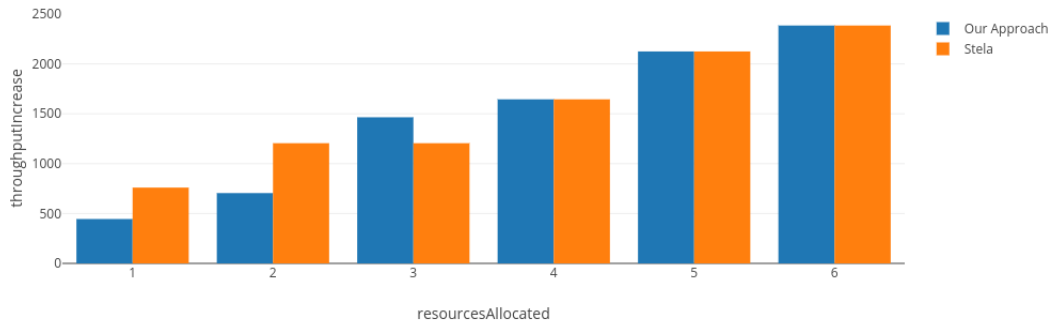

Figure 6.13: Throughput increase comparison for testMix with step capacity increase and child ratio swapped : testMixStepRev topology

when we swap the child ratio Stela returns better allocation for few cases, and our approach returns better allocation for others. (figure 6.11). We can see similar changes when we change the maximum processing rates of the components (figure 6.12). We infer that it is hard to predict which of the algorithms provide better allocations for a given topology given its initial state, as changes to child ratio or processing rate could shift the topology from favoring one approach to another.

We also observe that both algorithms work well individually, but neither of the algorithms work better than other in all circumstances. We have discussed the drawbacks of Stela before. Lets consider a scenario where our approach does poorly. In a subtree if most of the output is concentrated on few paths, then it is prudent to allocate resources to these paths to produce maximum output for each resource allocated. The rest of the subtree that has low contribution to the throughput can be deallocated later at lower priority. But because of our policy we do not consider this subtree until we have sufficient resources to completely decongest it. Even when we consider it, we have to make the allocation to even the low yielding paths in order to decongest the whole subtree. These resources could be better used elsewhere. This diminishes the effectiveness of our approach and these are exactly the scenarios where Stela does well.

Finally we show that the combined algorithms is an improvement over Stela because of the complementary approach used by our algorithm and Stela. Figure 6.14 and Figure 6.15 shows the throughput increase of the combined algorithm over Stela. The graph is a compilation of the results from previous experiments. The degree to which the the combined algorithm is better than Stela depends on the state of the topology as described before.

6.2.3 Validation in Testbed

Anvil is the Holland Computing Center's cloud computing resource at UNL. It is based on the OpenStack software, allowing the user to provision virtual machines of varying computing memory, cpu cores and disk size. For our experiments, we used machines with 2 CPU cores, 4GB of RAM, and 80GB of Hard Disk space. The machines are imaged with CentOS 7.4 XFCE. cgroups are configured to reserve 10% of the total CPU for each process created under it. Storm is configured to use the cgroup, and thus each worker process created by storm is guaranteed 10% of cpu. Each machine is configured with 7 slots, so a maximum of 7 worker process can run on it.

The experimental setup contains 8 machines. One machine is dedicated to be Storm server hosting Nimbus. It does not host any worker process. The rest of the 7 machines are configured to execute worker processes as described above. The setup was mainly used to execute larger topologies, with the main aim to validate that cgroups allow us to estimate the input and output rates within reasonable accuracy. The larger number of worker process gets distributed among the machines forcing some components to direct its output over the network to reach its intended child component. We execute a test topology with 10 components as shown in figure 6.16 with the configuration shown in table 6.11.

