Addressing the Challenges of Executing a Massive Computational Cluster in the Cloud

Brandon Posey
Clemson University
bposey@clemson.edu

Christopher Gropp
Clemson University
cgropp@clemson.edu

Boyd Wilson
Omnibond
boydw@omnibond.com

Boyd McGeachie
Amazon Web Services
boydm@amazon.com

Sanjay Padhi
Amazon Web Services
sanpadhi@amazon.com

Alexander Herzog
Clemson University
aherzog@clemson.edu

Amy Apon
Clemson University
aapon@clemson.edu

***Abstract*—A major limitation for time-to-science can be the lack of available computing resources. Depending on the capacity of resources, executing an application suite with hundreds of thousands of jobs can take weeks when resources are in high demand. We describe how we dynamically provision a large scale high performance computing cluster of more than one million cores utilizing Amazon Web Services (AWS). We discuss the trade-offs, challenges, and solutions associated with creating such a large scale cluster with commercial cloud resources. We utilize our large scale cluster to study a parameter sweep workflow composed of message-passing parallel topic modeling jobs on multiple datasets. At peak, we achieve a simultaneous core count of 1,119,196 vCPUs across nearly 50,000 instances, and are able to execute almost half a million jobs within two hours utilizing AWS Spot Instances in a single AWS region. Our solutions to the challenges and trade-offs have broad application to the lifecycle management of similar clusters on other commercial clouds.**

***Index Terms*—high performance computing cluster, parallel scientific applications, automatic resource provisioning and de-provisioning, massive scale workflow, cloud computing**

I. Introduction and Motivation

There are many different high performance computing (HPC) environments available to researchers today, including campus-scale clusters at academic institutions and supercomputing resources at national centers. However, the size of most academic resources and contention for access to the resources even at national supercomputing centers limit the availability of the systems to individual researchers. This limitation can slow the time-to-science for researchers who have an infrequent or urgent need to access large scale resources. For example, the computation required to predict or manage a natural disaster such as a hurricane may be much larger than the normal workload that is processed by the available institutional resources.

While HPC workloads are generally tightly coupled and depend on low latency networks, a high throughput computational (HTC) workload generally consists of independent programs that rely less on low-latency messaging. These HTC workflows can contain a large number of individual jobs that run independently of other jobs in the suite. Our particular HTC workload executes a suite of jobs, each with modest parallelism that executes on a single whole computer, that collectively evaluate a large range of parameters to a given modeling application. Acquiring enough resources to run a massive HTC workflow in a fixed turn-around time on traditional HPC resources can be difficult. One limitation may be the physical size or availability of computing, networking, and computing resources. Other limitations include administrative limits such as how many jobs can be submitted to a scheduler at one time by a single user, or how many jobs a single user is permitted to run during a given time period. When too few resources are available, the number of HTC jobs that can run simultaneously is decreased, leading to a longer completion time for the whole suite and delay in producing the desired computational results. When an HTC suite is massive, say, consisting of tens of thousands of jobs, a researcher could wait weeks for enough jobs to complete to get a usable result. Decreasing this processing delay for massive sized workloads is one of the main motivations for this research. This wait time limits the number of science questions that can be studied, slows the pace of scientific discovery, and can even be the critical difference between a result that arrives in time to help manage a disaster versus one that comes too late to help.

The increasing capability and elasticity of the commercial cloud provides a solution to this problem. Computing resources through commercial cloud providers such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP) are growing and maturing rapidly. According to Forrester, in 2018 the global public cloud market will grow at a 22% compound annual growth rate [1]. The allure of "unlimited" resources, the ability to have complete control over administrative policies of the environment, and the increasing variety of compute hardware offered by the commercial cloud can help to solve many of the issues currently faced when attempting to run HTC workflows of parallel applications. However, the challenges with dynamically provisioning traditional HPC-type environments at a massive scale on a commercial cloud have not been well studied.

In this study, we aimed to automatically provision and de-provision an environment large enough to push the limits of computation in the commercial cloud and to provide insight into the challenges and solutions when constructing such an environment. At the time of the study, the 1.1 million core HPC cluster that we provisioned in the cloud had more cores than all but three of the Top 500 largest supercomputers in the world, and was about forty-seven times larger than the number of cores available on the Clemson University campus cluster [2]. The HTC workflow that ran in less than an afternoon would have taken more than five days of dedicated access to the campus cluster. However, since the campus cluster is shared by Clemson users and has a typical utilization of about 90%, dedicated access to the system is not feasible. In the best case, our HTC workload would have taken weeks to execute. We pushed the limits of the commercial cloud to obtain useful scientific results much faster than is otherwise possible. We present in this paper the challenges, scalability, and job processing efficiency we can achieve with a fully cloud-based cluster environment.

The remainder of this paper is organized as follows. Section II discusses other research that has been performed in this area. Section III contains the technology background, including a discussion of the alternatives to the tools utilized in this experiment. Section IV discusses the limitations to scaling. Section V discusses the workload characteristics of the scientific application. Section VI discusses the execution and evaluation of the HTC workflow that we executed. Section VII describes conclusions and future work.

II. Related Research

Prior research related to this project has focused largely on the performance of specific applications in the commercial cloud or on the usability of the commercial cloud for scientific applications. Performance studies of HPC applications in the cloud typically compare different benchmarks or test applications to determine how best they can execute in the cloud [3]–[7]. Results have shown that the commercial cloud networking and virtualization overheads limit the effectiveness of the cloud for many HPC applications. These results led to a number of studies that sought to optimize various HPC applications in a cloud execution environment, such as [8], [9]. Other research has studied which types of applications and programming models are best suited for the cloud [10], [11]. The scale of these applications has been modest by our standards, ranging from a few nodes to a few thousand whole computing instances (i.e., nodes) in total. At peak, we achieved a simultaneous core count of 1,119,196 vCPUs across 49,925 instances, and executed almost half a million jobs within two hours utilizing AWS Spot Instances in a single AWS region.

Another area of research has centered around the usability of the cloud for scientific applications. Studies such as [12], [13] discuss the need for a simple usable cloud solution for running applications in the cloud. A study in the area of high energy physics [14] describes a hybrid deployment/cloud bursting model that enables high-energy physics users to utilize cloud resources in combination with local resources.

