December 2019

Evaluating and Enabling Scalable High Performance Computing Workloads on Commercial Clouds

Brandon M. Posey
Clemson University, bposey@g.clemson.edu

Evaluating and Enabling Scalable High Performance Computing Workloads On Commercial Clouds

---

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Science

---

by
Brandon Michael Posey
December 2019

Abstract

Performance, usability, and accessibility are critical components of high performance computing (HPC). Usability and performance are especially important to academic researchers as they generally have little time to learn a new technology and demand a certain type of performance in order to ensure the quality and quantity of their research results. We have observed that while not all workloads run well in the cloud, some workloads perform well. We have also observed that although commercial cloud adoption by industry has been growing at a rapid pace, its use by academic researchers has not grown as quickly. We aim to help close this gap and enable researchers to utilize the commercial cloud more efficiently and effectively.

We present our results on architecting and benchmarking an HPC environment on Amazon Web Services (AWS) where we observe that there are particular types of applications that are and are not suited for the commercial cloud. Then, we present our results on architecting and building a provisioning and workflow management tool (PAW), where we developed an application that enables a user to launch an HPC environment in the cloud, execute a customizable workflow, and after the workflow has completed delete the HPC environment automatically. We then present our results on the scalability of PAW and the commercial cloud for compute intensive workloads by deploying a 1.1 million vCPU cluster. We then discuss our research into the feasibility of utilizing commercial cloud infrastructure to help tackle the large spikes and data-intensive characteristics of Transportation Cyberphysical Systems (TCPS) workloads. Then, we present our research in utilizing the commercial cloud for urgent HPC applications by deploying a 1.5 million vCPU cluster to process 211TB of traffic video data to be utilized by first responders during an evacuation situation. Lastly, we present the contributions and conclusions drawn from this work.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In recent years, commercial cloud providers such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP) have been growing and maturing quickly. According to Forrester, in 2018 the global public cloud market will grow at a 22% compound annual growth rate [17]. While these commercial clouds provide the "allure" of unlimited resources, a variety of cutting edge resources, and high availability, academic researchers have been slow to adopt commercial clouds for their workloads. There are a number of reasons for this including, steep learning curves, funding challenges, and performance differences [14].

Traditionally, academic researchers are accustom to executing their workloads on a High Performance Computing (HPC) cluster. These HPC clusters are typically shared with other academic researchers and only have a finite amount of resources and resource types to offer the researcher. Utilizing these types of HPC resources generally requires minimal effort by the researcher as they do not have to have a deep understanding as to how the HPC cluster is deployed and managed. These HPC clusters are also generally highly optimized for HPC applications and parallel workloads which can increase the performance of the researcher's applications.

In contrast, the commercial cloud is a more general purpose computing environment where not all the components are specifically optimized for HPC. As a number of studies have shown, HPC application performance on the commercial cloud is not as good as it is on a traditional HPC cluster [57, 36, 33, 83, 93]. There are a wide variety of configuration options available to commercial cloud users, but since each commercial cloud is different, users have to spend large amounts of time learning how to properly configure each cloud to get the best performance. This is not feasible for many academic researchers as they do not have the time to take away from their research to learn how each cloud operates. Not only do they need to have an understanding of how the cloud works, they also must know how to deploy an HPC cluster environment on each cloud which only complicates the process. However, for academic researchers with certain HPC workloads the potential impact of utilizing the cloud is large. Even though the performance may not be as good as a local resource, an academic researcher can have full access to all the resources on the cloud, eliminating queue wait times and allowing them to shorten the overall time-to-science. Researchers also have the ability to scale their workloads up to a much larger scale on the commercial cloud due to the large amount of available resources.

1.1 Problem Statement

It is the goal of this work to examine how the architecture of different commercial clouds can affect the performance of different HPC workloads, explore how to better enable commercial cloud access to academic researchers, and to push the limits of the scalability of the commercial cloud.

1.2 Research Questions

We propose, evaluate, and answer the following research questions within this dissertation:

- What are the limitations to scaling HPC environments within a commercial cloud?
- How do the architectures of different commercial cloud environments, such as AWS,

Azure, and GCP effect HPC environment performance?

- How can these architectural differences be utilized to optimize HPC environment deployment, cost, and workload performance?
- How can certain workload and data characteristics effect the ability to utilize commercial cloud resources?

1.3 Research and Contributions

Our first contribution in this area is to identify which types of HPC applications perform well on Amazon Web Services (AWS). In this work, we observe that many non-network intensive applications, such as High Throughput Computing applications (HTC), perform decently on AWS compared to the local resources. We also observe that one application in particular, MPIFFT, performs exceedingly poorly. We investigate this more to determine if we can find a potential configuration where MPIFFT will perform closer to that of the local resources. In performing these experiments, we find that after reaching 32 processes, in any combination of processes and nodes, the performance started to degrade quickly.

Next, we set out to create a tool to help automate the deployment of an HPC cluster environment in the commercial cloud. This tool could then be utilized to execute HPC applications that can benefit from executing on the cloud. Our work on this tool, the automated Provisioning And Workflow management tool (PAW), was presented at *MTAGS17* [75]. During the initial design phases of PAW, we find that while there are many tools that manage either the workflow or the provisioning of the resources, there are very few tools that perform both functions outside of a few domain specific examples. For researchers to adopt the commercial cloud, a generalized end-to-end solution is required. PAW is built to fill this gap by providing a single interface for resource provisioning, workflow execution, and resource de-provisioning.

After building PAW, we look to examine and push the limits of the scalability of the commercial cloud. To do this, we build a cluster containing 1,119,196 vCPUs within AWS utilizing PAW. Our results, which were presented at the *CCGRID 2018* [72], describe a list of seven limitations to scaling and propose potential solutions to each limitation. We utilize this environment to execute just under a half a million CPU intensive topic modeling parameter sweep applications in just under two hours. On a traditional shared HPC resource executing a half a million jobs would have taken multiple days or even weeks.

Next, we examine the usecase of migrating the expanding field of Transportation Cyberphysical System (TCPS) to the commercial cloud utilizing the same Infrastructure as Code (IaC) paradigm used in our previous work [76]. In this work, we examine in detail the different layers of the TCPS systems and how we can utilize the IaC paradigm to help simplify the deployment and management of TCPS in the commercial cloud. We also discuss the feasibility of migrating the different layers of TCPS to the commercial clouds and the cloud native options for each. We find that the most attractive candidate layer of a traditional TCPS system to be migrated to the cloud is the batch processing layer as it has no real-time processing requirements and the jobs are designed to run non-interactively.

After examining the data intensive nature of TCPS, we examine the performance and scalability of a large-scale data intensive workflow in the commercial cloud in regards to an urgent computing scenario. To do this, we utilize PAW to deploy a 1.5 million vCPU cluster and execute our custom designed workflow in GCP to process 211TB of traffic video data. This simulates the data processing required to monitor the evacuation of a large region in preparation for an impending event. Our results will be presented at the *Urgent HPC 2019* Workshop at SC19 [74] and describe the architecture of our workflow and how the commercial cloud can be utilized for urgent computing tasks on-demand. We analyze the entire 211TB of video in approximately 8 hours which includes the infrastructure creation and deletion.

1.4 Dissertation Organization

This dissertation is organized as follows. Chapter 2 discusses the background information related to this work. Chapter 3 covers the evaluation of the performance of different types of workloads in the commercial cloud. Chapter 4 introduces PAW, our resource provisioning and workflow management tool. Chapter 5 discusses provisioning and executing a 1.1 million vCPU HPC environment within the cloud utilizing PAW to execute a CPU-intensive topic modeling workload. Chapter 6 discusses the challenges of migrating different analysis layers of data-intensive Transportation Cyberphysical Systems (TCPS) workloads to the commercial cloud. Chapter 7 discusses our work addressing the challenges of utilizing the commercial cloud for urgent HPC needs and describes how to execute a data intensive traffic analysis workload utilizing 1.5 million vCPUs to process 211TB of video data. Lastly, Chapter 8 will present our conclusions and direction for future work.

Chapter 2

Background

In this section we discuss background information that is relevant to the completed research and proposed research questions.

2.1 High Performance Computing (HPC) Environments

High Performance Computing (HPC) can be defined as aggregating computing power to deliver better performance than could be achieved on a single machine. HPC environments or clusters typically consist of clusters of computers that each have their own operating system (typically a Linux distribution), memory, and storage. These computers are typically connected by a high performance low-latency network to ensure the highest performance. HPC environments are typically utilized by researchers from academia, government, and industry to solve large complex problems that require a lot of computational power.

