Autoscaling High-Throughput Workloads on Container Orchestrators

Chao Zheng
University of Notre Dame
charleszheng44@gmail.com

Nathaniel Kremer-Herman
University of Notre Dame
nkremerh@nd.edu

Tim Shaffer
University of Notre Dame
tshaffe1@nd.edu

Douglas Thain
University of Notre Dame
dthain@nd.edu

***Abstract*—High-throughput computing (HTC) workloads seek to complete as many jobs as possible over a long period of time. Such workloads require efficient execution of many parallel jobs and can occupy a large number of resources for a long time. As a result, full utilization is the normal state of an HTC facility. The widespread use of container orchestrators eases the deployment of HTC frameworks across different platforms, which also provides an opportunity to scale up HTC workloads with almost infinite resources on the public cloud. However, the autoscaling mechanisms of container orchestrators are primarily designed to support latency-sensitive microservices, and result in unexpected behavior when presented with HTC workloads. In this paper, we design a feedback autoscaler, High Throughput Autoscaler (HTA), that leverages the unique characteristics of the HTC workload to autoscales the resource pools used by HTC workloads on container orchestrators. HTA takes into account a reference input, the real-time status of the jobs' queue, as well as two feedback inputs, resource consumption of jobs, and the resource initialization time of the container orchestrator. We implement HTA using the Makeflow workload manager, Work Queue job scheduler, and the Kubernetes cluster manager. We evaluate its performance on both CPU-bound and IO-bound workloads. The evaluation results show that, by using HTA, we improve resource utilization by 5.6× with a slight increase in execution time (about 15%) for a CPU-bound workload, and shorten the workload execution time by up to 3.65× for an IO-bound workload.**

***Index Terms*—Kubernetes, Autoscaling, High-throughput Workloads**

I. Introduction

High-throughput computing (HTC) workloads consisting of large numbers of parallel jobs often require a tremendous amount of computing resources for a long time. As a result, HTC facilities that have to execute many HTC workloads ordinarily operate at full utilization of limited resources. On the other hand, public cloud providers, like Amazon AWS [1] and Google GCP [2], render rapidly evolving infrastructure and almost infinite computing resources. Therefore, migrating HTC workloads to the public cloud may be a solution to the scarcity of local resources.

Existing solutions for running HTC workloads provided by the major cloud providers [3]–[5] include four steps: i) determining compute, network and storage requirements for workloads; ii) preserving computing instances on the cloud; iii) building virtual clusters atop of these virtual nodes; iv) setting up the HTC frameworks on the cluster and executing workloads. In this process, container orchestrators, like Kubernetes [6], have been widely adopted to build elastic virtual clusters on clouds.

However, container orchestrators are designed for **latency-sensitive** workloads, which consist of large fleets of services deployed to meet the varying loads imposed by external users, such as web servers, video conferencing, and online games. The performance objective of these workloads is to minimize the response time observed by external users interacting with the system. In contrast, HTC workloads consist of large numbers of discrete parallel jobs that start and end, such as genome sequence alignment, molecular dynamics simulation, and parameter space exploration. The performance objective of HTC workloads is to maximize the amount of work completed over a long period of time by using resources efficiently. The optimization goals of these two workload categories are so distinct that resource optimizations that apply to one do not work with the other.

One of the public cloud platform's critical characteristics is the **pay-as-you-go** pricing model, which requires the platform to **autoscale** a pool of resources to meet the needs of a given workload. However, as these systems are generally designed with latency-sensitive services in mind, regardless of what virtualization technologies (i.e., virtual machine or container) they use, most of the autoscaling mechanisms [7]–[9] they adopted are based on the application response time and resource metrics set by users. When any metric is too high/low, the autoscaler increases/decrease the resource pool, which has the effect of adjusting the metric to the desired degree. However, such strategies do not work for HTC workloads because high resource utilization is the ordinary case, and increasing the allocated pool only allows more jobs to run. An open challenge of running HTC workloads on container orchestrators is **how to autoscale resource pools accurately**.

Figure 1 shows the three essential components of an HTC system: a workflow manager, a job scheduler, and a cluster manager. A workflow manager is a user-facing tool that describes the overall structure of a workload, handles the job and data dependencies between components, and dispatches ready jobs to the underlying system. Examples include Kepler [10], Pegasus [11], and Galaxy [12]. A job scheduler handles the problem of assigning ready jobs to execution sites by prioritizing work, matching available resources, and handling runtime failures. Examples include Spark [13], Work Queue [14], Sparrow [15], or Spring Batch [16]. A cluster

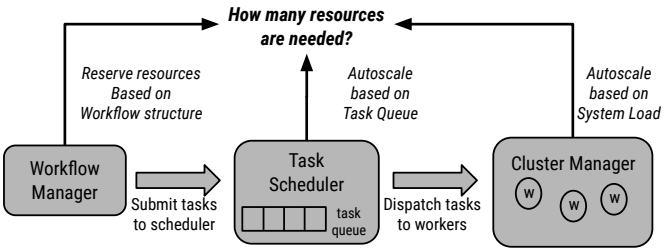


Fig. 1: Resource Provisioning for HTC Workload

manager abstracts resources of physical nodes into a unified resource pool of containers and virtual machines shared among multiple users, such as Apache YARN [17], Apache Mesos [18], or Kubernetes [6].

From the perspective of each component, there are three broad approaches to autoscaling (figure 1). First, one can let the workflow manager analyze the structure of the workload and reserve resources statically. This option usually needs to know the job resource requirements and relies on domain-specific prediction models [19]. Second, one can monitor the job queue's status through the job scheduling framework and resize resource pools dynamically. This option does not require advanced knowledge of workloads, but it does need to know the resource initialization time to prevent resources from over or under-provisioning [20]. Third, one can use local node metrics (e.g., CPU, storage, or network usage) and resize resource pools as demand changes. This approach can only be reactive and might miss the peak needs of an HTC workload.

We argue that a better mechanism that can accurately resize the resource pool for HTC workloads requires information from all three components, and a new middleware that has control over cluster resources based on the status of running workloads is necessary. In this paper, we refine the autoscaling problem into two sub-problems, i) what is the size of an essential resource unit? ii) how many resource units are required by the target workload? We resolve the first problem by comparing various system settings and observe that if we align each resource unit with an independent node and set the parallelism (i.e., the number of parallel jobs per node) correctly, we can achieve the highest workload throughput and resource utilization. To get the correct parallel degree, we leverage the fact that most HTC workloads consist of different categories of jobs with internal similarity. Jobs of the same stage can run in parallel and are usually copies of the same program with different input datasets. By collecting the resource usage of complete jobs, we can estimate the resource requirements of jobs belonging to the same stage and assign an appropriate number of jobs to a node. For the second problem, we design a well-informed feedback autoscaler which considers the resource consumption of completed jobs, the real-time status of the jobs' queue, and the resource initialization time reported by the cluster manager.

As a case study, we compose a software stack with Makeflow [21] as the workflow manager, Work Queue [14] as the job scheduling framework and Kubernetes [6] as the cluster manager. We implement the autoscaling mechanism in a middleware called High-Throughput Autoscaler (HTA), which sets up the Work Queue framework on Kubernetes and controls the life-cycles of deployment units based on the progress of running workloads.