The largest scale HTC or HPC job that has been executed on a commercial cloud and publicized at the time of this article was an HTC job that was designed to execute a single workflow on the Google Cloud Platform (GCP) [15]. At its peak size it utilized 580,000 cores simultaneously. This suite of jobs executed using a work queue model in which the compute resources receive work from a queue master, similar to [16]. All the instances utilized operated as separate entities, the tasks operated on one instance, and instances were spread over multiple GCP regions. The submission tools utilized in that experiment were different from typical scientific computing environments. In comparison, our experiment uses a traditional scientific computing environment containing a traditional HPC batch scheduler. It supports the execution of a wide variety of parallel applications, including Message Passing Interface (MPI) applications across multiple nodes. The instances resided in a single AWS region. The environment uses SLURM [17], a widely used HPC batch scheduler, which allows the execution of any workflow designed to work with SLURM, with only minor modifications.

III. Technology Alternatives and Design Decisions

In this section we examine various technology alternatives and design decisions that are required when considering how to provision a massive cluster in the commercial cloud. Decisions include choosing a particular commercial cloud provider, determining the cost model for the experiment, and determining how to manage the workflow at such a large scale. Also, the characteristics of the workload constrain the viable design choices for the cluster environment.

Our experimental workload consists of many small, parallel jobs. Unlike large jobs, small-scale parallel jobs are less reliant on networking capabilities and have different reliability considerations than larger parallel jobs. Small parallel jobs can more easily be located on one or two large computing nodes and can also be more easily replicated, or even restarted, in the case of node failures. Larger scale parallel jobs are not considered in this evaluation. An MPI application will impose constraints on the number of cores that are required. Our MPI-based application can run efficiently on most numbers of cores, and we do not consider jobs that require GPUs. Also, our HTC workload has features that enable efficient resource use. For example, sets of jobs are able to reuse input files or partial results, and the parameters file is small enough that parameters for all jobs can be loaded into a single machine image.

A. Which Commercial Cloud?

We considered the three largest commercial cloud providers, AWS, GCP, and Microsoft Azure, each offering a variety of different services, and each with advantages and disadvantages. Providers differ in areas of pricing structure, availability of resources, and the available tools and APIs, and these features are constantly updated. A summary of key features available at the time of the experiment is shown in Table I.

The pricing structure of the resources does not vary significantly between the different providers. AWS and GCP perform instance level billing at the second level with a one minute minimum charge [18], [20] while Azure charges at the minute level for each instance [21]. Prices at the per second level can greatly reduce costs when running at a very large scale.

The cost of compute instances within each commercial cloud is a key factor. One cost management feature is the "Spot Instance" concept within AWS and the "Preemptible Instance" concept within GCP. AWS Spot Instances allow users to bid on spare Amazon EC2 computing capacity, which can result in discounts of between 50-90% from the standard on-demand price [22]. This can help control costs with the caveat that the instance can be terminated by AWS at any time, with a two minute warning. Another AWS feature that utilizes Spot pricing is AWS Spot Fleet, which allows users to manage a collection of Spot Instances with a single request. Spot Fleets allow the user to set a target capacity and will automatically attempt to launch Spot Instances to fulfill this capacity. Spot Fleets can also attempt to maintain the target capacity if some of the initial Spot Instances are terminated [23].

GCP has a feature called "Preemptible Instances" that are similar to AWS Spot instances. With GCP, users can obtain preemptible instances at a discount of up to 80% of the standard pricing. GCP users do not bid on the capacity and the discount is fixed at the price set by Google [24]. GCP preemptible instances can be terminated at any time, however there is a reduced probability of termination as the instance runtime increases. At the time of this writing, Azure does not have a comparable feature available for general use. GCP also allows users to define machine type with custom vCPUs and memory. This flexibility allows more control over the hardware requirements and costs of instances.

All three cloud providers have software solutions that can dynamically create an HPC environment within the cloud. We chose to utilize an existing software solution (described further below) in order to eliminate unnecessary development time and costs, and to focus on the scaling challenges.

We selected AWS for the implementation. At the time of the study we found that AWS offered the most effective cost-saving options for a run of this size and had the largest selection of off-the-shelf provisioning tools. The ability to specify the prices we were willing to pay via Spot instances provided more flexibility then the set prices of GCP's pre-emptible instances. Collaboration between our group and AWS associates were key enablers of the project. The partnership provided the team with more insight into the underlying infrastructure than would have been possible otherwise, and aided in finding potential bottlenecks in the deployment. In addition, AWS places default limits on the count of resources that can be provisioned. These limits are designed to help users to not exceed their own budgets as well as to provide some level of resource protection for the cloud. The collaboration facilitated raising of these limits during the study, as described in Section IV.

B. Which HPC Environment Provisioning and Workflow Management Tool?

Scaling to a massive size requires management of the lifecycle of the HPC resources in the cloud, and management

TABLE I
COMPARISON OF COMMERCIAL CLOUDS AS OF AUGUST 17, 2017

	AWS	**GCP**	**Azure**
Reference	[19]	[20]	[21]
Worldwide Availability	✓	✓	✓
"Spot" Type Instances	✓	(✓)[†]	-
Biddable "Spot" Instances	✓	-	-
"Spot" Fleet Provisioning	✓	-	-
Custom Instance Types	-	✓	-
Per Second Billing	✓	✓	-
HPC Environment Tools	✓	✓	✓

Notes:[†] GCP's "Preemptible" instances are similar to AWS's "Spot" instances.

of the scientific workflow that executes on these resources. In this study we leverage our prior work with the Automated Provisioning And Workflow Management tool (PAW) [12] that evaluated alternatives and developed infrastructure software to support both of these management tasks.

PAW is a comprehensive resource provisioning and workflow tool that automates the steps of dynamically provisioning a large scale cluster environment, running and executing user defined workflows, and de-provisioning of the cluster environment, as desired. The PAW software system launches the HPC environments with a single command, automatically performing the two key management tasks, with (optionally) no further interaction from the user. PAW is built to be modular, extensible, and workflow-agnostic, which allows the execution of any type of workflow on the provisioned HPC environment. PAW does not require the use of a specific workflow management tool. The use of PAW accelerated the testing and development of the large environment as well as the workflow itself. Without PAW, performing the scaling tests would have been difficult and tedious. Here we push the limits of PAW to provision environments much larger than in previous work and test the scalability of the software.

We used an implementation of PAW that utilizes CloudyCluster [25] and CloudyCluster Queue (CCQ) [26] to manage the provisioning and de-provisioning of the HPC environment and to enable job-based autoscaling. The job-based autoscaling allows for the required computational resources to be provisioned dynamically when the job is submitted and dynamically de-provisioned when the job has completed. Prior to developing PAW we considered different existing tools for provisioning HPC environments in the commercial cloud. These tools included CfnCluster [27], CloudyCluster [25], [28], and Alces Flight Community Edition [29].