2.1.1 Terminology

In an HPC environment, each of the computers within the environment is referred to as a *node*. Typically most of the nodes within an HPC environment are referred to as *compute nodes*. These compute nodes are the nodes that perform all of the jobs that are submitted to the HPC environment. A *job* in the context of an HPC environment is a unit of work to be performed. This can range from an application that executes on a single node or an application that executes across multiple nodes. The scheduling and mapping of these jobs to the available compute nodes in the environment is typically done by an HPC batch scheduler.

An *HPC batch scheduler* is a piece of specialized software that handles the distributing of jobs along with monitoring the jobs and the handling of the job output files. Most HPC batch schedulers accept jobs in the form of *job scripts*. Job scripts are simply scripts that execute a set of shell (e.g. bash, sh, zsh) commands on the node or nodes that the scheduler gives it. Each HPC batch scheduler has a specific set of special scheduler directives that can be included in the job script in order to instruct the HPC batch scheduler how they would like their job executed. Example parameters are number of nodes to run on, amount of memory, type of GPU, etc.

2.1.2 Composition

A typical HPC environment consists of a combination of a few different types of nodes. These nodes are typically compute nodes, storage nodes, a scheduler node, and a login nodes. As previously mentioned compute nodes are the nodes that perform the computational tasks and typically have the most robust hardware. Storage nodes typically host some type of shared filesystem that can be accessed by all of the other nodes within the cluster. The type of shared filesystem will vary from environment to environment, however typical filesystems are usually either created by utilizing the Network File System (NFS) software or a parallel filesystem such as OrangeFS or Lustre. The scheduler node runs the specialized HPC batch scheduling software that distributes the jobs to the compute nodes within the environment. The login instance is the user facing instance that users will log in to and utilize to submit their jobs.

2.2 Computational Codes

There are a wide variety of computational codes that are in use by researchers today. These codes range from single threaded applications that execute on only a single core at a time all the way to parallel applications that can be spread across multiple cores and machines. While the codes are to numerous to list, there are a few general things about computational codes that should be noted.

While there are multiple types of computational codes that are in use today, we want to discuss two particular types of computational codes: High Performance Computing (HPC) Applications and High Throughput Computing (HTC) applications.

2.2.1 High Performance Computing (HPC) Applications

HPC applications are traditionally highly parallel codes that utilize the Message Passing Interface (MPI) library to allow multiple processes to communicate with each other. HPC applications tend to be tightly coupled and rely heavily on the network infrastructure in order to exchange information. This is traditionally why many HPC environments utilize a low-latency network, because the longer that the processes take to communicate, the longer the application will take to execute. Examples of some HPC applications are LAMMPS [71], Gromacs [12], and OpenFoam [103].

2.2.2 High Throughput Computing (HTC) Applications

In contrast to HPC applications, High Throughput Computing (HTC) applications are typically loosely coupled and tend to only execute on one or two nodes within the environment. HTC applications can be parallel applications as well, however typically the parallelism is contained on a single node to enable faster processing. HTC applications generally take in multiple parameters as input and are executed with a large number of different input combinations. This allows researchers to determine how changes in the parameters can affect the outcomes of the application.

2.3 Cloud Computing

Cloud computing is referred to as the on-demand delivery of different types of IT services through a platform via the internet with a pay-as-you-go pricing model. The services offered vary quite a bit depending on which commercial cloud is being used. Along with this variation in services, different commercial cloud providers also offer different types and combinations of hardware. Commercial clouds are similar to HPC environments as they are simply a large collection of computers that are controlled by software. However, unlike HPC environments that are designed to execute parallel workloads, commercial clouds are designed to be more general purpose and typically do not utilize a low-latency network.

2.3.1 Terminology

In a commercial cloud environment, each server is called an *instance* instead of a node. This is due largely in part to the fact that most commercial cloud environments are *virtual machine based*, that is the user is allocated a virtual machine (VM) managed by some *hypervisor* software instead of having access to the bare metal like they would in an typical HPC environment. The additional layer of the hypervisor does introduce some overhead which can lead to a possible decrease in the performance of certain components such as the network. Another area where the commercial cloud differs from the HPC environment is that typically in an HPC environment the number of processes a node can execute is referred to as the number of "cores" whereas in the cloud this is referred to as the number of "vCPUs". The formal definition for a vCPU is that it is a hyperthread of an Intel Xeon core [86].

2.3.2 Major Providers

When looking for a commercial cloud provider there are many different providers from which to choose and each one has certain strengths and weaknesses. For our research we have chosen to focus on two of the major commercial cloud providers Amazon Web

Services (AWS) and Google Cloud Platform (GCP).

2.3.2.1 Amazon Web Services (AWS)

Amazon Web Services (AWS) is the largest commercial cloud computing provider in the world [27]. AWS provides a number of services that cover all types of computing infrastructure from databases to machine learning to IoT. The core building block of a majority of the AWS services is their Elastic Compute Cloud (EC2) service. EC2 is a service that provides the compute infrastructure and allows users to provision instances (VMs) within AWS. EC2 is one of the services that is utilized by our research to construct our experimental HPC environments.

EC2 allows users to choose from a number of pre-configured instance types when launching their instances. There is no way to customize the amount of resources that come with each instance type so users are limited to the options that AWS provides. However, AWS does have a number of different instance type classes: General Purpose, Compute Optimized, Memory Optimized, Accelerated Computing, and Storage Optimized. Each class has different characteristics and hardware that make it more suitable for specific tasks.

EC2 also has a concept of a “Spot Market” where users can bid on unused EC2 instance capacity for discounts of up to 90% of the regular On-Demand price for an instance type. Within AWS the special instance types are called *Spot instances*. For Spot instances, the pricing per instance is based upon a bid structure that can yield discounts up to 90% [86]. The user sets the maximum price that they are willing to pay for the instance to run and then if the current Spot price, which is set by AWS, is below the price set by the user the instance will launch and run. However, once the price exceeds the price set by the user, the instance will terminate with only a two minute warning. Spot instances utilize the same instance type structure as traditional On-demand instances, and as such users can not customize the number of vCPUs or RAM that the instances have beyond what AWS offers. There is no timelimit on how long a Spot instance can run and the price a user is paying for any particular Spot instance can change over the run time of the instance. This dynamic pricing makes the overall cost planning more difficult as users can calculate the maximum that they will spend but will not know the actual cost until after the execution.

2.3.2.2 Google Cloud Platform (GCP)

Google Cloud Platform (GCP) is another large cloud provider in addition to AWS and Azure. While GCP does have a smaller marketshare than AWS and Azure, they do have a large number of services as well. Similar to AWS, GCP has services for everything ranging from database to IoT to machine learning but the implementation of these services is fundamentally different between the two clouds.

GCP has a Compute Engine service that users can utilize to launch instances (VMs) similar to AWS's EC2. However, in GCP users are not limited to a set of instance types when launching instances. Instead users are allowed to specify the number of vCPUs, amount of memory, and type of GPU in any supported combination. This allows researchers to better tailor the resources being provisioned in the cloud to more closely match their existing resources.

GCP does not have a "Spot Market", however they do offer a similar feature called "Preemptible Instances". Their functionality is similar to that of AWS's Spot instance in that they can be shutdown at any time with only a short notice. However, unlike Spot instances the pricing of preemptible instances is fixed for the duration of the instance runtime and is set by Google. This fixed discount can range up to 80% depending on the configuration of the GCP instance and the cost will not change while the instance is running. This makes it easier to calculate the actual cost of the execution which helps with cost planning. GCP preemptible instances also allow users to utilize custom instance types which can allow for even more cost savings. Users only need to request the number of vCPUs and the amount of RAM that they need for their workflow to execute effectively. Where Spot instances can run for an unlimited amount of time, GCP preemptible instances can run for a maximum of 24 hours at a time [30]. GCP preemptible instances meet our needs as they are customizable and since our workflow consists of a large number of small independent batch jobs it does not matter if a single instance is preempted as the other running jobs are not affected.

2.3.2.3 Microsoft Azure

Microsoft Azure is another one of the large cloud providers in addition to AWS and GCP. The strength of the Azure platform lies in its inherent compatibility and interfaces with Microsoft's other enterprise services that are already in use in many corporations. Similarly to the other large cloud providers Azure offers a large number of services that cover a wide spectrum of services. However, the implementation of these services vary from the other providers, especially with the additional tie-ins to traditional Microsoft infrastructure such as SQL Server and Microsoft Power BI.