We compare HTA to the default Horizontal Pod Autoscaler (HPA) by running a bioinformatics workload and an I/O bound synthetic workload. The experimental results show that HTA can increase the cluster resource utilization by **5.6**× with only a slight increase in execution time (around 15%) for the bioinformatics workload, and shorten the workload execution time by up to **3.6**× for the I/O bound workload.

The rest of the paper is organized as followed. We introduce the background knowledge in Section II. We describe and refine the autoscaling problem in Section III, propose solutions in Section IV, and describe the implementation details of HTA in Section V. The evaluation setup and results are presented in Section VI. After discussing related work in Secion VII, we conclude in Section VIII.

II. Background

There exist many tools to run HTC workloads on the cloud. We choose three of them to compose a software stack that uses the Makeflow as the workload manager, the Work Qeue as the job scheduler, and the Kubernetes as the cluster manager.

A. Makeflow

An HTC workload is often represented as a Directed Acyclic Graph (DAG), where the nodes of the graph are jobs to execute, and the edges of the graph represented dependencies between jobs. Makeflow [21] is a workflow manager for describing such DAGs. Makeflow's syntax is similar to that of GNU Make, which allows users to describe any workload expressible in a Directed Acyclic Graph (DAG) structure. After the user creates the workload description, Makeflow parses the description and generates an in-memory representation of the workload's DAG structure and parcels it out to an underlying execution framework.

B. Work Queue

Work Queue [14] is a framework for building large-scale master-worker applications that spawn workers across different cloud platforms. Each master program has a worker pool consisting of a set of connected workers. The size of the worker pool varies dynamically with the available computing resources. During runtime, the master finds available workers and assigns jobs to them, and then the worker will arrange data transfer and execute each job it receives. A worker may run multiple jobs simultaneously, as long as the sum of their declared resources (e.g., cores, memory, disk) does not exceed the machine's capacity.

C. Kubernetes

Kubernetes [6] is a container orchestration tool developed by Google, which allows the developer to manage distributed applications hosted in containers. Kubernetes allows users to describe resources using different objects. In this paper, we use three of them, i) a Pod, which is the primary deployment unit and a disposable object which might fail or restart; ii) a StatefulSet, which contains a set of pods and each of them has a unique and sticky identity; iii) a Service, which defines the network protocol for accessing the micro-services hosted on a set of pods.

For deploying Work Queue workers on Kubernetes, several configurations exist depending on which deployment unit we choose to manage worker containers. We anticipate that if the cluster needs to be shrunk, some workers will be removed. If we remove workers by deleting the deployment unit wrapping them, worker containers and jobs running on them will be interrupted. To avoid interrupting worker containers, rather than using advanced deployment units to control the life-cycle of worker containers, we align each worker container with an independent pod and manage the life-cycle of each worker container directly through the Work Queue.

III. Problems

We divide the autoscaling problem into two subproblems: what is the size of a worker-pod (section III-A) and how many total worker-pods are required (section III-B)?

A. Size of a worker-pod

HTC workloads are typically composed of loosely coupled jobs that can be executed concurrently. However, without knowing the resource requirements of each job, assigning multiple resource-intensive jobs to a single worker-pod and running them simultaneously may lead to resource starvation. To avoid starvation, if the resource requirements (cores, memory disk) of jobs are uncertain, the Work Queue framework will conservatively assign only one job to a worker at a time. (We will relax this assumption in the next section.) This setting makes the worker size critical to the performance of the individual job. Therefore, when setting up Work Queue on Kubernetes, the size of the worker-pod must be appropriately specified.

Assuming that the size of the resource pool is fixed, then a fine-grained configuration that has many small workers will be able to run more jobs concurrently, while a coarse-grained configuration with few, large workers will have a lower degree of parallelism. However, as the master's egress network bandwidth is fixed, the fine-grained configuration has to share limited bandwidth between more workers with more data movements. This imposes extra network overheads and might lead to longer workload execution time. Therefore, which configuration is better depends on whether the target workload is data-intensive or compute-intensive. However, this information is difficult to obtain without running the workload several times.

B. Number of worker-pods

Besides the worker size, another parameter that needs to be determined is the number of worker-pods. Resource demands of different workloads vary dramatically. Even for a single workload, resource usage can diverge significantly during the runtime. Therefore, the number of worker-pods needs to be changed frequently.

An existing option of adjusting the number of worker-pods is using the Horizontal Pod Autoscaler (HPA) of Kubernetes [22]. HPA adjusts the number of pods based on the ratio between a metric's desired value and its current value. For example, we can get the desired amount of CPU by equation (1), with $CurrentCPU$ and $CurrentCPUUse$ reported by Kubernetes and the $DesiredCPUUse$ set by users.

$$DesiredCPU = CurrentCPU \times \frac{CurrentCPUUse}{DesiredCPUUse} \quad (1)$$

However, the nature of HPA only allows it to make delayed responses to the varying resources load. Although this mechanism works well with latency-sensitive micro-services, it does not work with HTC workloads, resulting in three possible problems: i) the cluster could scale up too slowly and miss the peak resource demand. ii) resources could be over-provisioned when they are no longer needed. iii) workloads might never scale up to the desired degree.

We show these three results by running the BLAST bioinformatics workload [23] on a GKE[1] cluster that can be scaled up to 15 nodes with three different desired CPU usage, 10%, 50% and 99% (hereinafter referred as Config-10, Config-50, and Config-99). The BLAST workload we used comprises of 200 parallel jobs with each of them having the same size of input data. We assume that the resource requirements of individual jobs are known in advance, and consider four dimensions: i) the number of worker-pods connected, ii) the number of idle worker pods, iii) the desired number of worker-pods calculated by HPA, and iv) the number of worker-pods required in an ideal scenario. As shown in figure 2, Config-10, and Config-50 have a similar workload execution time (1294 versus 1304 seconds), close CPU usage (68.3% versus 65.2%), and the same maximum cluster size, i.e., 15 nodes. The primary difference is that Config-10 takes longer to scale up than Config-50. This is due to the larger disparity between current and target CPU load.

In contrast, Config-99 never scales up and results in four times longer workload execution time (4682 seconds) than the previous two configurations. In summary, even though Config-10 and Config-50 finally scale up to the desired degree, they are still far from optimal, which is to have the workload complete in 240 seconds. Therefore, the autoscaler reacting to system indicators does not always work with HTC workloads.