Typically, provisioning tools only manage the provisioning of the resources and another tool is required for workflow management. There are a number of existing workflow managers, such as Tigres [30], FireWorks [31], QDO [32], SWIFT [33], and Pegasus [34] that can be used to manage the workflow in combination with PAW or one of the other provisioning tools. However, the development of the interfaces would still have been required, and there would have been management overhead due to the use of different tool interfaces. PAW utilizes CCQ and a robust configuration file interpreter for workflow management. For more information about the design choices in PAW, see [12].

C. *What Autoscaling Model?*

AWS Spot Fleets were used to provision the compute nodes of our environment. We also considered the AWS Autoscaling service with Spot Instances. However, a Spot Autoscaling group can only use one AWS instance type per group, whereas AWS Spot Fleet can utilize multiple AWS instance types within a single fleet, depending on the Spot Bid Price and "weight" specified. If the price of the requested instance type in a Spot Autoscaling group spikes before all instances are allocated, the group may not reach its target capacity. Spot Fleets are protected from such a price spike for a given instance type by allowing use of other instance types when the price rises above a set threshold. While this capability could be created using multiple Spot Autoscaling groups, it would require more management and more API calls than utilizing a single Spot Fleet. Using multiple Spot Autoscaling groups can also cause an over or under provisioning of instances as the capacities are set per group. Due to these characteristics, Spot Fleets are a better choice for our implementation.

IV. Limitations to Scaling

Our research has identified and resolved a number of limitations to massive scaling in the cloud. Several of these limitations are common to execution on all commercial clouds, including those of a shared filesystem, network limitations such as NAT instances, launching of heterogeneous instance types to control costs, HPC scheduler stability, cloud vendor user limits, and API limits. A limitation that may be specific to AWS is the dynamic pricing effect on the Spot market. A summary of these limitations and our solutions can be found in Table II. Details are described in this section.

To identify and resolve the scaling limitations, we developed a plan for experimental testing at increasingly larger scales, beginning with a modest test of 1% of our goal, or 10,000 vCPUs. Medium-sized experiments of 5,000 instances in a single environment were designed to test the limitations of the scheduler and provisioning software and other limitations. Note that a system of this size is comparable to many campus-scale computing clusters. Successful resolution of problems within a single environment laid the groundwork for the simultaneous execution of multiple 5,000-instance environments.

A. *Shared Filesystem Challenges*

The CloudyCluster component in PAW creates a shared filesystem within the provisioned HPC environment that is mounted across all of the compute instances. However, globally shared filesystems, both in the cloud and on-premise, are known to not scale well in general [35].

Our scientific HTC workflow was originally designed to dynamically generate the analysis jobs based on a specified configuration file, and each of these generated jobs required specific data that was located on the shared filesystem. Without access to the data in the shared filesystem the workflow could not execute. Our massive scale environment contains hundreds of thousands of compute jobs that could bring down even a robust shared filesystem when all jobs require access to the filesystem at the same time.

To address this problem we implemented and tested multiple different solutions. Our first try was to create a version of the workflow that submitted a parent job that copied the dataset to a local scratch filesystem, generated the analysis job files, tarred the resulting directories, and uploaded the tar file to Amazon S3. Amazon S3 is an object store service that is built to store and retrieve any amount of data from anywhere. S3 is designed to be highly scalable and access to S3 is integrated into CloudyCluster. S3 is available globally, which enables access across all Amazon regions, if needed.

In this solution, when each compute instance was assigned a job, it would copy the tar file from S3 to local instance store, extract it, and then proceed to run the scientific application. This avoids utilizing a shared filesystem while also maintaining the dynamic nature of the workflow. It also still allowed changing the data used by each job and the dynamic creation of the analysis jobs. This solution worked well for the modest scale tests of a few hundred instances. However, issues with accessing the same file were discovered with the medium 5,000-instance tests. As multiple thousands of compute instances tried to copy the same file from S3 to the local instance, the time to do this increased, and most of the copy processes eventually timed out, causing jobs to fail.

Our second implemented approach was to upload multiple copies of the data to S3 so that different instances could copy different objects from S3, which reduced the load on a single object. Implementing and testing this solution reduced the number of timeouts but did not eliminate them, which pointed to a different problem – limitations in the network infrastructure to access S3 by our application at massive scale.

B. *NAT Instance Limitations*

Our analysis determined that the timeouts in accessing S3 were due to the use of a Network Address Translation (NAT) instance. The networks that AWS utilizes for EC2 instances are called Virtual Private Clouds (VPCs) and are private by default. In order to communicate with servers or other AWS services, network address translation using, say, a NAT instance, is required. CloudyCluster handles this setup for the user by dynamically creating a NAT instance to perform these tasks. However, the very large number of instances in the VPC was exceeding the throughput capacity of our NAT instance, which was limiting our download speeds from S3.

We initially identified two different approaches to solving this issue: implement an Amazon NAT Gateway, or implement Amazon VPC Endpoints. An Amazon NAT Gateway is a managed service from AWS that handles the NAT operations without the need for a dedicated NAT instance. It supports burst of up to 10Gbps of bandwidth. There is a charge for creating a NAT Gateway and an additional cost that depends on the amount of data that is processed by the NAT Gateway which can add up quickly if transferring large amounts of data. In order to avoid this processing charge, AWS recommends

TABLE II
SCALING LIMITATIONS AND SOLUTIONS

Limitation	Solution
Shared Filesystem Scaling	Build a tar file with the dataset and experiment files and upload it to Amazon S3.
NAT Limitations	Put the data on the image to reduce traffic to the NAT Instance.
Dynamic Pricing Effects On Spot Prices	Utilize the *Diversified* Spot Fleet allocation
Heterogeneous Instance Types With Spot	Create "classes" of Spot Fleets to launch containing instances with similar characteristics
Scheduler Scalability	Eliminated unnecessary reboots of the SLURM scheduler when adding compute instances.
User Limits	Request limit increases
API Limits	Slow down the launching of new Spot Fleets and turn off unnecessary monitoring

that the user set up VPC Endpoints if the data is being transferred to other supported AWS services [36]. Hence, we decided to utilize the VPC Endpoint feature instead of the NAT Gateway solution.

VPC Endpoints allow resources within an Amazon VPC access to certain AWS resources without having to go through the NAT process. With the implementation of VPC Endpoints we were able to obtain increased download speeds and did not experience any timeouts. This solved the problem at a moderate scale, but additional testing was needed to further validate the effectiveness of upload and download to S3 at the scale required. Due to time constraints we were unable to evaluate our download/upload solutions utilizing VPC Endpoints at the massive scale we required.