Azure has a similar instance type to GCP's "Preemptible Instances" instance type called *Low-Priority instances*, however these instances can only be utilized as a part of Azure VM Scale Sets or Azure Batch which limits their flexibility. This also puts Low-Priority instances behind AWS and GCP which allow users to utilize these special instances with their standard compute services. Low-Priority instances also have a fixed discount that is set by Microsoft and can range from 60%-80% and the instances will only launch when there is free capacity of the requested non-modifiable instance types [58].

2.4 Survey on Resource and Workflow Management Tools

There exists a large number of resource and workflow management tools, each of which have different strengths and weaknesses. Here we present a survey of both resource management and workflow management tools.

2.4.1 Resource Management Tools

There are tools that solely focus on the provisioning of HPC environment resources on the cloud such as CloudyCluster, CfnCluster, Alces Flight, StarCluster, and CycleCloud.

Each of these tools allows users to specify and create HPC environments that consist of a wide variety of components that can be customized to the user's liking. However, these tools do not have the ability to manage or submit workflows or jobs.

2.4.1.1 CloudyCluster

CloudyCluster is a commercial available tool that provides self-service HPC in the commercial cloud through a web-based UI [68]. CloudyCluster allows users to dynamically provision an HPC environment that closely resembles traditional HPC environments found at Universities and National Labs. The created HPC environments can contain a combination of HPC schedulers, login instances, shared filesystems, parallel filesystems, NAT instances, and compute instances. CloudyCluster also contains a meta-scheduler called CloudyCluster Queue (CCQ) [67] that enables job-driven autoscaling within CloudyCluster. CCQ works by parsing the submitted *job script* and determining the resources requested to execute the job. This information is parsed directly from the supported HPC scheduler specific options (cores/memory requirements) or through CCQ specific directives specified within the job script. After determining the required resources, CCQ dynamically provisions the resources of the required type (GPU, large memory, etc) and once they are running submits the job to the HPC scheduler. Upon the completion of the job, if there are no other jobs to execute, the provisioned resources are de-provisioned in order to save cost.

However, CloudyCluster in its current form does not support workflow management. While it allows users to submit job scripts which themselves could contain workflows, it does not allow a way to automate both the resource creation and workflow submission. The user is still required to manually submit the job script to the HPC scheduler in order to start the workflow. This means that users cannot script their full workflow from start to finish which is not appealing for researchers with long running jobs or that have to launch and tear down multiple HPC environments for their research.

2.4.1.2 CfnCluster

CfnCluster or "cloud formation cluster" is a free framework that deploys and maintains HPC environments on AWS [87]. CfnCluster is a commandline based utility that is driven by AWS's CloudFormation service. CfnCluster allows users to create an HPC environment within AWS that contains a "master server" and a compute fleet. The master server is the instance that runs the HPC scheduler software and the Network Filesystem (NFS) software for sharing files between instances. It is also the instance where the user will login to submit their job scripts. There is not a lot of customization that can be done to the architecture of the system as the configuration options are limited. CfnCluster does support dynamic autoscaling of the number of compute instances based upon the number of jobs in the HPC scheduler queue. However, this autoscaling can only add more instances of the same type to the environment so if a user creates an environment that utilizes non-GPU instances and then has a job that requires a GPU instance they will have to create a new environment in order to get a GPU instance.

Similarly to CloudyCluster, CfnCluster does not provide any workflow submission capabilities beyond the ability for a user to submit a job script. Users still must manually submit their job script to the HPC scheduler to start their workflow. Another possible issue that can arise for workflow management is the probability that users will need more than one type of resource during their workflow. CfnCluster does handle autoscaling of single instance types, but does not handle multiple instance types. Due to many workflows requiring different resources depending on the stage, this could be a limitation. Another limitation of CfnCluster is that since it was developed by AWS, it is vendor specific and users cannot utilize it on other commercial clouds.

2.4.1.3 Alces Flight

Similarly to both CfnCluster and CloudyCluster, Alces Flight is another commercially available resource management tool that allows the creation of HPC environments within the cloud [25]. Alces Flight comes in both a Community and Enterprise edition which each have different capabilities. The Community Edition is available to users at no extra charge while the Enterprise edition costs extra but providers more features like the ability to have multiple users in an HPC environment. We will focus on the Community (Solo) edition of Alces Flight since that will be more appealing to researchers because of the cost. In the Solo edition, similar to CfnCluster, users are provided with a login instance, a shared filesystem, an HPC batch scheduler and a configurable number of compute instances. Also similar to CfnCluster, Alces Flight Solo provides autoscaling of a single instance type depending on the number of jobs in the job queue.

Alces Flight Solo does not provide any special tools for workflow management. It provides access to the HPC scheduler where the user can log in in submit a job script to begin the workflow similar to both CloudyCluster and CfnCluster. Similar to CfnCluster support for heterogeneous resource types within the same environment is not supported which can limit the effectiveness of workflow execution. Alces Flight does support more HPC scheduler options then CloudyCluster (Torque/SLURM) and CfnCluster (Torque/SLURM/SGE) by offering SLURM, SGE, OpenLava, Torque, and PBS Pro.

2.4.1.4 StarCluster

Another solution that provisions resources is StarCluster [61]. StarCluster is similar to CfnCluster as it is a free commandline driven utility that launches HPC environments on AWS. StarCluster was one of the first opensource tools for creating HPC environments in the cloud, unfortunately there have not been any updates to StarCluster since 2016. In the fast moving world of the commercial cloud that is an eternity and it means that StarCluster doesn't support many of the new advanced features that are now offered. StarCluster also only offers one HPC scheduler, Oracle Grid Engine, whereas the other offerings all offer at least two options. Although, StarCluster does support minimal autoscaling of a single instance type based upon the number of jobs in the scheduler queue.

StarCluster additionally does not have any workflow management capabilities outside of job submissions. It also requires that the user launch a "StarCluster Amazon Machine

Image (AMI)" which hasn't been updated in a while. Users are able to add StarCluster capability to other AMIs but this is generally beyond the scope of many researchers abilities and time.

2.4.1.5 CycleCloud

CycleCloud is a tool utilized for creating, managing, operating, and optimizing HPC compute clusters in Microsoft Azure [99]. Utilizing CycleCloud users can dynamically provision HPC environments within Azure and orchestrate data and jobs from within a web based application. CycleCloud allows users to monitor and submit jobs to the environments and automatically scale the resources provisioned based upon job load, availability, and time requirements. It also includes cost auditing functionality and alerts that can warn users if they go over a certain billing threshold. CycleCloud requires an additional server to operate that can be run on premise or within Azure itself. This server runs the web based application and can deploy HPC environments into Azure utilizing templates that can be shared between users. These HPC environments support traditional HPC schedulers such as Grid Engine, SLURM, and HTCondor along with the Redis and Avere filesystems for shared file access across all the compute nodes.

CycleCloud is used to support multiple cloud providers, however the company was recently bought by Microsoft and support for the other cloud platforms was dropped. Now, utilizing CycleCloud requires that researchers be executing their workloads on Azure. While CycleCloud does provision resources, it does not have the ability manage workflows. Users can create the resources and then log in to the cluster in order to submit their jobs/workflows.

2.4.2 Workflow Management Tools

There are also a variety of tools that solely focus on the management and execution of workflows such as Tigres, FireWorks, QDO, Swift, Pegasus, and Cluster Flow. These tools do not provision any types of resources, but instead assume that all of the resources have already been created and are ready for use. There are also quite a few domain specific workflow management tools that focus solely on one research area and are not translatable to other domains.

2.4.2.1 Tigres

Template Interfaces for Agile Parallel Data-Intensive Science (Tigres) is designed with the scientist in mind and utilizes concepts from the User-Centered Design (UCD) process [77]. Tigres utilizes an application programming interface (API) approach to allow users to construct a parallel/distributed scientific workflow in a programming language and have it be able to be executed across multiple platforms [35]. Tigres supports the iterative workflow development cycle of data-intensive workflows and provides a set of templates that can be utilized to compose and execute computational and data pipelines. The currently available templates are: sequence, parallel, split and merge and they can be combined to form a cohesive and efficient scientific workflow. Each template is composed of individual tasks that are the units of work from the end-user that need executed.

Tigres is designed to provide a library that will not mandate users to use a separate stand-alone tool, but instead it allows users to utilize the Tigres library in existing programming languages. Tigres workflows can execute scientific codes that are created in any language which makes it flexible and applicable to a wider variety of researchers. Although it is open-source and freely available for download, it has not been updated since 2016.