The test uses the modified version of Exclamation Topology. Exclamation topology is one of the sample topologies provided by Storm starter kit. The topology has three

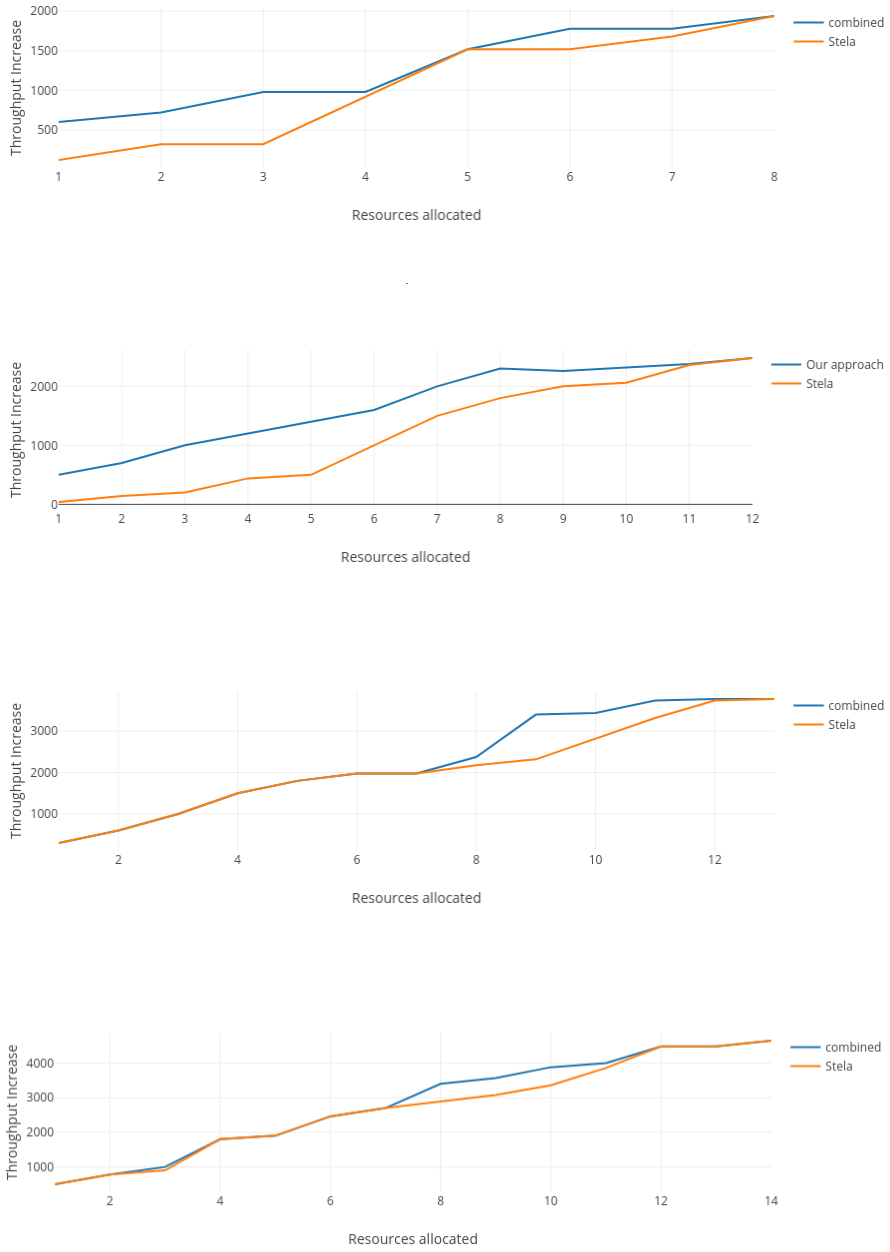

Figure 6.14: Throughput increase comparison for combined algorithm and Stela

components S1, C1 and C2. The spout S1 generates a random word, passes it onto its child component C1. The C1 adds a Exclamation character to the word passed to it by S1. C1 passes the modified word to its child component C2 which does the same and outputs the word.

Table 6.11: Input, output and max processing rates for topology : topology 10

Name	Res	procRate	OutputRate	MaxProcRate	childRatio
1	2	-	16000	-	{2:0.44, 3:0.33, 4:0.22}
2	1	7040	4700	4700	{5:0.6, 6:0.4}
3	1	5280	3500	3500	{7:0.6, 8:0.4}
4	1	3520	2500	2500	{9:0.6, 10:0.4}
5	1	4224	2800	2800	
6	1	1880	1880	2100	
7	1	2100	2100	2700	
8	1	1400	1400	3800	
9	1	1500	1500	3800	
10	1	1000	1000	3600	

The test modifies the components C1 and C2. To simulate components with various cpu loads, we add a spin loop in the component. A series of components are prepared with varying cpu load and profiled for maximum input and output rates. The number of children and the output ratio to children are modified as needed to build more complex topologies.