[1]Google Kubernetes Engine

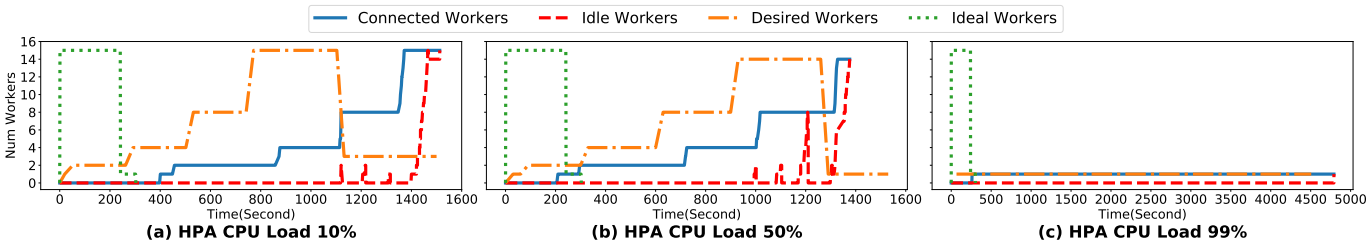


Fig. 2: Workload Runtime Statistics with Different HPA Target CPU Load

IV. Proposed Solution

A. Large Pod with Resource Monitoring

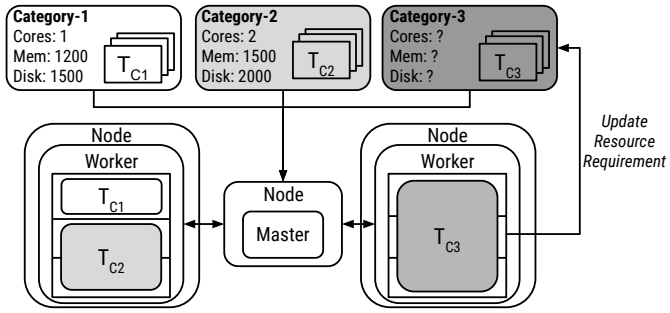


Fig. 3: One Worker-Pod per Node with Runtime Resource Monitoring

As discussed in section III-A, for an arbitrary workload, it is challenging to decide the size of the worker-pod. However, if each job's resource requirements are explicitly stated, we can run multiple jobs parallelly in a worker-pod. The configuration with a larger worker-pod can benefit from larger network bandwidth as well as a high degree of parallelism. We verify this by running the BLAST workload [23]) comprising of 100 parallel jobs, each job having a (cacheable) 1.4GB shareable input and 600KB output, on a GKE cluster with three configurations. Concretely, the GKE cluster consists of 5 physical nodes with each node having 3 vCPUs and 12 GB RAM. Configuration (a) has 15 worker-pods with each worker-pod occupying 1 vCPU, 4 GB RAM, configuration (b) has 5 worker-pods with each worker occupying an entire physical node, and configuration (c) has a similar worker-pods setting as (b) knowing resource requirements of all jobs.

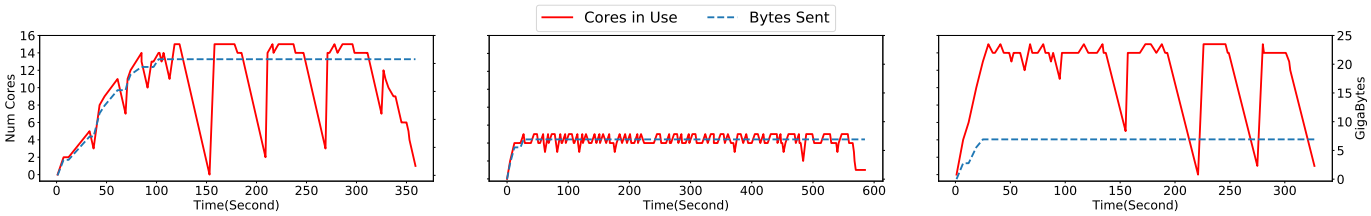
As shown in figure 4, the fine-grained configuration has 411 seconds of workload execution time, 278.382 MB/S average bandwidth, and 87.21% CPU usage. The coarse-grained configuration has 632 seconds of workload execution time, 452.138 MB/S average bandwidth, and 32.43% average CPU use. However, if the resource requirements of jobs are explicitly known, the coarse-grained configuration can complete the workload in 330 seconds with 466.173 MB/S average bandwidth and 85.73% average CPU usage, which outstrips the other two configurations in terms of both resource utilization and network bandwidth. Therefore, if the job resource requirement can be accurately estimated, the configuration with larger worker-pod should be preferred.

A primary characteristic of the HTC workload is that parallel jobs from the same stages are usually copies of the same program that works on different input datasets. Besides, the size and the pattern of the input dataset is usually the same between jobs as we usually divide a large dataset into equally sized small portions and assign them to each job. We can leverage this fact by splitting jobs into sub-categories according to their belonging stages. By referring to the resource consumption of completed jobs, we can obtain resource requirements for waiting jobs belonging to the same category. Concretely, we achieve this in three steps (figure 3), i) tag jobs of the workload by category before execution; ii) for the first job of a category, uses a worker-pod exclusively, has resource consumption measured, and applies the resource requirement to all jobs belonging to the same category; iii) after resource requirement of jobs are updated, run jobs parallelly in worker-pod that has enough resources.

B. Well-informed Autoscaling

The majority of HTC workloads are resource-hungry, which often results in steady and high system loads. Therefore, rather than using a reactive autoscaler that scales the resource pool based on system loads, a better-informed autoscaling approach should also take into account workload level (i.e., the status of the job scheduler) and job level (i.e., the resource requirement of each job) information.

In a queue-based submission model, jobs are queued up and wait for appropriate resource slots. An autoscaling mechanism that works with this model is to provision new resources to remedy the resource shortage when the length of the job queue increases while removing idle resources vice versa. As mentioned in the last section, we can estimate resource requirements of waiting and running jobs based on completed jobs, for any given time point, we can calculate the resource shortage by considering resource requirements and quantity of waiting and running jobs. However, a critical problem of this approach is that the length of the job queue might keep changing while the cluster manager is initializing new resources. This can result in resource over-provisioning or under-provisioning. In order to tackle this problem, we propose an autoscaling mechanism that resizes the resource pool by considering the gap between resource supply and demand during the initialization of the new resources.



(a) 15 workers with 1 core per worker and uknown resource requirement

(b) 5 workers with 3 cores per worker and uknown resource requirement

(c) 5 workers with 3 cores per worker and known resource requirement

Fig. 4: Runtime Statistics of Workload with Unknown Resource Requirements

To illustrate the basic idea of this mechanism, we define five terminologies, i) Resource In-use (RIU), the amount of resources currently being used by running jobs; ii) Resource Shortage (RSH), the amount of resources desired by waiting jobs; iii) Resource Demand (RD), the sum of resources in-use and resource shortage; iv) Resource Supply (RS), the amount of available resources supplied by the cluster manager; v) Resource Waste (RW), the amount of idle resources. During runtime, resource demand is uncontrollable, and there usually exists a maximum resource quota depending on the user budget. Despite the above factors, an efficient autoscaling mechanism should **maximize resources in-use and workload throughput; meanwhile, minimize resource waste, resource shortage, and workload execution time**.