However, as with all of the tools in this category, for operation, Tigres assumes that all the computational resources already exist and are ready to use and therefore does not deal with the provisioning or de-provisioning of resources. This works well on traditional HPC resources where the nodes are known ahead of time and ready to go but requires a tool like one of the previously discussed resource provisioning tools to be effective in the cloud. This can deter a cloud seeking user as now they have to learn two separate tools in order to get their workflow running on the cloud.

2.4.2.2 FireWorks

Fireworks is another free and open source workflow management tool that is designed for defining, managing, and executing workflows [37]. Fireworks workflows consist of three main components: FireTasks, Fireworks, and Workflows. A Firetask is defined to be an atomic computing job such as a single shell script or a user defined Python function. A Firework contains that Javascript Object Notation (JSON) spec that includes all of the information required to bootstrap the job. This JSON spec can include a variety of things including the input parameters for the workflow, or an array of Firetasks to execute in sequence. A Workflow is simply a set of FireWorks that have dependencies between them.

An initial FireWorks installation consists of two components: a server (LaunchPad) and one or more workers (FireWorkers). The LaunchPad is the server that manages the workflows, from the LaunchPad users can monitor or rerun their workflows. The FireWorkers can be though of as the compute nodes in a traditional HPC environment as they are the servers that are performing the computaiton. These FireWorkers request Workflows from the LaunchPad and then simply communicate the results back to the LaunchPad when the tasks have ended. FireWorks supports multiple HPC schedulers such as PBS/Torque, Sun Grid Engine (SGE), SLURM, and IBM LoadLeveler which increases the odds that it will be able to understand a researcher's workflow.

One downfall with FireWorks is that due to its client/server nature, it takes a user with a decent amount of system administration experience in order to configure the FireWorks system. It also requires an additional server to run the LaunchPad processes which can increase costs when executing in the cloud. This drastically raises the barrier to entry for most researchers especially considering when executing in the cloud they still have to provision the resources as well.

2.4.2.3 QDO

QDO (kew-doo) is a lightweight high-throughput queuing system for workflows that have many small tasks to perform [6]. QDO was designed by the astrophysics community in order to manage queues of HTC jobs that otherwise may exceed the capacity of the underlying batch systems. QDO can help to combine the many small tasks found in HTC workflows into a single HPC batch job that can help to reduce the number of jobs executing on the system at once. One of the key concepts of QDO is the flexibility. Unlike other systems that have a more rigid structure, QDO allows users to add additional tasks to the queue after the initial workflow has started and even after it has divided up the tasks into larger batch jobs. It also supports the aggregate management of tasks, task dependencies, and priorities.

However, although QDO is flexible and easy to use, it is lacking the constructs that allow for the chaining of the generated batch jobs into a workflow which limits its usefulness for more complex workflows that consist of more then one stage. Also, like the other tools in this section, QDO is designed to integrate with existing resources that have already been provisioned and are ready to execute batch jobs. It does not contain any resource management component so if a researcher wants to execute a QDO workflow in the cloud, they must set up all the resources manually or through another tool.

2.4.2.4 Swift

Swift is a parallel scripting language that is designed for composing application programs into parallel applications that can be executed on parallel resources [104]. It is designed to execute many instances of ordinary application programs concurrently on distributed parallel resources. While Swift does have a limited set of data types, operators, and built-in functions it does not attempt to replicate the functionality of other scripting languages. Swift scripts utilize a C-like syntax and are written as a set of functions. Swift also allows users to express operations on datasets in terms of their logical organization utilizing XML Dataset Typing and Mapping (XDTM). Another advantage of Swift is that it is portable and the same script can execute on multiple types of resources. This makes it an excellent option for users who want to be able to execute their workflows both on premise and on the cloud.

However, while Swift is a powerful and flexible tool for orchestrating workflows in order to utilize it researchers are required to learn an entirely new domain specific programming language. Some of the programming concepts found within may be intimidating for less technical researchers. For technically proficient researchers this may not be a large hurdle, but for most academic researchers who know just enough about how to execute their workloads having to learn a programming language is outside of the scope of their research. Swift also requires that all of the resources exist before execution and does not have a resource provisioning component. This again puts researchers at a disadvantage as they have to learn both a new programming language and a resource provisioning system.

2.4.2.5 Pegasus

Pegasus is another workflow manager that was designed in a way that separates the workflow description from the description of the execution environment [20]. This separation allows workflows to be portable across multiple execution environments while also allowing certain optimizations at both compile time and runtime which can help improve both the reliability and performance of the workflow execution. Workflows within Pegasus are defined based upon the Directed Acyclic Graph (DAG) representation where the tasks to be executed are represented as nodes and the data/control flow dependencies between them are represented as the edges. By utilizing a DAG representation, Pegasus can utilize the wealth of research in graph algorithms to make the compile time and runtime optimizations to the workflow execution. Pegasus workflows are composed via Directed Acyclic graph in XML (DAX) APIs available for higher level programming languages such as Python, Java, and Perl. Pegasus also provides users with the capabilities to track and monitor their workflows automatically as well as the ability to automatically handle workload failures by retrying the failed workload, checkpointing, or retrying the specific failed tasks.

Pegasus does not have any particular resource management features, however it does integrate with HTCondor that can perform certain resource management tasks. HTCondor will be discussed in more detail in a later section. Another possible issue that new researchers face when attempting to utilize Pegasus is that due to the different optimizations that happen at compile time/runtime the DAG that gets executed may not necessarily be the exact same as what was originally specified. Users can spend a great deal of time debugging their workflow to ensure that the DAG is output and executed correctly.

2.4.2.6 Cluster Flow

Cluster Flow is a flexible and simple pipeline tool designed for use in the bioinformatics domain [24]. Cluster Flow is a commandline based utility that is designed to work with an HPC environment. It currently supports the LSF, SLURM, and SGE HPC schedulers and utilizes them in order to submit and execute the bioinformatics pipelines. Cluster Flow was not desgined to allow for advanced features, but instead deliberately restricted the data flow patterns avaliable to the user to keep a simple pipeline syntax. This makes it easier for researchers to get started executing their pipelines and allows them to get their results faster. It comes with a large number of packaged modules and preconfigured pipelines for common bioinformatics tools making setup even easier.

Cluster Flow does not include a resource management component and requires that either an HPC environment is provisioned or that it executes locally on the machine. Cluster Flow was designed to tackle the issue of creating specific bioinformatics pipelines and as such is limited to that specific domain. Cluster Flow is not going to be able to help users execute large machine learning workflows. Our research looks to create a tool that will compliment domain specific tools like Cluster Flow and the other previously mentioned tools in order to make their transition to the cloud easier and seamless for the researchers.

2.4.3 Resource And Workflow Management Tools

There are a few examples of tools that try to manage both resource and workflow management at the same time such as AWS Batch, Galaxy CloudMan, and HTCondor. However, these tools tend be focused on a specific domain of users or are vendor locked into a specific resource provider. This takes some control away from the researcher by ensuring that if they want to move to another platform they have either re-write their workflow or learn another tool.

2.4.3.1 AWS Batch

AWS Batch is an AWS managed service that enables users to execute batch computing workloads within AWS [86]. AWS Batch dynamically provisions the optimal quantity and type of computing resources based on the specific resource requirements of the batch jobs submitted. AWS Batch eliminates the need for having to manage a traditional batch computing software or server clusters as it is all created on demand. It can take advantage of the AWS Spot Market which helps to lessen the overall cost of computing. AWS Batch also provides the ability for users to submit workflows and pipelines as well by enabling users to express interdependencies that exist between the different submitted jobs. The jobs within AWS Batch can be any job that can be executed as a Docker container. AWS Batch utilizes the concept of a *job definition* which can be thought of as similar to a traditional batch job script. It defines the jobs to execute, compute requirements, environmental variables, parameters, and any other information required to execute the jobs.

If a researcher only wants to execute their specific workload on AWS, then AWS Batch would be a potential solution for them. However, if a researcher wanted to take the workflow to another cloud or even back to on premise, they would have to re-write the workflow utilizing a different tool. This may cause issues for grant funded researchers in performing their research, as they may need to move to other platforms depending on the source of their funding. Another possible stumbling point when attempting to utilize AWS Batch is that researchers will need to be able to create their own Docker containers and upload them to AWS. This additional step requires re-installing all the libraries and dependencies of their applications and understanding how running within a Docker container can affect the execution of their jobs. AWS Batch also does not allow users to SSH into their instances to troubleshoot the jobs, instead it provides the logs to users

2.4.3.2 Galaxy CloudMan

Galaxy CloudMan enables Galaxy to be quickly configured and deployed on different cloud resources [3]. CloudMan is heavily customized for use by researchers in the fields of genetics and biology by enabling the dynamic provisioning of Galaxy on cloud resources. Galaxy is a web-based platform that focuses primarily on biomedical research and each CloudMan cluster can be optionally provisioned with the Galaxy software suite. CloudMan currently supports the AWS, OpenStack, and OpenNebula clouds which allows researchers to migrate their workflow between the different providers based upon availability and cost.