For testing, the components of topology is profiled to collect the maximum processing and output rates. A JSON file with all the components and its rates is created and provided to the algorithm. The program is executed for the input file with different number of free resources to be allocated. The allocations and the throughput is collected from the program. The topology is executed again with the allocation provided by the program, and output is noted from the test bench. The predicted throughput from the program is compared with the values obtained from the test bench.

The test on the test bench is executed three times for each case, and the average of the values are considered. Each test is executed for 30 minutes, and the average input, output rates of the last 15 minutes is considered.

We simulate the topology to obtain allocations for our approach and Stela. We present the comparison between the Stela and our approach in terms of the throughput increase (figure 6.17) as we have done in software simulation section. We also show that

the combined algorithm does better than Stela (figure 6.18).

We apply the allocations obtained by software simulation for both our approach and Stela on the test bed. We measure the throughput from the test bed and compare it with the throughput predicted by simulation. The results are shown in Figure 6.19

By testing a simulated topology on the test bed, we see that the predicted throughput is fairly close to the measured throughput. We further measure the processing rate of each during the test. For most cases the measured value stays consistent and within 10% difference from the estimated value. This shows that the model can be used in a reliable way to estimate the throughput of a allocation.

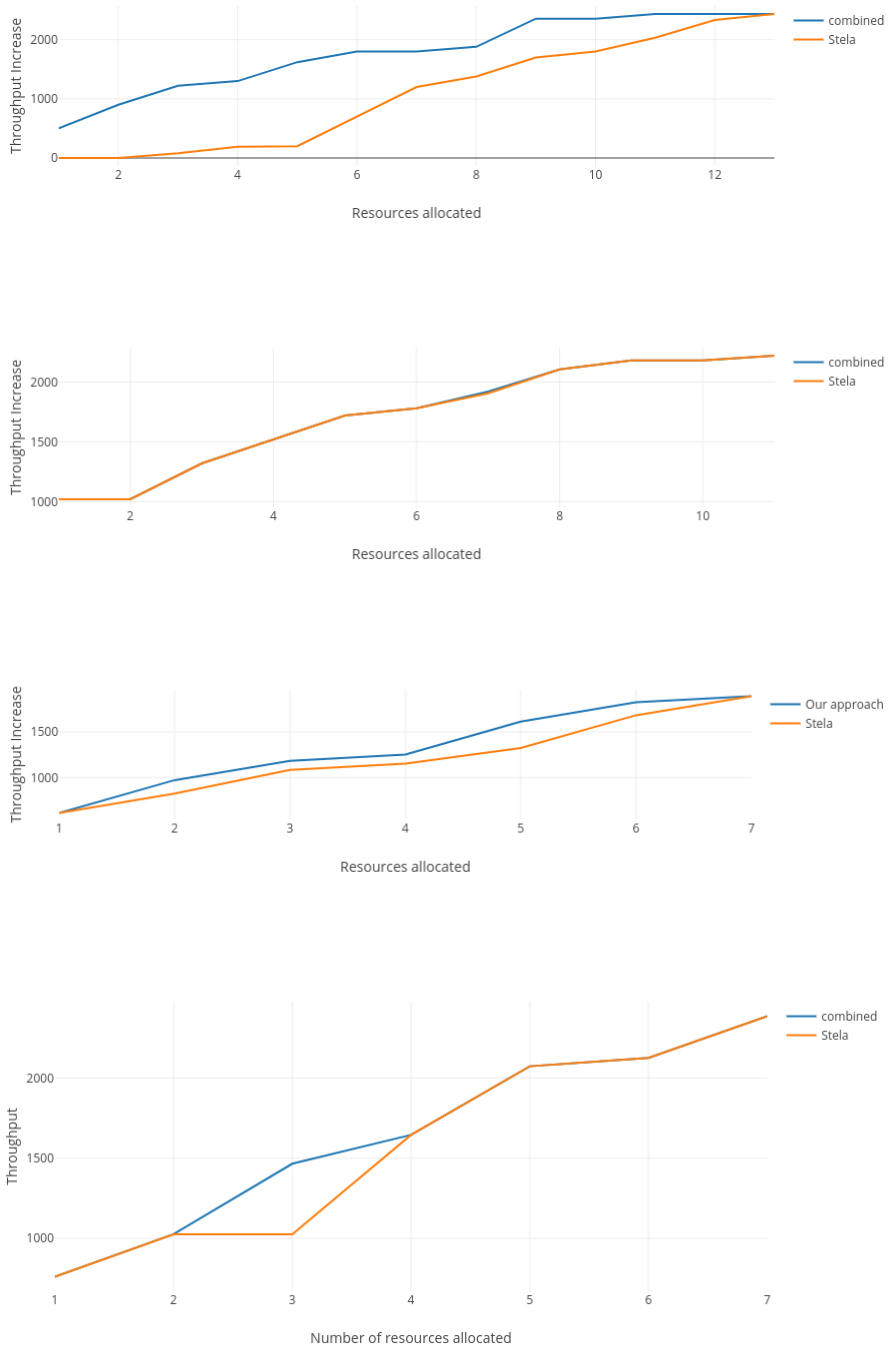

Figure 6.15: Throughput increase comparison for combined algorithm and Stela

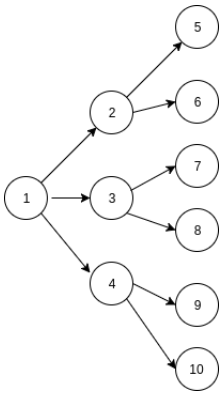

Figure 6.16: Test topology : topology 10

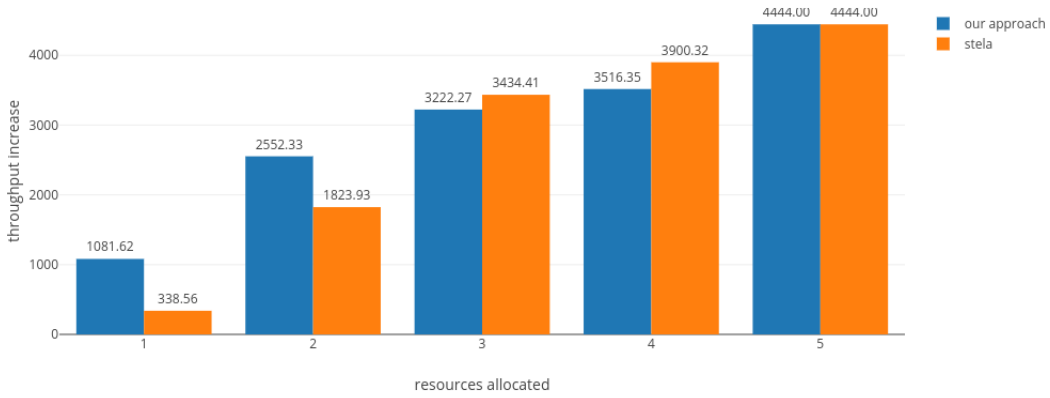

Figure 6.17: Throughput increase comparison Stela and our approach : topology 10

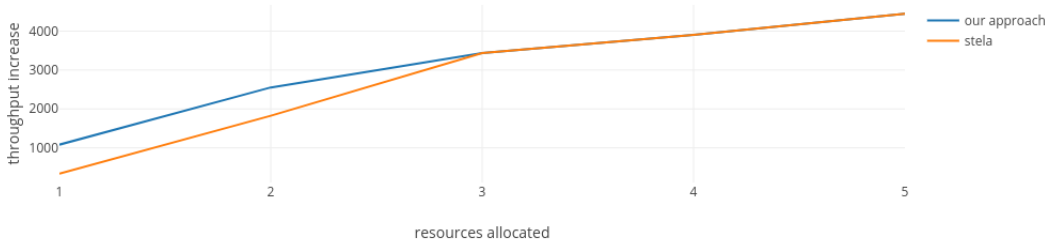