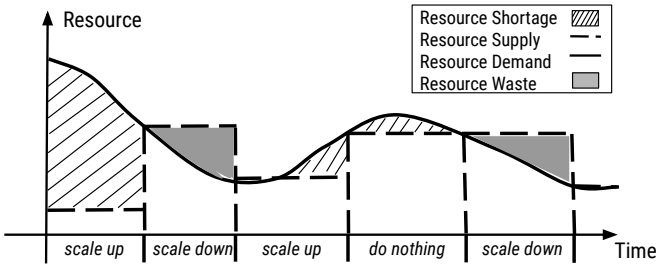


Fig. 5: An Example of workloads resource relationship on cluster

Formally, we define a time interval between the time point of submitting new resources request (t_{nr}) and the time point when all new resources are ready (t_{rr}) as a resource initialization cycle. Then we establish the objective function (2), whose objective is to minimize the resource shortage at the time point t_{rr}. Specifically, $\Delta RSH(t)$ is the resource amount of newly enqueued waiting jobs at t, and $\Delta RIU(t)$ is the resource amount of finished jobs at t. When evaluating the resource efficiency, both the resource shortage and the workload execution time can act as indicators, and we chose resource shortage here to better demonstrate the resource relationship on the cluster.

Then resource shortage of the system can be calculated by equation (2), specifically, $\Delta RSH(t)$ is the resource amount of newly enqueued waiting for jobs at t, and $\Delta RIU(t)$ is the resource amount of finished jobs at t

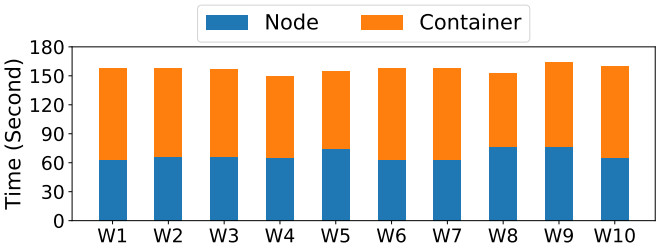


Fig. 6: GKE Resource Initialization Latency

$$RSH(t_{rr}) = RSH(t_{nr}) + \sum_{t=t_{nr}}^{t_{rr}} (\Delta RSH(t) - \Delta RIU(t)) \quad (2)$$

A potential problem of this mechanism is, if the variation rate of the resource pool – when and how many resources the cluster manager will add – is unknown, it will be challenging to estimate $\Delta RSH(t)$ and $\Delta RIU(t)$. However, we notice that cluster managers usually process reservation requests in batches; thus, requests submitted in the same batch that ask for the same machine types and container images in the same geographical region should experience similar resource initialization latency.

To verify this speculation, we measure the resource initialization time (including machine reservation and container pulling time, see figure 6) by creating pods that have resource requirements which cannot be met by existing nodes. We ran the benchmark 10 times on GKE and found that the resource initialization latency alters little (mean: 157.4 seconds, standard deviation: 4.2 seconds). **Therefore, we can assume that the resource pool's size is constant during a resource initialization cycle**. Furthermore, we divide the workload's lifetime into consecutive time intervals, with each interval equal to a resource initialization cycle, then the relationship between the resource supply and demand should look like figure 5. To resize the resource pool correctly, we only need to calculate the RSH at the end time point of each cycle and then create/delete worker-pods accordingly. As the average job execution time, job resource requirement, and the total amount of available resources (i.e., the size of the resource pool) is known, we can easily calculate the $\Delta RSH(t)$ and $\Delta RIU(t)$ for any given time point.

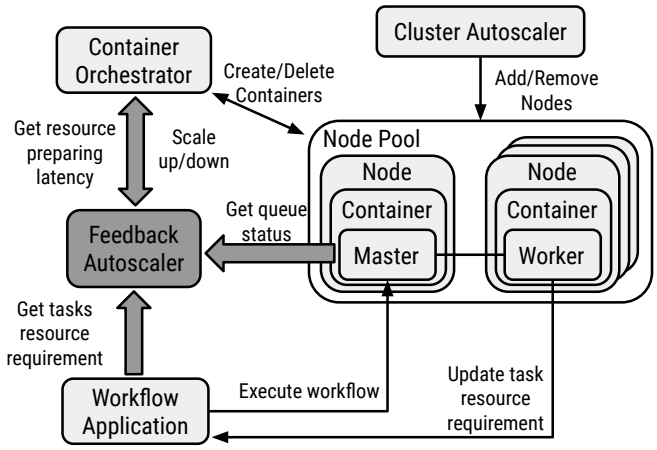


Fig. 7: A Well-Informed Autoscaling Approach

Based on this mechanism, we design a well-informed feedback autoscaler (figure 7) that autoscales the resource pool for HTC workloads by **considering two inputs, the length of the job queue as well as the resource initialization time of the cluster manager, and a feedback input, runtime statics (including the job execution time and the resource consumption) of completed jobs**.

Specifically, the autoscaler resizes the resource pool in three steps, i) measuring the latest resource initialization time of the cluster manager, obtaining the runtime statics (including resource consumption as well as the execution time) of completed jobs reported by the workload manager, and getting the length of the job queue reported by the job scheduling framework; ii) estimating the resource shortage (RSH) at the end of next resource initialization cycle by applying equation (2) and scale the resource pool accordingly. (i.e., scale down if $RSH < 0$, scale up if $RSH > 0$, and do nothing if $RSH = 0$) iii) scaling the resource pool by adjusting the number of worker-pods (changing the number of worker-pods could result in, pending pods with no available node or idle nodes that are underutilized, and the cloud controller manager [24] will add/remove nodes accordingly).

V. Implementation

A. System Components

We implement above autoscaler in a new middleware and named it High Throughput Autoscaler (HTA). HTA deploys Work Queue job scheduling framework on Kubernetes, helps Makeflow to submit jobs and manages the container cluster through the Kubernetes during runtime. Figure 8 shows the system architecture of HTA, which includes two main components,

1) Makeflow Kubernetes Operator, which contains four sub-components, i) *Informer Cache*, which receives a notice when registered objects (i.e., Pod, Service, and Statefulset) are created, updated, or deleted. By monitoring the worker-pod lifecycle, it can always get the latest resource initialization time, i.e., how long does it take for a worker-pod with no available node to be ready; ii) *Resource Provisioner*, which calculates the resource shortage and resizes clusters based on the status of the job queue, statistics of completed jobs, and resource initialization time; iii) a *TCP Client*, which sets up a network connection to the Work Queue master, submitting ready jobs, receiving completed jobs, and getting the job queue status for the resource provisioner; iv) and a *TCP Server*, which gathers jobs specifications from Makeflow, including input/output information and resource requirements. HTC workloads often produce a large number of data transmission during runtime. To decrease network overheads, we keep the connections between Makeflow, HTA, and Work Queue alive during the entire lifecycle of workloads.

2) Container Cluster On GKE, HTA sets up a container cluster to run the Work Queue framework. As discussed early IV-A, we apply the configuration of large worker-pods with each pod occupying an entire physical node. Initially, the cluster has 3 nodes [2] and will scale up on-demand later. To avoid loss of intermediate data and ensure a restarted master pod can run on the same physical node with the same identity, we encapsulate the master pod inside a StatefulSet and dump intermediate data into a persistent volume. Since users often start workflow applications locally or from a network namespace different from container clusters, we set up dedicated services for HTA and worker-pods to access the master pod from outside and inside of the cluster.