If the Galaxy software option is not chosen, then the cluster comes pre-configured with the SLURM HPC scheduler, NFS storage, and interactive access. However, when operating in this mode CloudMan does not support any workflow management as the Galaxy software is what is utilized for the workflow management portion. So while CloudMan works great for Galaxy based workflows, it is not really the generalized solution that most researchers are looking for as it is mainly targeted at biology and genetic researchers.

2.4.3.3 elasticHPC

Another similar tool is elasticHPC which is designed for use in the bioinformatics domain [23]. elasticHPC allows users to provision traditional looking HPC environments through a configuration file and a commandline driven interface. elasticHPC also supports the three major cloud providers: AWS, Microsoft Azure, and GCP which allows users to chose the best cloud provider for their needs and allowing users to transfer their solutions from one cloud to another. elasticHPC takes advantage of different cost saving techniques on the different cloud providers such as the Spot Market on AWS and “Sustained Usage” discounts in GCP which helps to minimize the overall costs of the workflow execution. elasticHPC also comes pre-packaged with a number of bioinformatics packages and tools already installed to help researchers get up and running quickly.

elasticHPC has very minimal workflow management system in that it allows users to submit and monitor jobs through the same interface that was utilized to launch and create the HPC environments. While this is similar functionality to the tools that just create resources, a user does not have to log in to the cluster in order to submit jobs so this submission can be scripted more like a workflow management system. Unfortunately the images utilized by elasticHPC and their documentation have not been updated since 2014 which is an eternity in the cloud computing world so many of the features are no longer functional.

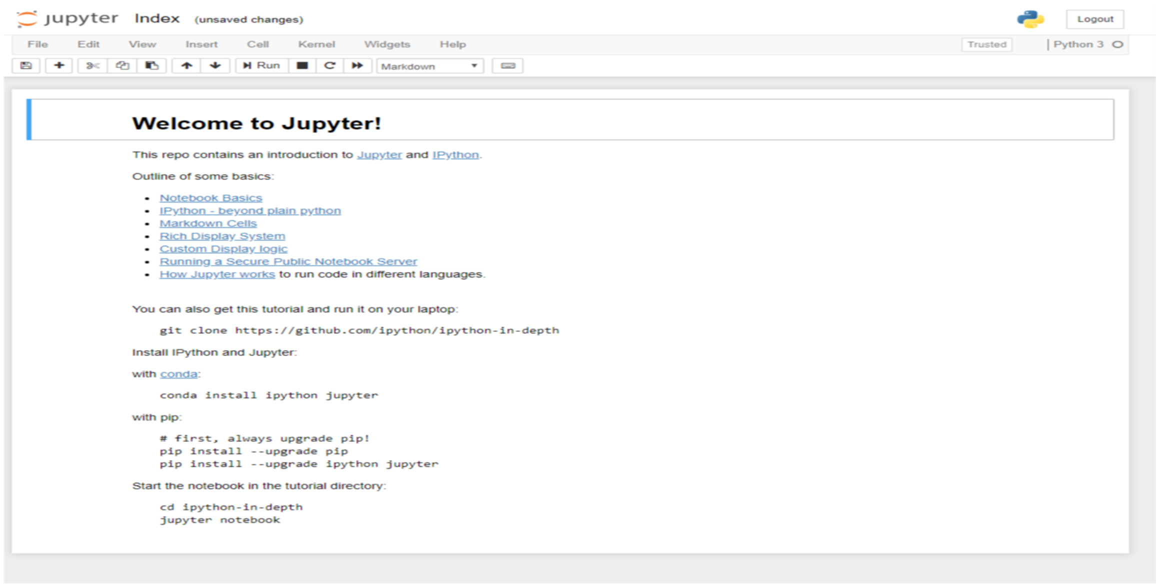
2.4.3.4 HTCondor

HTCondor is a software system that creates a high-throughput computing environment by using the power of computers/workstations that communicate over a network [96]. HTCondor, like other full-featured batch systems, features job queuing mechanisms, resource monitoring, and resource management among other things. HTCondor provides a unique framework that allows jobs to execute on workstations that are idle along with being able to execute on more traditional HPC environments as well. Although HTCondor is more similar to a batch scheduler then a workflow or resource management tool, we include it in our list of tools because it does have multiple annexes that allow for creating and utilizing resources in different commercial clouds.

While these annexes help increase the appeal to more researchers, they are not included with the default HTCondor installation and take some additional configuration to enable. Also on top of that researchers have to be able to manage and tune HTCondor in order to obtain the best performance possible. Typically HTCondor is configured and administered by experienced system administrators not researchers which makes the learning curve to start utilizing HTCondor steeper then some of our other tools. Also there are certain limitations that are imposed on HTCondor jobs. These limitations include that workflows cannot do interactive input and output, the jobs must not be multi-process, and the jobs are not able to have interprocess communication. These limitations also rule out HTCondor as a generalized tool for multiple types of workflows as some of those limitations are critical to certain workload types.

2.4.4 Jupyter

Project Jupyter is a non-profit, open-source project that was born out of the IPython Project in 2014 as it evolved to support interactive data science and scientific computing across all programming languages [38]. Jupyter is becoming increasingly popular in the academic research community to do its easy to use web-based interface and the ability to easily save and share the Jupyter Notebooks with other researchers. Jupyter Notebook is an open-source web based application that allows users to create and share documents that can contain equations, visualizations, and even live code. A sample of a Jupyter Notebook is shown in Fig. 2.1. These Jupyter Notebooks are designed to be utilized by a single-user, while JupyterHub [39] is another open-source project that allows for the hosting of multiple Jupyter Notebook servers which can be utilized by a group of users.



Welcome to Jupyter!

This repo contains an introduction to Jupyter and IPython.

Outline of some basics:

- Notebook Basics
- IPython - beyond plain python
- Markdown Cells
- Rich Display System
- Custom Display logic
- Running a Secure Public Notebook Server
- How Jupyter works to run code in different languages.

You can also get this tutorial and run it on your laptop:

```
git clone https://github.com/ipython/ipython-in-depth
```

Install IPython and Jupyter:

with conda:

```
conda install ipython jupyter
```

with pip:

```
# first, always upgrade pip!
pip install --upgrade pip
pip install --upgrade ipython jupyter
```

Start the notebook in the tutorial directory:

```
cd ipython-in-depth
jupyter notebook
```

Figure 2.1: Example of a Jupyter Notebook running in a web browser [38].

Due to their interactive nature and the availability of browser based access, Jupyter Notebooks can be a valuable tool for introducing new researchers to scientific computing. There have been a number of Universities that have already made some form of Jupyter Notebooks available to their researchers by integrating it in various ways, these include but are not limited to: University of California Berkely, Clemson University, MIT and Lincoln Labs, Michigan State University, University of Minnesota, Texas Advanced Computing Center (TACC)/University of Texas, and the University of Illinois [39]. Along with these academic and research institutions, three of the major commercial cloud providers also provide guides on how to configure JupyterHub on their resources [86, 82, 81]. However, where the academic institutions have the resources and interface pre-configured for the researchers, all the commercial cloud providers require the user to launch resources and perform some of the configuration manually which increases the learning curve for researchers.

The integration of Jupyter have varied quite a bit across the deployments. At the University of Minnesota, in order to address the issue of getting Jupyter Notebooks and JupyterHub executing on traditional HPC resources they developed a tool that would handle the integration with batch job scheduling, control of job profiles, and a central authentication service [60]. Through the use of this tool, the University was able to give researchers access to their own Jupyter Notebooks that utilize the existing HPC resources already available on campus. This tool was called *BatchSpawner* and by plugging into JupyterHub it allowed a traditional HPC batch scheduling system to launch user's notebook servers. This is accomplished through the use of job script templating for various HPC schedulers including Torque, SLURM, Condor, Moab, and Grid Engine style schedulers. Since its initial release *BatchSpawner* has been adopted by the JupyterHub developers and is now officially a supported component of JupyterHub.