The inability to fully evaluate our VPC Endpoint solution led us to implement another solution for the final experiment. Our final, implemented solution was to pre-generate the required datasets and experiment files and build them into the machine image that the compute instances used. This solution solved the network limitations of the copy, and also reduced the chance of failure when running the massive scale environment.

C. Dynamic Pricing Effects On The Spot Market

Our original Spot Fleet configuration in the initial modest-scale tests utilized the *LowestPrice* allocation strategy, which is set by the Amazon API to be the default strategy. In this mode, the Spot Fleet launches Spot Instances into the Spot Pool that has the lowest Spot Price [23]. During testing we observed scenarios in which the number of instances launching leveled out for a few minutes before increasing. Examination of the Spot pricing during the tests revealed the reason. Amazon Spot pricing changes based upon supply and demand [22], and by requesting all of our Spot Instances in the same pool, we were increasing the demand and driving the Spot Price up. Once the Spot Price reached our maximum bid price (i.e., the maximum amount we were willing to pay for a Spot Instance), our new Spot Instance requests would fail to launch because our maximum bid price was below the current Spot Price. However, the Spot Fleet would still attempt to launch more instances in the same Spot Pool for a few minutes until it reached a maximum of failed attempts. It would then move on to the Spot Pool with the next lowest price, which caused a delay in the launch of instances.

We identified the *Diversified* allocation method as a solution. This is not the default option. The Diversified provisioning method attempts to diversify where the Spot Fleet bids on the Spot Instances by launching groups of Spot Instances into each Spot Pool, as specified in the request [23]. Along with increasing the efficiency of the instance creation, this also helps to keep the price down in each Spot Pool by spreading out the Spot Instances into different Spot Pools. The Diversified allocation method helps to avoid driving up the price for a single Spot Pool, which at massive scale can be significant. If the average price for each Spot Instance is lower, the total cost will be lower.

D. Heterogeneous Instance Types With Spot Allocations

A Spot Fleet is defined by a target capacity that is the total number of capacity units desired. The provisioning of one million vCPUs requires tens of thousands of compute instances. However, a limit on Spot Fleet requests prevents specifying target capacity of more than 3,000 capacity units per fleet. This posed a challenge as a limit of 3,000 instances in a fleet requires management of an excessive number of Spot Fleets and is a limitation to scalability.

The Spot Weight parameters define the number of capacity units represented by a single Spot Instance type [23]. AWS supports two default modes for Spot Weights: Instance mode and vCPU mode. The Instance mode sets the Spot Weight for each specified instance type to 1. In this mode, the total capacity describes the total number of Spot Instances desired in the Spot Fleet. The vCPU mode sets the Spot Weight for each specified instance type to the number of vCPUs of the instance type. In the vCPUs mode the total capacity specifies the total number of vCPUs desired in the Spot Fleet. Since each EC2 Instance contains a different number of vCPUs, some instances count for more capacity units than others. However, these default Spot Fleet Weights do not work for a massive-scale environment. An alternative approach is needed.

A key observation is that Spot Fleet Weights can be a fractional value less than 1.0. The effect of fractional Spot Fleet Weights is shown in Table III. Using vCPU mode and a capacity of 3,000, the total instances per fleet is just 83. But a Spot Fleet Weight of 0.05 for a c4.8xlarge provides 20 Spot Instances for a single capacity unit. The Spot Fleet Weight of 0.05 for a c4.8xlarge provides 60,000 Spot Instances within a single Spot Fleet when the target capacity is 3,000.

Determining the optimal values for Spot Fleet Weights for each Spot Instance Type took some trial and error. During execution, a change in the Spot Fleet Weights caused a change in the types of Spot Instance types that were launched. Several

TABLE III
EXAMPLE OF EFFECT OF CUSTOM FRACTIONAL SPOT FLEET SYSTEM LIMITS WITH C4.8XLARGE INSTANCE TYPE

Instance Type (vCPU=36)	Capacity Units	Spot Weight	Total Instances Per Fleet	Spot Weight Type
c4.8xlarge	1	1	1	Instances
c4.8xlarge	3,000	36	83	vCPU
c4.8xlarge	1	0.05	20	Custom
c4.8xlarge	3,000	0.05	60,000	Custom

TABLE IV
SPOT FLEET DETAILS BY WORKFLOW CLASS

Workflow Class	Instance Types	Max Spot Bid Price	Spot Fleet Weights	Capacity Units	Number Used in 1M Run
Huge	x1.16xlarge m4.16xlarge	\$1.570 \$1.670	0.064 0.064	320	1
Large	c4.8xlarge c3.8xlarge r4.8xlarge	\$0.800 \$0.876 \$0.700	0.036 0.032 0.032	160	4
Medium	c4.4xlarge hi1.4xlarge i3.4xlarge r4.4xlarge m4.4xlarge	\$0.370 \$0.370 \$0.418 \$0.470 \$0.514	0.014 0.014 0.015 0.015 0.015	70	3
Small	m4.2xlarge m3.2xlarge c4.2xlarge	\$0.258 \$0.236 \$0.190	0.007 0.007 0.006	32	5
Tiny	c4.xlarge	\$0.100	0.025	100	1

adjustments were made in the Spot Fleet Weights so that Spot Fleets with larger vCPU counts were launched with higher frequency than those with smaller vCPU instances.

Launching larger AWS Instance types is a necessity at massive scale. For example, the use of only Spot Fleets with 4-vCPU instances requires many more Spot Instances than the use of 36-vCPU instances. For a goal of one million vCPUs, use of only the 4-vCPU instance type requires 250,000 Spot Instances. In comparison, use of only 36-vCPU instances requires 27,778 Spot Instances. Management overhead increases greatly with each additional Spot Instance, making resource management unwieldy with smaller instances. In practice, not all instance types are always available at a competitive price and a mix of instance types must be used to obtain the best price/runtime tradeoffs.

Our first Spot Weighting attempt was to specify a Spot Fleet that contained a wide range of instance types ranging from a "Huge" 128-vCPU instance to a "Tiny" 4-vCPU instance. However, it was difficult to specify the Spot Fleet Weights in such a way that would consistently select more larger expensive instances over the smaller cheaper instances. After several failed tests with the wide range of instance types, we created "workflow classes" that only contained instances that had similar characteristics. This ensured a certain capacity from each Spot Fleet and allowed more control over the launching process. The Spot Fleets details for each workflow class are shown in Table IV. The table shows the five workflow classes along with instance types and the Spot Fleets details that were utilized. Analysis, modeling, and prediction of the effects of Spot Fleet weights are areas of future work.