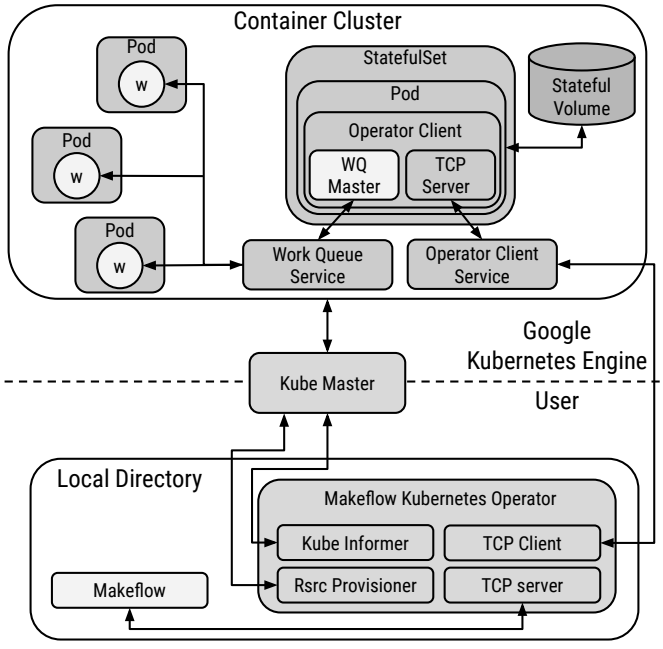


Fig. 8: Makeflow Kubernetes Operator Architecture

[2] A GKE cluster with a size smaller than 3 nodes that might be unreachable during the Kubernetes master node upgrade. To avoid unnecessary disruption, we use 3 nodes by default.

B. Resource Initialization Time

To calculate the resource shortage at the end time point of a resource initialization cycle, we need to gather three pieces of information: i) resource requirement of waiting jobs, which can be estimated by referring to complete jobs belonging to the same categories; ii) number of waiting and running jobs, which can be obtained by inquiring Work Queue; iii) resource initialization time of the cluster manager. To obtain the latest resource initialization time, we use the log data of the informer API to track the lifecycle of each worker-pod, which includes four states (see figure 9):

1) No Available Node This state happens when Kubernetes receives requests for creating a new worker-pod, but there exists no physical node that has enough resources. The worker-pod will stay in `Pending` state with event `Insufficient Resource`. Then the cloud controller manager will detect the pending pods and reserve new physical nodes.

2) No Container Image The `Pending` worker-pod has been scheduled on the new node, but the container image has not been pulled yet. The worker-pod will stay at the `Pending` state with a new event `Pulling Image` while waiting for the Kubelet to pull the image.

3) Worker-Pod Running After Kubelet pulls the container image, the worker-pod starts running.

4) Worker-Pod Stopped When deciding to scale down the resource pool, the HTA will stop the worker process running inside the worker-pod. After the worker process completes all running job, the worker-pod will turn into `Succeeded` phase and will be removed.

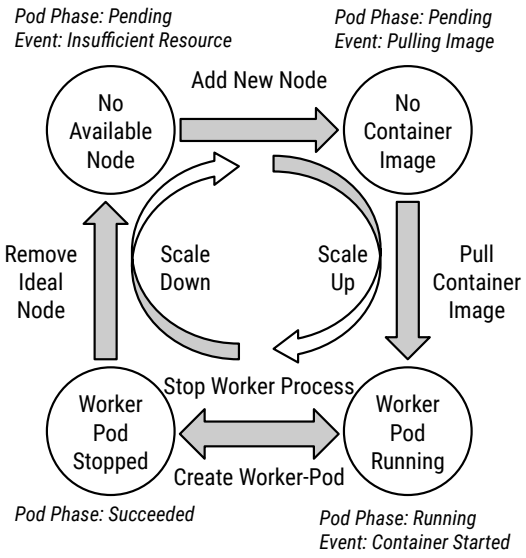


Fig. 9: Lifecycle of a Worker-Pod

We leverage client-go's informer cache to track the lifecycle of each worker-pod and its event. If the creation process of a worker-pod experience three states – No Available Node, No Container Image, Worker-Pod Running – we will use the time interval between HTA generating worker-pods creation request and the worker-pod becoming ready as the latest resource initialization time.

C. Resource Autoscaling

The autoscaling process includes three stages:

1) Warm-Up Stage In this stage, HTA sets up the Work Queue framework on Kubernetes with 3 nodes and start tracking the resource initialization time. Makeflow submits the first batch of jobs to HTA. Instead of fanning out all jobs at once, HTA sends out only a portion of jobs with one job per category to collect resource statistics of each category.

2) Runtime Stage During runtime, HTA periodically estimates resource needs and resizes the resource pool on-demand. As shown in algorithm 1, when calculating the resource shortage, the latest resource initialization time, information about running/waiting jobs, jobs runtime information (i.e. resource requirement and execution time) grouped by categories and information of active workers are passed to the function. The estimation function checks the current resource balance (line 3 - 18), evaluate the relationship between resource supply and demand (line 18 - 24), and returns the desired scale variation (line 25).

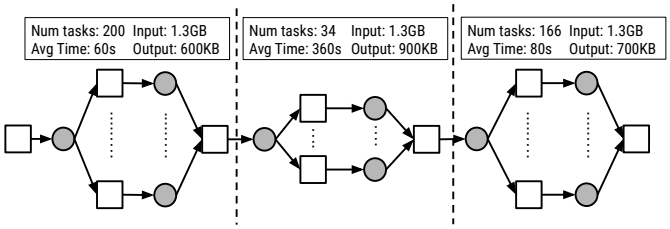
If the scale variation is larger than zero, a scaling up action will be applied which creates new worker-pods, while scaling down action will be taken if the scale variation is smaller than zero, which drains worker pods, i.e., stop the worker once all running jobs on it are finished. Furthermore, to avoid system thrashing caused by frequently resizing, time intervals between two resizing actions is always set as the latest resource initialization time.

3) Clean-Up Stage, when there are no more jobs need to run, HTA will receive a notification from Makeflow and drain all workers. Once all jobs are complete, HTA will erase intermediate data, delete all deployment units left on Kubernetes, and send out a notification to the user.