There are also a few Jupyter deployments that are mainly focused on a particular application/science domain. One of these implementations is by the Ohio Supercomputing Center which developed a interactive HPC web application on an Open OnDemand deployment that can be used to launch and connect to Jupyter notebooks [66]. This application is geared more towards the deployment and utilization of Apache Spark clusters for data processing. When a user requests an instance of the Jupyter with Spark Interactive App, the App submits a job on the pre-configured OnDemand deployment that launches a Spark cluster and Jupyter Notebook server. This allows researchers to have quick and dynamic access to the resources they need as well as the simplicity and Graphical User Interface (GUI) that Jupyter Notebooks provide. This minimizes the learning curve for new researchers as they are still able to access all the resources but are not required to know all the details.

Another domain specific implementation is CyberGIS-Jupyter (geographic information science and systems (GIS) based on advanced cyberinfrastructure) which is an innovative cyberGIS framework for achieving data-intensive, reproducible, and scalable geospatial analytics utilizing Jupyter Notebooks [106]. These Jupyter Notebooks can then interact with ROGER which is the first cyberGIS supercomputer for the computationally intensive tasks. The framework consists of four different stages and is shown in Fig. 2.2. As shown in Fig. 2.2, the first stage is where the users access the JupyterHub installation to request a Jupyter Notebook. The second stage is where the Jupyter Notebook is launched via Docker Swarm to a container running on the OpenStack cloud. Once the user has the Jupyter Notebook session running, they can then utilize specialized widgets within the Jupyter Notebooks to submit jobs to the ROGER HPC cluster. Similar to the other deployments mentioned, this also depends on the HPC environment already being created and ready to execute. It does support dynamic creation of more Jupyter Notebooks via adding additional instances to the OpenStack deployment but the HPC backend and resources available for batch processing are largely static and pre-configured.

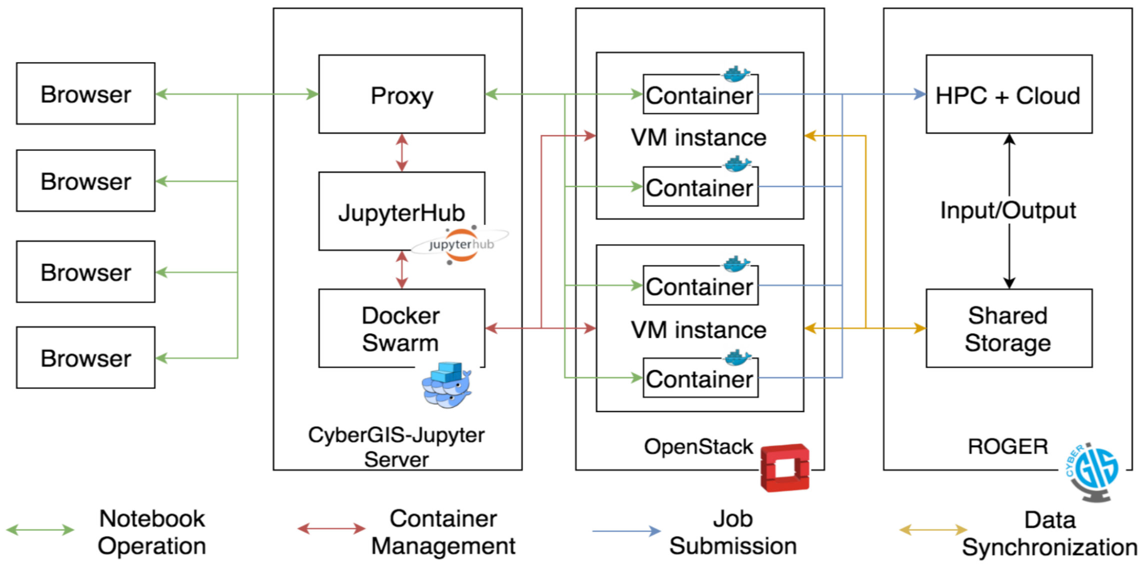
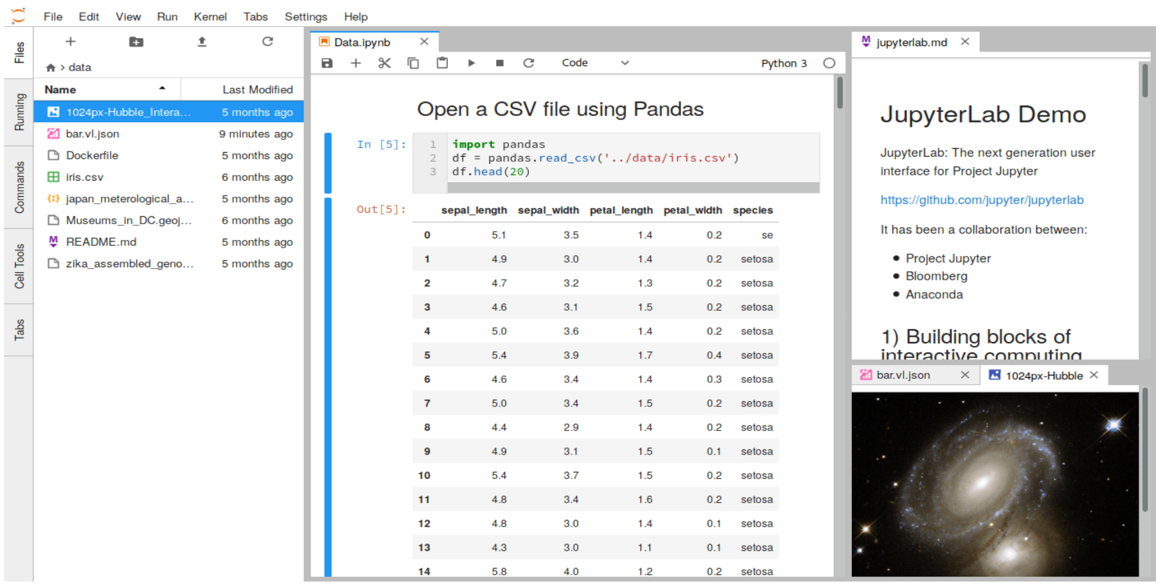


Figure 2.2: Architecture of the cyberGIS implementation of Jupyter. [106].

While the classic Jupyter Notebook interface is powerful and user friendly, there is a new cleaner and more powerful user interface that is being developed called JupyterLab. JupyterLab allows users to work with documents and activities in a flexible, integrated, and extensible manner [40]. JupyterLab includes the ability to work on multiple documents and activities side by side through the use of tabs and splitters where the original interface only allowed for one task at a time. These activities and documents can include things such as Jupyter Notebooks, text editors, terminals, and other custom components enabling the user to multitask more effectively. This new interface is shown in Fig. 2.3.



Name	Last Modified
1024px-Hubble_Intera...	5 months ago
bar.vl.json	9 minutes ago
Dockerfile	5 months ago
iris.csv	6 months ago
japan_meterological_a...	5 months ago
Museums_in_DC.geoj...	6 months ago
README.md	5 months ago
zika_assembled_geno...	5 months ago

Open a CSV file using Pandas

```
In [5]:
import pandas
df = pandas.read_csv('../data/iris.csv')
df.head(20)
```

Out[5]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	se
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa
5	5.4	3.9	1.7	0.4	setosa
6	4.6	3.4	1.4	0.3	setosa
7	5.0	3.4	1.5	0.2	setosa
8	4.4	2.9	1.4	0.2	setosa
9	4.9	3.1	1.5	0.1	setosa
10	5.4	3.7	1.5	0.2	setosa
11	4.8	3.4	1.6	0.2	setosa
12	4.8	3.0	1.4	0.1	setosa
13	4.3	3.0	1.1	0.1	setosa
14	5.8	4.0	1.2	0.2	setosa

JupyterLab Demo

JupyterLab: The next generation user interface for Project Jupyter

https://github.com/jupyter/jupyterlab

It has been a collaboration between:

- Project Jupyter
- Bloomberg
- Anaconda

1) Building blocks of interactive computing

Figure 2.3: The new JupyterLab user interface currently in development. Eventually JupyterLab will replace the classic Jupyter Notebook [40].

As shown in Fig. 2.3, the new JupyterLab interface looks similar to a traditional desktop interface where a user can have multiple windows or tabs open at a time. This will make new Jupyter users feel more comfortable with the layout and allow experienced users to multi-task more efficiently and effectively. While JupyterLab will eventually replace the classic Jupyter Notebook, JupyterLab is still under active, although stable, development so the transition is not yet complete.

2.5 Benchmarks

In this work, we utilize a number of different benchmarking suites in order to quantify and validate the performance of the HPC environments that we are creating in the different commercial cloud environments. This section provides an overview of the different benchmark suites that we utilize and an overview of the tools included in each one.