E. Scheduler Scalability

A medium-scale test of a few thousand compute instances revealed the next potential bottleneck: the scalability of SaltStack and SLURM. SLURM is a HPC batch scheduler through which users submit jobs. SLURM schedules the jobs on the available compute resources. According to the SLURM website, the largest SLURM cluster contains 98,304 nodes, which is much less than our maximum instance limit of 250,000 [17]. We set a goal to keep each environment under 98,304 nodes to avoid potential issues with SLURM.

As is typical of HPC schedulers, SLURM is built to be statically configured. Node configurations are known well before job scheduling, and are coded in a configuration file. In a traditional cluster, all of the IP addresses, hardware configurations, and total number of compute instances are known. But in the cloud environment all of these are decided at run time and have to be added dynamically to the SLURM configuration file as the instances launch. This new configuration file has to be pushed to each compute instance for use by SLURM.

The CloudyCluster component of PAW utilizes SaltStack to push out the configuration file to compute instances from a single Salt Master, but this is a bottleneck in our deployment. Technically, there is no limit to the number of Salt Minions (i.e., compute instances) that are supported by a single master. However, the resources required to run the Salt Master increase with the number of Salt Minions. A larger number of minions is generally handled by utilizing *salt-syndic*, but this feature was not implemented in CloudyCluster. To avoid possible resource issues and bottlenecks with the single Salt Master configuration, we created multiple CloudyCluster environments with a maximum of 5,000 compute instances per environment in order to ensure that we would stay well within the scaling limits of both SaltStack and SLURM.

The creation of multiple cluster environments is a strategic optimization of the management of the massive number of resources, and does not detract from the execution of our HTC workload. Creating multiple environments allowed us to stay well under the scaling limits imposed by our versions of SLURM and SaltStack, to maintain the dynamic nature of our workflow, and to minimize the effects of a failure. A benefit of this approach is that a catastrophic failure in a single environment does not affect other environments or the executing instances in those environments. In addition, we can partition the workflow logically and run different analyses on different environments. Since HTC jobs execute independently, their execution is independent of the creation and management of the cluster environments.

We performed rigorous testing of environments at the 5,000-instance scale. During the initial 5,000-instance tests, we noticed that the instances were not registering with the scheduler in a timely manner. In initial testing it took almost 45 minutes for all 5,000 compute instances to register with the scheduler. This is too long, because until the instances are registered they cannot run jobs and because execution of resources that are not doing useful work at such a large scale is wasteful.

Upon close examination of the instance registration process we found an issue with frequent restarts of the SLURM scheduler by CloudyCluster. When a new instance is added CloudyCluster restarts the SLURM scheduler which re-calculates the instance bitmaps used for scheduling. These restarts do not take long with just a few compute nodes, but as the number of compute instances grows so does the time for SLURM to restart. Detailed examination revealed that the restart call was being called too often, causing the delay in the registering of instances. When the extra restart call was removed the time for all 5,000 instances to register dropped from 45 to 25 minutes. The first instances registered within five minutes of the workload being submitted, and 90% of the instances were registered within twenty minutes.

F. User Limits

By default, AWS imposes limits on the number of resources that users can create. This is a common practice among commercial cloud providers as it prevents users from accidentally incurring an unexpectedly large bill while also making sure that the commercial cloud provider can support the requests of all users. All of the commercial cloud providers also provide mechanisms to increase certain limits. When running at a large scale increasing these limits is required.

The first limits that we identified that needed to be increased were the EC2 On-Demand Instance, VPC, and EC2 Spot Instance limits. These limits prevent the launching of a large number of instances and the creation of a large number of networks. A user typically fills out an online form requesting the limit increases for specific AWS services and resources along with the reason and use case for the request. Working with AWS associates resolved these issues.

Another limit encountered during the scalability tests was the VPC endpoint limit. This was discovered after implementing the VPC Endpoint solution for the issues encountered with S3. We worked directly with AWS to raise this limit.

Amazon Elastic Block Storage (EBS) limits are another crucial limit for our experiment. All EC2 instances that we used were backed by an EBS volume. By default, an AWS account is limited to 20TB of EBS storage. Each compute instance needed a 40GB EBS volume attached to it, which means 20TB would only support 500 instances. In the extreme case, using 250,0000 4-vCPU instances to reach the one million core goal, about 10PB of EBS storage is required. We requested and received an increase of the limit to 8PB of EBS storage.

G. API Limits

Like most APIs, AWS imposes limits on the number of API calls that can be issued within a certain period of time. This helps to prevent the misuse of the APIs and keeps the underlying system from becoming overwhelmed too many calls. This API throttling also allows AWS to ensure that other users are not affected by another user abusing the APIs.

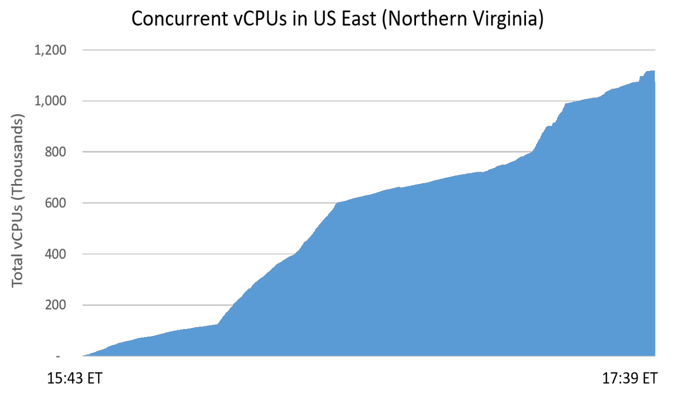

Fig. 1. Timeline of vCPU count from the start of the run to the time of the peak vCPU use count.

However, running at a massive scale with tens of thousands of instances making API calls from the same account at the same time generates a large number of legitimate API requests in a short period of time. This was an issue that we encountered.

After we had launched 33,787 Spot Instances, we noticed that our new calls to launch more Spot Instances were failing consistently. Investigation revealed that we were being throttled by the system because of the number of API calls we were making within a short time period. Post execution analysis revealed that the culprit generating a majority of the API calls was the *desribeInstances* API call.

We were utilizing the *describeInstances* call to obtain statistics about our AWS account and the EC2 instances running within it. The statistics generated by the *describeInstances* API call enabled observation of execution progress. We knew that there was the possibility of throttling of these calls at massive scale. To stop the throttling, we turned off our monitoring tools and slowed the launching of new workflow classes. Although this limited the amount of observational data we could collect during execution, it also reduced our API usage, and after a few minutes we were able to resume launching new workflow classes.