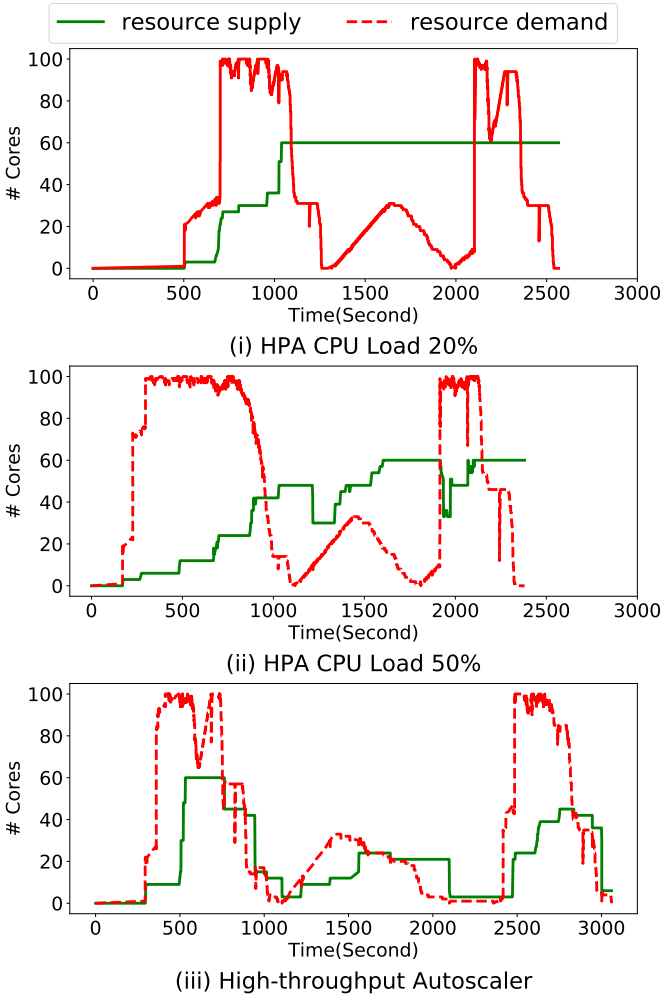
VI. Evaluation

In section (section III), we show that with CPU load higher than 50%, HPA would rarely scale up the cluster. Therefore, we compare HTA to two setups: (i) HPA-20%, which uses HPA of Kubernetes with target CPU load 20%; and (ii) HPA-50%, which use HPA of Kubernetes with target CPU load 50%.

We run our experiments on a Google Kubernetes Engine (GKE) with Kubernetes version 1.13 using 20 n1-standard-4 instances. Each instance has 4 vCPUs, 15 GB RAM, and 100 GB SSD with Container-Optimized OS from Google. To avoid network speed variations between a public Docker registry and the daemons, we set up a private container registry on Google cloud. As discussed in section (section IV-A), we set up the Work Queue framework with one worker per pod. To monitor the resource consumption of tasks, we enable the resource monitor [25] of Work Queue.



(a) Blast Workflow Structure



(b) Resource Supply and Demand of Blast

Resource Autoscaler	Workflow Runtime (*second*)	Accumulate Waste (*core* × *s*)	Accumulate Shortage (*core* × *s*)
HPA(20% CPU)	2656	51324	34813
HPA(50% CPU)	2480	39353	66611
HTA	3060	9146	40680

(c) Blast Workflow Performance Summary

Fig. 10: Blast Workflow

Algorithm 1: Resource Estimation Algorithm

Data: $rsrcInitTime$, $runningTasks$, $waitingTasks$, $taskCategoryInfo$, $activeWorkers$
Result: $scaleChanged$, $timeToNextAction$
/* simulate the execution of workflow */
$rsrcCap$ = TotalRsrc(activeWorkers)
$avaRsrc$ = AvaRsrc(activeWorkers, runningTasks)
for $t = 1$; $t < rsrcInitTime$; $t++$ **do**
  $completeTasks = TasksCompleteAt(t, runningTasks)$
  /* return resource used by complete tasks */
  foreach *task in* $completeTasks$ **do**
    $avaRsrc = avaRsrc + task.rsrc$
  end
  /* simulate task dispatching */
  foreach *task in* $waitingTasks$ **do**
    /* no resource available, moving on */
    if $AvaRsrc == 0$ **then**
      break
    end
    /* dispatch waiting tasks */
    if $task.rsrc < avaRsrc$ **then**
      $avaRsrc = avaRsrc - task.rsrc$
      $runningTasks = append(runningTasks, task)$
      $delete(waitingTasks, task)$
    end
  end
end
/* resources are enough, do nothing */
if *Len(waitingTasks) == 0* **then**
  return *0, DefaultCycle*
end
/* scale down if there is spare resources */
if $avaRsrc > 0$ **then**
  return *-NumIdleWorkers(avaRsrc), MaxRuntime(runningTasks)*
end
/* scale up, otherwise */
return *WorkerRequired(waitingTasks), rsrcInitTime*

A. Multistage Workload

We start by considering a multistage BLAST workload, which contains three stages with each stage involves three steps, i.e., splitting an input data, aligning subsequences, and reducing intermediate results. We consider five dimensions: i) workload execution time; ii) resource shortage; iii) resource supply; iv) accumulated resource shortage; and v) accumulated resource waste. The accumulated resource shortage/waste is the definite integral of resource shortage/waste over the workload runtime.

As shown in figure 10a, the first and last stages of the workload contains tasks more than the second stage (200,

164 compared to 34), an optimal autoscaler should resize the cluster follow the same pattern, i.e., a decrease in resource demand in the middle of the lifecycle, and a bump up once the workload entering into the third stage.

However, as shown in figure 10b, if HPA is applied, cluster size will gradually increase and stay at the capacity limit (i.e., 20 nodes, 60 cores) until workload complete. This is because to avoid pods from thrashing, there is a stabilization interval between two downscale operations, and the default value is 5 minutes. Even though we can increase the frequency of downscale by tuning this value, different workloads have various resource changing rates, without running the same workload multiple times, it is challenging to pick the right value.

In contrast, HTA autoscale the cluster as expect. To take As shown in table 10c, even though we see a slight increase in workload execution time (12.5% compare to HPA-20%, 16.6% compare to HPA-50%), HTA reduces the resource waste dramatically ($5.6\times$ compare to HPA-20%, $4.30\times$ compare to HPA-50%).

In general, when resizing resource pool for workload with fluctuant resource demands, HTA can make a more accurate autoscaling plan compare to HPA as HTA considering information from every component of the software stack.

B. I/O Intensive Workload

While CPU load is often a good indicator of system load, applications' performance might be bound by other resources. Choosing a wrong indicator might cause HPA scaling cluster to an inappropriate degree. To reveal how will the autoscaler behave for workload bounded by resources other than CPU, we create a synthetic workload that contains 200 I/O intensive parallel tasks. Each task of them runs `dd` commands to read/write data from the disk device. We consider the same dimensions as the previous benchmark VI-A.

As shown in sub-figures (i) and (ii) of figure 11b, while tasks are queuing up on Work Queue, the cluster size maintain in 1. The reason is that each task is busy at reading/writing data, and the CPU load is rarely over 20%. In contrast, HTA can scale up the cluster to the desired size as it considering CPU load as well as usages of other resources (e.g., max number of processor required by task) when establishing an autoscaling plan. As a result, by using HTA, we successfully scale up the cluster and shorten workload execution time by around $3.66\times$.

In terms of resource waste, even though configuration using HPA does not have resource waste, the significant resource shortage and small cluster scale result in unacceptable throughput and execution time. In contrast, when running with HTA, even though there is a small amount of resource waste at the beginning as Work Queue master assigning tasks to workers, once the cluster upscaled to the desired degree, we see no resource waste during the entire lifecycle of workload.

In general, using HPA require users to know the workload well and pick the correct resource indicator. Moreover, in order to scale the cluster to the desired degree, users need to fine-tune multiple system options. However, it is challenging for regular users to choose appropriate parameters without running workloads multiple times. By contrast, HTA estimates the resource shortage based on the real-time status of different system components, and dynamically adjust the stabilization cycle by considering the latest resource initialization time. Therefore, by using HTA, we can resize cluster on-demand without user intervention.