Figure 6.18: Throughput increase comparison Stela and Combined : topology 10

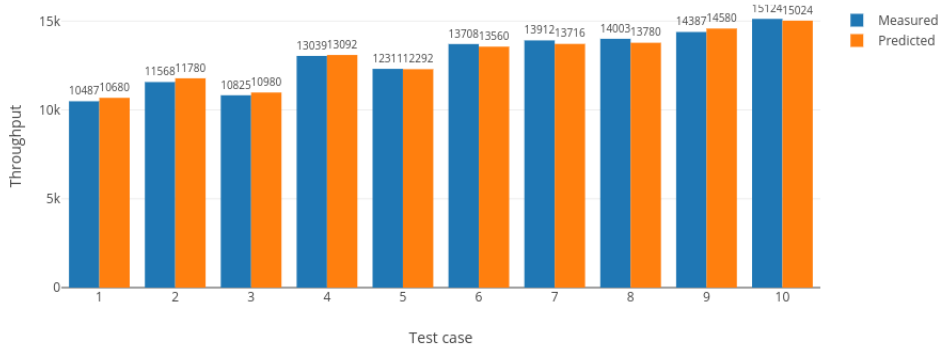

Figure 6.19: Throughput comparison measured and estimated

Chapter 7

Future Work

In this section, we identify some of the areas that can be improved in the algorithm, and some related approaches to ours that can be used to solve other problems.

Firstly in our testing, we see that the processing and output rates remain reasonably constant for a component when executed in a worker process created under cgroup. The input rate to the worker process was somewhat constrained within a range. When the input rate increases, more of the processing power would be used to drop the tuples instead of processing them. This can affect the processing rate of the worker. A more detailed analysis of the worker process created under cgroup could help create better mathematical models.

An improvement to the process could be to profile the components automatically at runtime. Profiling the components in general is tedious. If the characteristics of the component changes over time, then the process cannot rely on the existing processing and output rates to produce good allocations. One approach to profile the components would be to monitor the amount of cpu used by the process, and then correlate it with the number of tuples being processed in unit time. This can be used to estimate the maximum processing rate the worker can support.

Our algorithm uses a policy to avoid introducing or worsening congestion. If an

allocation does introduce any congestion in its subtree, it is rejected. If most of the throughput is produced by fewer paths, our approach won't be able to consider the subtree until we have enough resources to decongest all of the components in the subtree. This policy can be too restrictive, and strategies for partially decongesting the subtree need to be explored further.

Chapter 8

Conclusion