2.5.1 HPC Challenge Benchmark Suite

The first benchmark suite that we will be utilizing is the HPC Challenge (HPCC) benchmark suite developed at the University of Tennessee [49]. The HPCC suite is designed to measure a range of memory access patterns and is an open source project. HPCC consists of basically seven tests: HPL, DGEMM, STREAM, PTRANS, RandomAccess, FFT, and the Communication bandwidth and latency tests. In this section we will give a brief background on each of these tests and explain what each one is testing.

2.5.1.1 HPL

HPL or the High Performance Linpack benchmark is software that solves a (random) dense linear matrix in 64-bits double precision arithmetic on distributed computers [70]. This package quantifies accuracy and the time taken to complete the solution utilizing an included testing and timing program. HPL solves a system of linear equations of order n:

$$Ax = b; \quad A \in \mathbf{R}^{n\times n}; \quad x, b \in R^n$$

by first computing LU factorization with partial row pivoting of the n by $n+1$ coefficient matrix

$$P\big[A, b\big] = \big[[L, U], y\big]$$

Then the solution can be obtained in one step solving the upper triangular system

$$Ux = y$$

[50]. HPL is utilized within the HPCC benchmark suite to evaluate the floating point rate of execution for solving a linear system of equations. This is a "global" challenge where all the processors are working in coordination in order to solve a singular problem.

2.5.1.2 DGEMM

Double-precision general matrix multiplication (DGEMM) is another measure of the floating point arithmetic performance of an HPC environment. Within the HPCC benchmark suite, DGEMM is a level three BLAS routine that performs one of the matrix-matrix operations

$$C \leftarrow \alpha AB + \beta C$$

where

$$A, B, C \in \mathbf{R}^{n \times n}; \quad \alpha, \beta \in \mathbf{R}^{n}$$

In this case alpha and beta are scalars while A, B, and C are matrices. A is an m by k matrix B is a k by n matrix and C is a m by n matrix and A and B can optionally be transposed [22, 50].

DGEMM is utilized along with HPL to measure the floating point performance for the HPC environment though matrix-matrix multiplication and is executed in multiple configurations. The first configuration is the “single” configuration where DGEMM is executed on a randomly chosen processor on the system while the second configuration is the “star” configuration where a copy of DGEMM is executed across all processors concurrently. executing DGEMM in multiple configurations allows for a more complete view of system performance and can showcase any issues with contention among resources when executing concurrently.

2.5.1.3 STREAM

The STREAM benchmark is a synthetic benchmark program that measures the sustainable memory bandwidth in GB/s along with the corresponding computation rate for simple vector kernels [56]. The simple vector kernels utilized by STREAM are:

$$\begin{aligned} COPY:&\quad c \leftarrow a \\ SCALE:&\quad b \leftarrow \alpha c \\ ADD:&\quad c \leftarrow a + b \\ TRIAD:&\quad a \leftarrow b + \alpha c \end{aligned}$$

where:

$$a, b, c \in \mathbf{R}^m; \quad \alpha \in \mathbf{R}$$

[50]

STREAM is designed to work with datasets that are much larger then the cache available on any given system. This helps to provide results that more closely align with the performance of very large vector style applications. STREAM is again ran in both the "single" and "star" configurations in order to show the differences between executing on a single processor versus executing on all the processors concurrently.

2.5.1.4 PTRANS

PTRANS or parallel matrix transpose is a benchmark that measures the performance of transposing a large array. By doing this, PTRANS is exercising the communications where pairs of processes have to communicate with each other simultaneously [69]. This benchmark sets a random n by n matrix to a sum of its transpose with another random matrix:

$$A \leftarrow A^T + B$$

where:

$$A, B \in \mathbf{R}^{nxn}$$

while the data transfer rate, which is measured in Gbytes/s, is calculated by dividing the size of n^2 matrix entries by the time it took to perform the transpose [50]. This is a "global" experiment which means that all of the processes on the system are working towards solving a single problem which is what leads to the large amount of messages exchanged as each processor needs to exchange information with their neighboring processes in order to transpose the entire matrix. This is a good way to test the overall communication capacity of the network which is one area where commercial clouds have struggled in the past. This benchmark will be an area of focus for our experiments discussed later.

2.5.1.5 RandomAccess

The RandomAccess benchmark measures the rate of integer random updates from memory. This is measured in Giga updates per second (GUPS) [49]. The operation being performed on an integer array of size m is:

$$x \leftarrow f(x)$$

$$f : x \mapsto (x \otimes a_i); \quad a_i - pseudo - randomsequence$$

where:

$$f(\mathbf{Z}^m) \rightarrow \mathbf{Z}^m; \quad x \in \mathbf{Z}^m$$

[50]. The RandomAccess benchmarks focus on random memory access because as random memory access becomes more expensive relative to processor operations there needs to be a way to measure performance based upon the random memory access performance. Random memory performance has been found to often be a good indicator of application performance as even a small percentage of random memory access within an application can drastically affect the overall performance of the application itself [45]. GUPS is calculated by identifying the number of memory locations that can be randomly updated in one second and dividing it by one billion. Within the HPCC benchmark suite, the RandomAccess benchmarks are

executed in all three different configurations: “single”, “star”, and “global” in order to test both the overall system performance and to test the subcomponents performance as well.

2.5.1.6 FFT

FFT or Fast Fouier Transform is a benchmark that is utilized by HPCC to measure the floating point rate of execution of double precision complex one-dimensional Discrete Fourier Transform (DFT) [49]. This one-dimensional Discrete Fourier Transform (DFT) is of size m:

$$Z_k \leftarrow \sum_{j}^{m} z_j e^{-2\pi i \frac{jk}{m}}; \quad 1 \leq k \leq m$$

where:

$$z, Z \in \mathbf{C}^m$$

[50]. Where the operation count is taken to be $5mlog_2m$ for the calculation of the computational rate in Gflops/s. The FFT benchmark is also run in the three configurations: “single”, “star”, and “global” which allows the observations about how much the network is impacting the results to be seen in the results from each configuration. Similar to PTRANS, the FFT benchmark also performed poorly in our testing due in large part to the amount of communication required between processes. This will be discussed in more detail later on.

2.5.1.7 Communication Bandwidth and Latency

The Communication Bandwidth and Latency benchmark is an MPI centric set of tests to measure both latency and bandwidth of a number of simultaneous communication patterns. These patterns are based on the b_eff, effective bandwidth benchmark, however they are slightly different as the operation count is linearly dependant on the number of processors in the tested system [50]. The major results that are reported by this benchmark are: maximal ping pong latency, average latency of parallel communication in randomly ordered rings, minimal ping pong bandwidth, bandwidth per process in the naturally ordered ring, and average bandwidth per process in randomly ordered rings. This information is very useful when attempting to determine the network characteristics of the system which can help determine which applications are best suited for that particular system.

2.5.2 NASA Parallel Benchmark Suite

Another benchmark suite that we will be utilizing for this work is the NASA Advanced Supercomputing (NAS) Parallel Benchmarks (NPB). The NPB are derived from computational fluid dynamics (CFD) applications and consist of a number of benchmarks that can be utilized to test the performance of HPC environments [5]. The suite consists of twelve benchmarks and seven different problem classes that test different components of the HPC environment. These benchmarks include the original eight benchmarks consisting of five kernels: Integer Sort (IS) that utilizes random memory accesses, Embarrassingly Parallel (EP), Conjugate Gradient (CG) that utilizes irregular memory access and communication, Multi-Grid on a sequence of meshes (MG) a memory intensive application that utilizes long and short distance communication, and discrete 3D fast Fourier Transform (FT) that utilizes all-to-all communication. The original eight benchmarks also include three pseudo applications: Block Tri-diagonal solver (BT), Scalar Penta-diagonal solver (SP), and Lower-Upper Gauss-Seidel solver (LU). Recently there have been four new benchmarks added for unstructured computation, parallel I/O, and data movement: Unstructured Adaptive mesh (UA), Block Tri-diagonal solver I/O (BT-IO), Data Cube (DC), and Data Traffic (DT).

The NPB also come with a range of different benchmark classes that can be utilized for testing. These test classes are the small starter Class S that can be utilized for small quick tests, Class W for a workstation sized test, Classes A, B, and C which are the standard test problems that have roughly a $\tilde{4}$x increase when going from one class to the next, and lastly Classes D, E, and F which are the large test problems and have roughly 16x increase when going from class to class [5].