VII. Related Work

A. Autoscaling on the cloud

Autoscaling on the cloud is not a new topic, regardless of the underlying virtualization technology, researchers in previous studies have proposed various efficient autoscaling mechanisms that can be divided into three categories.

Rule-based autoscaling mechanisms [26], [27] usually require users to specify a set of fixed thresholds (e.g., CPU, I/O, bandwidth), and resize the cluster once these thresholds are reached. These mechanisms are generic, work to different workloads, but they only consider infrastructure-level metrics and, hence, do not work with HTC workloads that are not resource-bound. HTA takes into account infrastructure-level (resource initialization time), framework-level (job queue length) as well as application-level metrics (resource requirement and execution time of jobs) to resize the cluster more accurately.

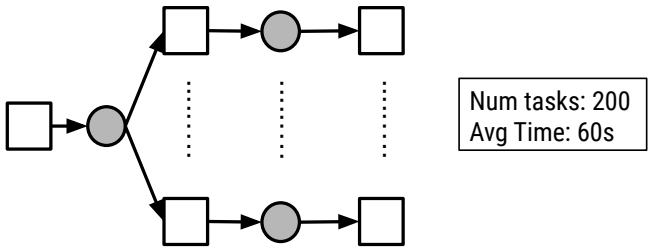
Learning-model based approaches apply linear regression [28], reinforcement learning [29]–[31] or other machine learning models [32] to predict future resource demands and resize the cluster in advance. However, these approaches usually require a long time to train the models before they can accurately predict the resource demands, which might result in the poor quality of service (QoS) during the early stage of the learning period. In contrast, by leveraging the fact that HTC workloads usually comprise of many small parallel jobs with similar resource requirements, HTA can accurately estimate the resource requirement of workloads at the early stage.

Control-theory based mechanisms [33]–[35] use adaptive feedback controllers to scale the resource pool by monitoring not only the system load but also taking application-specific metrics (e.g., requests arrival rate) into account. Comparing to them, HTA considers the resource initialization time of the cluster manager and estimates the resource demands on the job level, which allows HTA to predict future resource demands more accurately and perform proper autoscaling actions more timely.

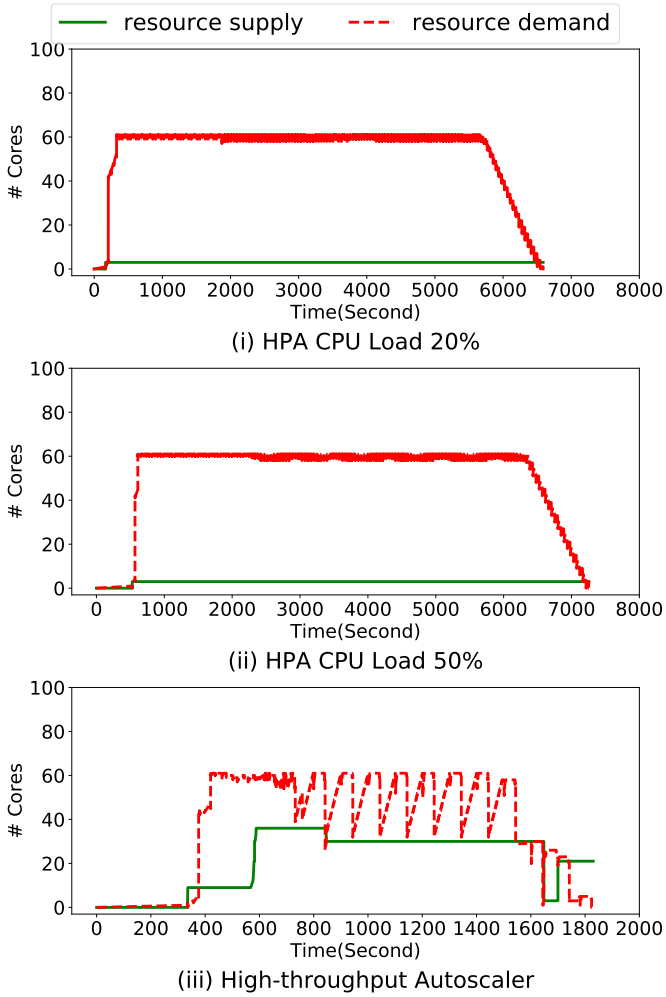
B. Autoscaling batch workloads with Kubernetes

With the rise of Kubernetes as the new standard of container orchestration, there emerged many systems that attempt to autoscale batch workloads on Kubernetes.

KFServing [36] is designed for serving machine learning (ML) frameworks, like Tensorflow [37] and PyTorch [38], on Knative platform [39]. It leverages Knative's request-based autoscaling mechanism [40], which autoscales the cluster based on how many concurrent requests can be handled by a container. This mechanism works well with ML workloads



(a) I/O Bound Workflow Structure



(b) Resource Supply and Demand of I/O Bound Workflow

Resource Autoscaler	Workflow Runtime (*second*)	Accumulate Waste (*core* × *s*)	Accumulate Shortage (*core* × *s*)
HPA(20% CPU)	6670	159	337737
HPA(50% CPU)	7230	82	357640
HTA	1823	2028	31840

(c) I/O Bound Workflow Performance Summary

Fig. 11: I/O Bound Workflow

that are not bound by common computing resources. However, HTC workloads have unique characteristics different from ML and other batch workloads, by leveraging these characteristics, consisting of parallel jobs with a similar resource requirement and execution time, HTA can make scaling plans that best fit the HTC workload.

Escalator [41] is an optimizer for the HPA of Kubernetes. It is designed for large batch workloads that cannot be interrupted and migrated when the cluster needs to shrink. As an optimizer of HPA, Escalator still relies on system indicators to resize clusters, while HTA makes scaling plans based on system metrics collect from Kubernetes as well as real-time status of job scheduling framework running on Kubernetes, which can estimate the scale variation more accurately and timely.

C. Running HTC workloads with containers

Other than setting up the job scheduling framework on container orchestrators, There exist various configurations of running HTC workloads with container technologies. One option is integrating container runtime into the workload system. In our previous work, we successfully integrated Docker and Singularity into Makeflow and Work Queue [42]. Other workload systems like IBM Spectrum LSF [43] and Altair's PBS Professional [44] also includes support for Docker containers. Though these deployments exploit the existing workload system that has been developed over the years to support high-throughput workloads, it is complicated to deploy such a system on a commercial cloud, which often renders more and newer hardware with evolved and compelling infrastructure. Our solution builds elastic container clusters that follow the pay-as-you-go approach with dedicated autoscaler, as well as accommodates properties of both high-throughput workload container orchestrator.

VIII. Conclusion

In this paper, we explored how to autoscale HTC workloads on the cloud through Kubernetes. We show that the default autoscaler of Kubernetes – which use system indicator to resize resource pool – does not work to the HTC workload, and propose a new autoscaling mechanism that leverages the unique characteristics of the HTC workload to automatically resizes the resource pool. Based on this mechanism, we developed a High-throughput Autoscaler (HTA), which manages the workload scheduling framework and resizes the container cluster based on the resource utilization of complete jobs, the real-time status of the job queue, and the resource initialization time of the cluster manager. Our evaluation shows that HTA can improve the resource usage of CPU-bound workloads by 5.6×, and shorten the execution time of IO-bound workloads by up to 3.66× compare to the default autoscaler of the Kubernetes.