In our work we make a case to offer Storm as a service, as it is a natural progression from the way storm in IaaS or PaaS flavors. We use cgroups to allow users to procure resources smaller than a machine. We use a design where a worker process executes tasks of a single component. We implement a new scheduler to allow users to control the allocations of the resources to the components in their workflow. This design allows us to estimate the processing rate of the components for a given input rate and for the number of resources allocated. We build a model to estimate the state of the topology for a given allocation. We build a scheduler based on Stela to scale out the workflow. We discuss the weakness of Stela, and design a new algorithm using a predictive model. Further we test our approach by testing it against different topologies and show that our algorithm improves upon the the allocation provided by Stela in several cases. Finally we show that our predictive model is reasonably accurate in estimating the throughput of the topology and verify it against the topology by executing on a physical test bed.

References

[1] Michael G. Noll. Understanding the internal message buffers of storm. https://www.michael-noll.com/blog/2013/06/21/understanding-storm-internal-message-buffers/, June 21 2013.

[2] Louis Columbus. Roundup of cloud computing forecasts, 2017. https://www.forbes.com/sites/louiscolumbus/2017/04/29/roundup-of-cloud-computing-forecasts-2017/#63be5dd831e8, April 29 2017.

[3] Apache hadoop. http://hadoop.apache.org/.

[4] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 147–156, New York, NY, USA, 2014. ACM.

[5] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 239–250, New York, NY, USA, 2015. ACM.

[6] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[7] Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: Fault-tolerant stream processing at internet scale. In *Very Large Data Bases*, pages 734–746, 2013.

[8] Apache zookeeper. https://zookeeper.apache.org.

[9] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *2010 IEEE International Conference on Data Mining Workshops*, pages 170–177, Dec 2010.

[10] X. Liu and R. Buyya. Performance-oriented deployment of streaming applications on cloud. *IEEE Transactions on Big Data*, pages 1–1, 2018.