Another solution to this issue is to utilize the Amazon CloudWatch service for the monitoring instead of the *describeInstances* API calls. CloudWatch provides events that allow users to be notified as EC2 instances are launched and terminated. By using CloudWatch, the number of API calls required to monitor the account and instances is reduced, which reduces the risk of API throttling.

V. Scientific Workload

Our target scientific application is a large scale parameter sweep workflow that executes a message-passing parallel topic modeling application on multiple datasets. Topic modeling based on Latent Dirichlet Allocation is a common approach to text analysis in machine learning [37]. In this context, documents are considered to be generated by a combination of topics, each of which is a distribution over the vocabulary. Only the resulting documents are observed, and the topics and topic mixtures specific to each document are inferred. While

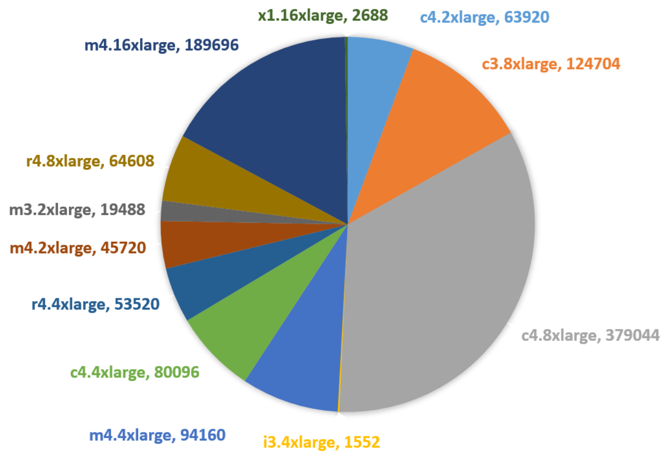

Fig. 2. The distribution of vCPUs among the instance types used at peak.

these models are widely used, evaluation of their outputs and sensitivity to the various input parameters is an active area of research [38].

Our scientific workload examines the impact of the alpha, beta, and topic count parameters, which are input parameters required by the topic modeling algorithm, on topic models of two different text datasets. The workflow performs a wide sweep of hundreds of thousands of parameter combinations. Each parameter combination is input to a topic modeling job executing Parallel Latent Dirichlet Allocation (PLDA) [39] on one of the two datasets. The first dataset includes full text conference proceedings from Advances in Neural Information Processing Systems (NIPS) [40]. The second dataset include seventeen years of abstracts from a wide variety of computer science publications that was provided to us from Elsevier Scopus. Each job outputs a set of topic vectors which future work can utilize to explore the sensitivity of topic quality with respect to input parameters.

The topic modeling workflow has a number of characteristics that are typical of parameter sweep workflows and that impact the design of our system. First, each job is a parallel application that requires MPI. Secondly, most of the jobs use the same input data and are distinguished from each other by minor parameter changes that can be captured in small configuration files. We take advantage of these characteristics by loading the common input data into the machine image, so as to avoid requiring each job to download the same data file during the execution and placing unnecessary load on the NAT instances as described previously.

VI. Execution and Evaluation

In this section we discuss the execution and evaluation of the scientific workflow, including detailed discussion of the technical aspects and cost analysis necessary to achieve our goal of at least one million concurrent vCPUs.

A. Experimental Execution

The workload executed across a set of cluster environments. Each of these environments was configured identically with the exception of the selection of instance types used. We prepared

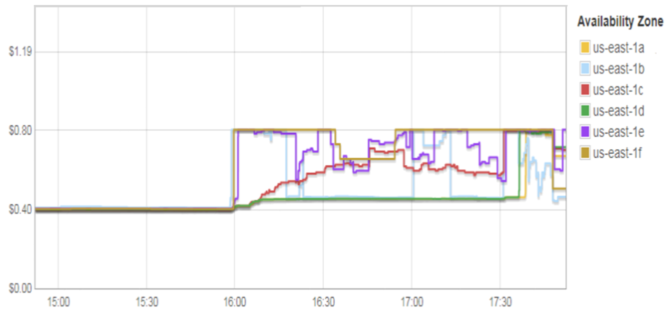

Fig. 3. The effect of our experiment on the Spot Prices of the c4.8xlarge instances in each Availability Zone. The chart shows that prices were near US $0.40 for all Availability Zones prior to the start of the experiment, and that prices rose by as much as a factor of two during the experiment.

and launched the core components only of forty cluster environments, which provided extra environments ready to launch instances in the case that some environments failed.

The environments were launched in two different stages. During the first stage, PAW launched the minimal set of standard cluster environmental components including the Control, Scheduler, Login, and NAT instances. To accomplish this, PAW utilized a common configuration file with a pre-defined environment template. The second phase, which included launching 5,000 compute instances in each environment, was also initiated by PAW using configuration files with different topic modeling workflow configurations specified. During this stage, PAW processes the topic modeling workflows and then automatically submits them to the running environments via CCQ. CCQ then handles the dynamic provisioning of the requested compute instances utilizing AWS Spot Fleets.

The configuration files used in the second stage specify the experiment to run and which workflow class to launch. Each workflow class specifies different instance types and Spot Fleet configurations, as shown in Table IV. These classes are color coded to show which Amazon Instance Types were specified in each workflow class.

The first workflow class was submitted to the first environment at 3:43PM EST. During the next two hours we successfully launched 1 Huge, 4 Large, 3 Medium, 5 Small, and 1 Tiny workflow classes, for a total of 14 workflow classes. We had previously prepared 40 environments for launching workflow classes in case of failures of various kinds. However, only 14 environments were required to obtain a total number of executing vCPUs of more than one million.

Some API throttling during the experiment caused the pace of launching of new instances to be slowed considerably. This can be observed in Fig. 1 around the 600,000-vCPU mark. As mentioned in the API Limits section, to move past this issue we lengthened the time between launches and turned off some of our monitoring tools to decrease the number of API calls being made.

At 5:39PM EST, we reached the peak vCPU count of 1,119,196 vCPUs running within 49,925 Spot Instances spread across 12 instance types. A breakdown of the total number of vCPUs per instance type at the peak time is shown in Fig. 2.

TABLE V
INSTANCE TYPE CLASSES AND AVERAGE vCPU HOUR COSTS

Workflow Class	Instance Type	vCPUs	vCPU hour Cost
Huge	x1.16xlarge	64	$0.0242
	m4.16xlarge	64	$0.0193
Large	c4.8xlarge	36	$0.0150
	c3.8xlarge	32	$0.0188
	r4.8xlarge	32	$0.0186
Medium	i3.4xlarge	16	$0.0257
	r4.4xlarge	16	$0.0218
	m4.4xlarge	16	$0.0197
	hi1.4xlarge	16	$0.0187
	c4.4xlarge	16	$0.0131
Small	m4.2xlarge	8	$0.0213
	m3.2xlarge	8	$0.0204
	c4.2xlarge	8	$0.0132
Tiny	c4.xlarge	4	$0.0128

The entire execution was contained within a single Amazon Region, N. Virginia. This is of note because the AWS Spot market utilizes unused capacity in order to fulfill instance requests. The provisioning of more than 1.1 million vCPUs in a single Amazon region highlights the scalability of our approach and the massive capability of the cloud for scientific computation.