References

[1] "Global infrastructure." Amazon Web Services, 2019, https://aws.amazon.com/about-aws/global-infrastructure/.

[2] "About google data centers." Google, 2019, https://www.google.com/about/datacenters/.

[3] "High performance computing - virtually unlimited infrastructure and fast networking for scalable hpc." aws, 2019, https://aws.amazon.com/hpc/.

[4] "High performance computing." Google Cloud, 2020, https://cloud.google.com/solutions/hpc.

[5] "High performance computing (hpc) on azure." Azure, 2020, https://docs.microsoft.com/en-us/azure/architecture/topics/high-performance-computing.

[6] "Production-grade container orchestration," 2017, https://kubernetes.io/.

[7] "Horizontal pod autoscaler," 2019, https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler.

[8] "Scaling based on cpu or load balancing serving capacity," 2020, https://cloud.google.com/compute/docs/autoscaler/scaling-cpu-load-balancing.

[9] "Target tracking scaling policies for amazon ec2 auto scaling," 2020, https://docs.aws.amazon.com/autoscaling/ec2/userguide/as-scaling-target-tracking.html.

[10] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock, "Kepler: an extensible system for design and execution of scientific workflows," in *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on*. IEEE, 2004, pp. 423–424.

[11] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. F. da Silva, M. Livny *et al.*, "Pegasus, a workflow management system for science automation," *Future Generation Computer Systems*, 2014.

[12] E. Afgan, D. Baker, B. Batut, M. Van Den Beek, D. Bouvier, M. Čech, J. Chilton, D. Clements, N. Coraor, B. A. Grüning *et al.*, "The galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2018 update," *Nucleic acids research*, vol. 46, no. W1, pp. W537–W544, 2018.

[13] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[14] P. Bui, D. Rajan, B. Abdul-Wahid, J. Izaguirre, and D. Thain, "Work Queue + Python: A Framework For Scalable Scientific Ensemble Applications," in *Workshop on Python for High Performance and Scientific Computing (PyHPC) at the ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (Supercomputing)* , 2011.

[15] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: distributed, low latency scheduling," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 69–84.

[16] "Spring batch," 2019, https://spring.io/projects/spring-batch.

[17] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 5.

[18] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in *NSDI*, vol. 11, 2011, pp. 22–22.

[19] N. Hazekamp, N. Kremer-Herman, B. Tovar, H. Meng, O. Choudhury, S. Emrich, and D. Thain, "Combining static and dynamic storage management for data intensive scientific workflows," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 2, pp. 338–350, 2017.

[20] N. Kremer-Herman, B. Tovar, and D. Thain, "A lightweight model for right-sizing master-worker applications," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 504–516.

[21] M. Albrecht, P. Donnelly, P. Bui, and D. Thain, "Makeflow: A Portable Abstraction for Data Intensive Computing on Clusters, Clouds, and Grids," in *Workshop on Scalable Workflow Enactment Engines and Technologies (SWEET) at ACM SIGMOD*, 2012.

[22] "Kubernetes cluster autoscaler," 2019, https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

[23] "Basic local alignment search tool," 2019, https://blast.ncbi.nlm.nih.gov/Blast.cgi.

[24] "Cloud controller manager," 2020, https://kubernetes.io/docs/concepts/architecture/cloud-controller/.

[25] B. Tovar, R. F. da Silva, G. Juve, E. Deelman, W. Allcock, D. Thain, and M. Livny, "A job sizing strategy for high-throughput scientific workflows," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 2, pp. 240–253, 2017.

[26] Z. A. Al-Sharif, Y. Jararweh, A. Al-Dahoud, and L. M. Alawneh, "Accrs: autonomic based cloud computing resource scaling," *Cluster Computing*, vol. 20, no. 3, pp. 2479–2488, 2017.

[27] P. Dube, A. Gandhi, A. Karve, A. Kochut, and L. Zhang, "Scaling a cloud infrastructure," Mar. 29 2016, uS Patent 9,300,552.

[28] S. Islam, J. Keung, K. Lee, and A. Liu, "Empirical prediction models for adaptive resource provisioning in the cloud," *Future Generation Computer Systems*, vol. 28, no. 1, pp. 155–162, 2012.

[29] P. Jamshidi, A. M. Sharifloo, C. Pahl, A. Metzger, and G. Estrada, "Self-learning cloud controllers: Fuzzy q-learning for knowledge evolution," in *2015 International Conference on Cloud and Autonomic Computing*. IEEE, 2015, pp. 208–211.

[30] H. Arabnejad, C. Pahl, P. Jamshidi, and G. Estrada, "A comparison of reinforcement learning techniques for fuzzy cloud auto-scaling," in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 2017, pp. 64–73.

[31] D. Tsoumakos, I. Konstantinou, C. Boumpouka, S. Sioutas, and N. Koziris, "Automated, elastic resource provisioning for nosql clusters using tiramola," in *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. IEEE, 2013, pp. 34–41.

[32] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang, "Adaptive, model-driven autoscaling for cloud applications," in *11th International Conference on Autonomic Computing ({ICAC} 14)*, 2014, pp. 57–64.

[33] L. Baresi, S. Guinea, A. Leva, and G. Quattrocchi, "A discrete-time feedback controller for containerized cloud applications," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 217–228.

[34] A. Ali-Eldin, M. Kihl, J. Tordsson, and E. Elmroth, "Efficient provisioning of bursty scientific workloads on the cloud using adaptive elasticity control," in *Proceedings of the 3rd workshop on Scientific Cloud Computing*, 2012, pp. 31–40.

[35] A. Ali-Eldin, J. Tordsson, and E. Elmroth, "An adaptive hybrid elasticity controller for cloud infrastructures," in *2012 IEEE Network Operations and Management Symposium*. IEEE, 2012, pp. 204–212.

[36] "Kfserving," 2020, https://github.com/kubeflow/kfserving.

[37] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265–283.

[38] A. Paszke, S. Gross, S. Chintala, and G. Chanan, "Pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration," *PyTorch: Tensors and dynamic neural networks in Python with strong GPU acceleration*, vol. 6, 2017.

[39] "Kubernetes-based platform to deploy and manage modern serverless workloads," 2020, https://knative.dev/.

[40] "Configuring autoscaling for knative," 2020, https://knative.dev/docs/serving/configuring-autoscaling/.

[41] "Atlassian escalator," 2019, https://github.com/atlassian/escalator.

[42] C. C. Zheng and D. Thain, "Integrating Containers into Workflows: A Case Study Using Makeflow, Work Queue, and Docker," in *Workshop on Virtualization Technologies in Distributed Computing (VTDC)*, 2015.

[43] "Ibm spectrum lsf v10.1 documentation." IBM Knowledge Center, 2017, https://www.ibm.com/support/knowledgecenter/en/SSWRJV_10.1.0/lsf_welcome.

[44] "Pbs professional." Altair Engineering, Inc., 2017, http://www.pbsworks.com.