[11] Get started with apache storm on hdinsight using the storm-starter examples. https://docs.microsoft.com/en-us/azure/hdinsight/storm/apache-storm-tutorial-get-started-linux.

[12] Sebastian Lambert. 2018 saas industry market report: Key global trends & growth forecasts. https://financesonline.com/2018-saas-industry-market-report-key-global-trends-growth-forecasts/.

[13] Apache storm. https://github.com/apache/storm.

[14] L. Xu, B. Peng, and I. Gupta. Stela: Enabling stream processing systems to scale-in and scale-out on-demand. In *2016 IEEE International Conference on Cloud Engineering (IC2E)*, pages 22–31, April 2016.

[15] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[16] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 18:1–18:17, New York, NY, USA, 2015. ACM.

[17] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.

[18] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[19] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 285–300, Broomfield, CO, 2014. USENIX Association.

[20] Narayanan A. Tupperware: containerized deployment at facebook. http://www.slideshare.net/dotCloud/tupperware-containerized-deployment-at-facebook, June 2014.

[21] J. Cervino, E. Kalyvianaki, J. Salvachua, and P. Pietzuch. Adaptive provisioning of stream processing systems in the cloud. In *2012 IEEE 28th International Conference on Data Engineering Workshops*, pages 295–301, April 2012.

[22] J. Xu, Z. Chen, J. Tang, and S. Su. T-storm: Traffic-aware online scheduling in storm. In *2014 IEEE 34th International Conference on Distributed Computing Systems*, pages 535–544, June 2014.

[23] Leila Eskandari, Jason Mair, Zhiyi Huang, and David Eyers. A topology and traffic aware two-level scheduler for stream processing systems in a heterogeneous cluster. In Dora B. Heras, Luc Bougé, Gabriele Mencagli, Emmanuel Jeannot, Rizos Sakellariou, Rosa M. Badia, Jorge G. Barbosa, Laura Ricci, Stephen L. Scott, Stefan Lankes, and Josef Weidendorfer, editors, *Euro-Par 2017: Parallel Processing Workshops*, pages 68–79, Cham, 2018. Springer International Publishing.

[24] George Karypis and Vipin Kumar. A fast and highly quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1999.

[25] Leila Eskandari, Jason Mair, Zhiyi Huang, and David Eyers. Iterative scheduling for distributed stream processing systems. In *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems*, DEBS '18, pages 234–237, New York, NY, USA, 2018. ACM.

[26] M. Lee, M. Lee, S. J. Hur, and I. Kim. Load adaptive distributed stream processing system for explosive stream data. In *2015 17th International Conference on Advanced Communication Technology (ICACT)*, pages 753–757, July 2015.

[27] Anshu Shukla and Yogesh L. Simmhan. Model-driven scheduling for distributed stream processing systems. *J. Parallel Distrib. Comput.*, 117:98–114, 2018.

[28] Y. Wang, Z. Tari, M. R. HoseinyFarahabady, and A. Y. Zomaya. Model-based scheduling for stream processing systems. In *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 215–222, Dec 2017.

[29] Christoph Hochreiner, Michael VÃűgler, Stefan Schulte, and Schahram Dustdar. Cost-efficient enactment of stream processing topologies. *PeerJ Computer Science*, 3:e141, December 2017.

[30] Christoph Hochreiner. Visp testbed - a toolkit for modeling and evaluating resource provisioning algorithms for stream processing applications. In *ZEUS*, 2017.

[31] Y. Zhai and W. Xu. Efficient bottleneck detection in stream process system using fuzzy logic model. In *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 438–445, March 2017.

[32] J. Lin, M. Lee, I. C. Yu, and E. B. Johnsen. Modeling and simulation of spark streaming. In *2018 IEEE 32nd International Conference on Advanced Information Networking and Applications (AINA)*, pages 407–413, May 2018.

[33] Boyang Peng, Mohammad Hosseini, Zhihao Hong, Reza Farivar, and Roy Campbell. R-storm: Resource-aware scheduling in storm. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, pages 149–161, New York, NY, USA, 2015. ACM.

[34] Introduction to control groups (cgroups). https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/ch01.