We were able to fully execute, from start to finish, just under a half a million jobs submitted by our topic modeling workflow in under two hours. On a local shared cluster resource, running half a million jobs would have taken several days or weeks.

B. Cost Analysis

An important question when considering the use of commercial cloud resources is the cost comparison between the commercial cloud and local resources. These comparisons are complex and depend on a variety of factors such as the size, utilization, and maintenance costs associated with the local resource along with the characteristics of the workflow being executed. Cost estimation done at a representative university shows on-premise computational costs to be well under US $0.02 cents per core hour, regardless of job type (serial, shared memory parallel, distributed parallel). This estimate includes costs for hardware, software, space, power, cooling, labor, network etc, but excludes user-facing services such as user support and research computing facilitation [41].

We set our maximum Spot Bid Price per instance type at the price that would keep us close to US $0.02 per vCPU hour. This strategy helped to keep the costs down and narrow down the choices of instance types. Another strategy that we employed to keep costs down was to utilize the Spot Fleet Weighting feature which allowed us to steer our Spot Fleet to launch certain more desirable types of instances over other less desirable instances. Our maximum Spot Bid Price and our Spot Fleet Weights per instance type are shown in Table IV.

Another cost tradeoff when running at a large scale is that when utilizing a massive number of resources users begin to compete against themselves, which drives up Spot Pricing. During our experiment, we found that launching many instances raised the Spot Price up to our maximum Spot Bid Price for certain instance types utilized. Fig. 3 shows Spot prices for the c4.8xlarge instance type in the six Availability Zones. Fig. 3 also shows that the average instance price for the period during the massive run is as much a twice the price during the hour prior to the run.

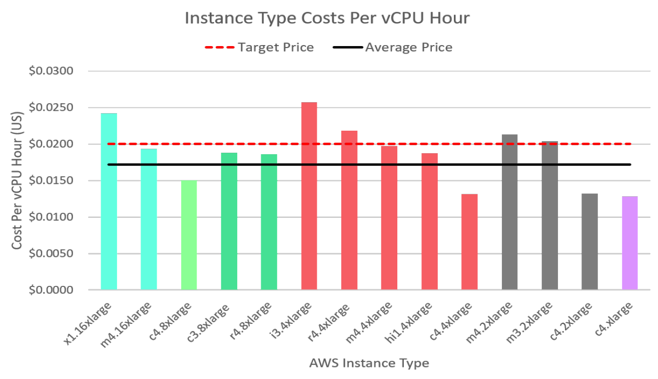

Fig. 4. The cost per vCPU hour of each AWS Instance Type used. The solid line is the average price per vCPU hour across all Instance Types and the dotted red line is the target price per vCPU hour.

During our experiment, we used a total of 1,832,923 vCPU hours at an average cost of $0.0172 per core hour. While other costs may need to be considered when running workflows in AWS, for our scientific application the costs for network egress out of AWS and data storage within AWS were negligible. The total cost for the computation resources was $32,423 for the two hours of execution. A summary of the cost per vCPU hour by instance type and "workflow class" is shown in Table V. A graph showing the cost per vCPU hour, the target price, and average price for all instance types is shown in Fig 4.

VII. CONCLUSIONS AND FUTURE WORK

In this paper we present the challenges and solutions associated with launching a massive scale computational cluster environment in the AWS commercial cloud. We utilized the automated Provisioning And Workflow management tool (PAW) [12] for the lifecycle tasks of cluster provisioning, workflow execution, and cluster de-provisioning. Utilizing PAW, we were able to run our topic modeling workflow on 1,119,196 vCPUs simultaneously with minimal user input at an average cost of $0.0172 per vCPU hour.

A number of limitations to massive scaling on the commercial cloud were discovered and resolved in this research. Several of the limitations we identified and resolved are generally common to execution on all commercial clouds. These include those of a shared filesystem, network limitations such as NAT instances, launching of heterogeneous instance types to control costs, HPC scheduler stability, cloud vendor user limits, and API limits. A limitation that may be specific to AWS is the dynamic pricing effect on the Spot market. We provide a detailed cost analysis for running on AWS, and describe how costs may be lowered with smaller workloads using the same technologies.

The execution of a massive HTC application in the commercial cloud is a promising demonstration of the use of the cloud for scientific applications, and suggests opportunities for the emergence of new synergies between the scientific community and commercial cloud providers.

Acknowledgements

We acknowledge support from Amazon Web Services, Omnibond, HPCC Systems, LexisNexis Risk Solutions, Elsevier Scopus, GAANN award #P200A150310, and NSF #1228312.

References

[1] Columbus, Louis. "Forrester's 10 Cloud Computing Predictions For 2018." Forbes, Forbes Magazine, 20 Nov. 2017, www.forbes.com/sites/louiscolumbus/2017/11/07/forresters-10-cloud-computing-predictions-for-2018/#2719d1694ae1.

[2] "Top 500 List, http://www.top500.org".

[3] Roloff, Eduardo, et al. "High Performance Computing in the cloud: Deployment, performance and cost efficiency." *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*, 2012, doi:10.1109/cloudcom.2012.6427549.

[4] Mehrotra, Piyush, et al. "Performance evaluation of Amazon EC2 for NASA HPC applications." *Proceedings of the 3rd workshop on Scientific Cloud Computing Date - ScienceCloud 12*, 2012, doi:10.1145/2287036.2287045.

[5] Gupta, Abhishek, and Dejan Milojicic. "Evaluation of HPC Applications on Cloud." *2011 Sixth Open Cirrus Summit*, 2011, doi:10.1109/ocs.2011.10.

[6] Iosup, A, et al. "Performance Analysis of Cloud Computing Services for Many-Tasks Scientific Computing." *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 6, 2011, pp. 931945., doi:10.1109/tpds.2011.66.

[7] Tomic, D., et al. "Running HPC applications on many million cores Cloud." *2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2017, doi:10.23919/mipro.2017.7973420.

[8] Gupta, Abhishek, et al. "Evaluating and Improving the Performance and Scheduling of HPC Applications in Cloud." *IEEE Transactions on Cloud Computing*, vol. 4, no. 3, Jan. 2016, pp. 307321., doi:10.1109/tcc.2014.2339858.

[9] Gupta, A., et al. "Improving HPC Application Performance in Cloud through Dynamic Load Balancing." *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, 2013, doi:10.1109/ccgrid.2013.65.

[10] Gonzalez, Patricia, et al. "Using the Cloud for Parameter Estimation Problems: Comparing Spark vs MPI with a Case-Study." *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2017, doi:10.1109/ccgrid.2017.58.

[11] Mariani, Giovanni, et al. "Predicting Cloud Performance for HPC Applications: A User-Oriented Approach." *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2017, doi:10.1109/ccgrid.2017.11.

[12] Posey, et al. "Automated Cluster Provisioning And Workflow Management for Parallel Scientific Applications in the Cloud." *MTAGS'17* (2017).

[13] Jonas, Eric, et al. "Occupy the cloud." *Proceedings of the 2017 Symposium on Cloud Computing - SoCC 17*, 2017, doi:10.1145/3127479.3128601.

[14] Holzman, Burt, et al. "HEPCloud, a New Paradigm for HEP Facilities: CMS Amazon Web Services Investigation." *Computing and Software for Big Science*, vol. 1, no. 1, 2017, doi:10.1007/s41781-017-0001-9.

[15] Barrett, Alex. "220,000 cores and counting: MIT math professor breaks record for largest ever Compute Engine job." Google Cloud Platform Blog, Google, 20 Apr. 2017, cloudplatform.googleblog.com/2017/04/220000-cores-and-counting-MIT-math-professor-breaks-record-for-largest-ever-Compute-Engine-job.html. Accessed 24 Oct. 2017.

[16] Sadooghi, Iman, et al. "Achieving Efficient Distributed Scheduling with Message Queues in the Cloud for Many-Task Computing and High-Performance Computing." *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2014, doi:10.1109/ccgrid.2014.30.

[17] "Large Cluster Administration Guide." Slurm Workload Manager, *SchedMD*, slurm.schedmd.com/big_sys.html. Accessed 24 Oct. 2017.

[18] "Announcing Amazon EC2 per second billing." Amazon Web Services, Inc., 2 Oct. 2017, aws.amazon.com/about-aws/whats-new/2017/10/announcing-amazon-ec2-per-second-billing/. Accessed 24 Oct. 2017.

[19] "Amazon Web Services (AWS) - Cloud Computing Services." *Amazon Web Services, Inc.*, aws.amazon.com/. Accessed 3 Sept. 2017.

[20] "Extending per second billing in Google Cloud." Google Cloud Platform Blog, *Google*, 26 Sept. 2017, cloudplatform.googleblog.com/2017/09/extending-per-second-billing-in-google.html. Accessed 24 Oct. 2017.

[21] "Linux Virtual Machines Pricing." Pricing - Linux Virtual Machines — Microsoft Azure, *Microsoft*, azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/. Accessed 24 Oct. 2017.

[22] "Amazon EC2 Spot Instances." *Amazon Web Services, Inc.*, aws.amazon.com/ec2/spot/. Accessed 24 Oct. 2017.

[23] "How Spot Fleet Works." How Spot Fleet Works - Amazon Elastic Compute Cloud, *Amazon Web Services*, docs.aws.amazon.com/AWSEC2/latest/UserGuide/spot-fleet.html. Accessed 24 Oct. 2017.

[24] "Preemptible VM Instances — Compute Engine Documentation — Google Cloud Platform." Google, *Google*, cloud.google.com/compute/docs/instances/preemptible. Accessed 24 Oct. 2017.

[25] "Self Service HPC In The Cloud." *CloudyCluster*, www.cloudycluster.com/. Accessed 3 Sept. 2017.

[26] "Using CCQ." CloudyCluster Documentation, *Omnibond*, docs.cloudycluster.com/home/submitting_and_managing_jobs/using_ccq.htm. Accessed 6 Nov. 2017.

[27] "CfnCluster." CfnCluster CfnCluster 1.3.1, *Amazon Web Services*, cfncluster.readthedocs.io/en/latest/. Accessed 24 Oct. 2017.

[28] Posey, Brandon M. "Dynamic HPC Clusters within Amazon Web Services (AWS)." *Clemson University*, 2016, tigerprints.clemson.edu/all_theses/2392. Accessed 3 Sept. 2017.

[29] "Flight Appliance Documentation - 2017.1r1." Flight Appliance Documentation - 2017.1r1 flight-Appliance-Docs 1.0 documentation, Alces Flight, docs.alces-flight.com/en/stable/index.html. Accessed 24 Oct. 2017.

[30] Hendrix, Valerie, et al. "Tigres Workflow Library: Supporting Scientific Pipelines on HPC Systems." *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2016, doi:10.1109/ccgrid.2016.54.

[31] "Is FireWorks for me?" Introduction to FireWorks (Workflow software) FireWorks 1.5.2 documentation, materialsproject.github.io/fireworks/index.html. Accessed 3 Sept. 2017.

[32] "QDO." *QDO Overview*, bitbucket.org/berkeleylab/qdo. Accessed 4 Sept. 2017.

[33] "A simple tool for fast, easy scripting on big machines." *The Swift Parallel Scripting Language*, swift-lang.org/main/. Accessed 4 Sept. 2017.

[34] Vahi, Karan, et al. "Pegasus WMS."*Pegasus WMS*, 24 Aug. 2017, pegasus.isi.edu/. Accessed 4 Sept. 2017.

[35] Apon, A.w., et al. "Sensitivity of Cluster File System Access to I/O Server Selection." 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID02), 2002, ieeexplore.ieee.org/document/1540455/. Accessed 6 Nov. 2017.

[36] "Amazon Virtual Private Cloud (VPC).' *Amazon Web Services, Inc.*, Amazon Web Services, aws.amazon.com/vpc/pricing/. Accessed 24 Oct. 2017.

[37] Blei, David M., Andrew Y. Ng, and Michael I. Jordan. "Latent dirichlet allocation." *Journal of machine Learning research* 3.Jan (2003): 993-1022.

[38] Blei, David M. "Probabilistic topic models." *Communications of the ACM* 55.4 (2012): 77-84.

[39] Wang, Yi, et al. "PLDA: Parallel Latent Dirichlet Allocation for Large-Scale Applications." *AAIM* 9 (2009): 301-314.

[40] Amir Globerson, et al. "Euclidean Embedding of Co-occurrence Data." *JMLR* 8, 2007.

[41] Neeman, Henry. personal communication, 6 Nov. 2017.