2.6 Determining The Optimal Configuration For Parallel Computation

Within an on premise or local HPC environment, there are a finite number of resources and resource combinations that can be utilized for the execution of parallel computation. The resources available in these environments are chosen and configured for the researchers by the system administrators who have extensive knowledge in how all the components fit together and which configurations will yield the desired performance level. Due to these constraints, the researcher has a limited number of configurations of resources that they can utilize to execute their application. Although the resources are limited in scope, generally most of the configurations available to the researchers will yield decent performance due to optimizations made by the system administrators. Also, due to the resources already being provisioned, trying different resource configurations and combinations to determine what performs the best is mainly limited by the availability of the resources and the time of the researcher so if a researcher does not choose the proper configuration on the first try it is not a large issue. However, when executing in the commercial cloud determining the correct configuration and combination of resources to utilize on the first try is much more important and more difficult.

There are many different resource options and configurations to consider when deploying the resources to execute a particular HPC application. Each commercial cloud vendor has multiple types or resources with differing hardware such as vCPUs, RAM, GPUs, Network, and Disks and these different types of resources can greatly effect the execution of the HPC applications. In contrast to local resources that have a finite number of resources that researchers can choose from, the commercial cloud has an almost infinite amount of possible resource combinations for researchers to choose from. This can lead to uncertainty for researchers when attempting to deploy their scientific workflows on the commercial cloud as they may not have all the information required to make an informed decision on which resource configuration best fits their scientific workflow's requirements. Also unlike local resources that are already provisioned, in order to test multiple configurations and combinations, a researcher must invest time and money into provisioning the different combinations of resources in order to find the proper configuration.

Also, just looking at the application execution times on the commercial cloud can sometimes miss the point. There are many more variables that must be considered when executing in the commercial cloud. While on premise many of the available configurations are highly optimized for scientific workflows, the commercial cloud is more general purpose and the performance tricks and tips that work on local resources might not increase performance as a researcher would traditionally expect. This can lead to researchers dramatically over or under provisioning commercial cloud resources which can negatively affect the performance while even potentially increasing the cost. As our previous research has shown, there are cases where adding more instances to the application can actually negatively affect the performance of the job to the point where it is unusable [73]. Since this is different than the traditional way of thinking, researchers may not think to try smaller configurations of instances when evaluating different configurations.

There have been a number of studies that have looked at estimating the execution time of different scientific applications in general as well as predicting their performance on the commercial clouds such as [64, 54, 19, 105, 53, 31]. These studies have looked at how researchers can profile or utilize existing profiles of applications to determine a rough estimate of the execution time for their application. While both [105, 53] both assume that the target architecture being studied is dedicated to the execution of the specific application being studied. While this holds for a traditional HPC environment, this is not true for the commercial cloud as it has a more general purpose architecture that is designed to execute many different workloads and workflows.

In [64], the goal is to predict the runtime for each application right after it has been submitted to the cloud. This does not help to solve the issue of what configuration to utilize as the configuration would have already be decided. Studies such as [19, 31] attempt to address the optimal resource configuration for different types of workloads but they still hold on to the assumption that the target application has already been profiled on the target system which is an additional step for the researcher. This additional step also incurs costs and takes time that a researcher may not have which could discourage them from trying the commercial cloud.

Only [53, 54] analyze different workloads with varying configuration and architectural parameters. However, only [54] addresses the need to be able to predict application execution time without having to deploy the entire configuration first. This study was tested on the OpenStack research cloud with the thought to utilize it in other clouds as well. The study focuses mainly on CPU intensive applications and relies on the cloud provider generating the profiles for the hardware configurations in the cloud. However, getting large cloud providers to generate the required profiles may prove difficult especially at the scale required. Also the study only looks at the number of nodes, CPUs per node, and RAM per node. It does not take into consideration network performance as it is fixed on the test cloud nor do they look at the storage system as their benchmarks are not sensitive to its performance. Within the commercial cloud such as AWS, performance can vary heavily depending on a large variety of factors including placement of instances, utilization of instances, time of day, instance class, etc. All of these factors need to be included in the calculation as well as the number of instances, CPUs per instance, and RAM per instance are only a small part of the configuration in the cloud.

2.7 Urgent Computing

In this section we discuss the current research that has been performed in the area of urgent computing. We will discuss three different categories of research that have been performed in this field. These categories include research focused on building infrastructure and support systems, on the generation of damage prediction simulations, and research on creating a more generalized definition of urgent computing.

2.7.1 Building Infrastructure and Support Systems

The area of building infrastructure and support systems for urgent computing focuses on how best to take advantage of existing computational resources for urgent computing. Generally this is accomplished through some type of priority system in order to allow for urgent computing tasks to be completed in a timely manner. One such system that was designed to help with the urgent computing issue is the Special Priority and Urgent Computing Environment (SPRUCE) [11]. SPRUCE utilizes a novel token-based authorization system that can be used to facilitate and track urgent computation sessions by approved users. SPRUCE is designed to work closely with existing resource providers and supercomputing centers to allow each resource provider to have full control regarding the policies around which resources and parts of their system can be utilized for urgent computation. The way that SPRUCE works is that a user or group of users is given a special generated "token" that gives the jobs that they submit to the system special "priority". This means that their jobs will execute first and in a more timely manner than a standard system user. By submitting at a higher priority than the standard users, SPRUCE users can also preempt or cancel another standard users' jobs. However, these preemption and priority settings vary as SPRUCE does not provide standard rules. Instead these priority and preemption settings are defined by the resource providers. This can lead to inconsistencies between SPRUCE sites and could lead to some confusion amongst researchers attempting to submit their urgent computing jobs as the processing delay can vary widely between different SPRUCE sites.

Within the SPRUCE paper, the authors do make a mention of the feasibility of possibly utilizing commercial cloud resources as part of SPRUCE. However, the majority of the paper is targeted mainly at existing supercomputing facilities that have dedicated and pre-configured resources. The use of commercial cloud is also discussed in [47]. This work discusses the tradeoffs between utilizing multiple types of e-infrastructure to perform urgent HPC tasks. The types of infrastructure discussed cover both existing HPC, Grid, and Cloud resources. This paper discusses how usually most researchers with an urgent computation workflow and need typically do not have the funding for always-on and dedicated resources. The commercial cloud is presented as a potential solution to this problem depending upon the frequency of execution and the type of urgent computational workload.

2.7.2 Damage Simulation Generation

Another major area of research in the area of urgent computing is the generation of simulations that can be utilized to predict when and where an impending event will happen and how much damage it could cause. These events can be hurricanes, forest fires, tornadoes, tsunamis, or even man-made disasters such as a chemical spill. Some examples of these types of studies that focused on simulating different flooding events in different areas include [63, 8] and [15]. In [63] the authors utilize an existing supercomputing resource, SX-ACE, which is located at Tohoku University. The goal of this case-study was to provide up-to-date information about impending Tsunamis within 20 minutes of the latest earthquake. This would allow citizens within the affected zones time to evacuate and prepare for the event which could help prevent loss of property and lives. In order to gain the priority needed to run this simulation within the 20 minute window, the researchers utilized and modified the job management system of the environment, NQS II, to support prioritizing urgent computing tasks. This involved ensuring that all other running jobs and computations were suspended upon the submission of the tsunami prediction code. This allows the tsunami prediction code to meet the time constraints and provide more instantaneous results. After the tsunami prediction code has executed, all the other previously executing jobs and computations are resumed.

In [8], the authors create a system for monitoring the levees in the Malopolska region of Poland over a certain period of time. The authors propose a hybrid solution that can utilize both commercial cloud resources as well as local resources to perform the computation. The reasoning for this is that the commercial cloud provides reliability and resiliency in the case that local resources are taken offline by the impending event. Also the execution of these types of workloads can be characterized as "spikey" and the computation is not continuously needed. Hence the commercial cloud can offer on-demand infrastructure that can be created and destroyed when needed. The workflow itself utilizes the HyperFlow workflow management systems [7] which executes the specified jobs and returns the results back to the users through a graphical user interface (GUI).

Another study [15] describes a workflow that is implemented to help automate the process of lowring the flood gates in the Saint-Petersburg Barrier. The paper describes how the workflow is implemented within the context of urgent computing. There are a large number of steps within the workflow that the urgent computing paradigm can help increase the efficiency of and provide additional computational power when needed. By enabling the key decision makers to have more up-to-date information in their hands, they can make more informed decisions which can help to limit the loss of both property and lives. This also allows for the minimization of the period of time that the gates remained closed which helps to regulate the flow of water and keep things moving normally and smoothly.

There is another group of simulation studies that have been done regarding simulating traffic [16, 2], which deal with the simulation of evacuations ahead of an impending event. In [16], the authors discuss an implementation of a generic incident model to show the impact that traffic incidents have on evacuation times at large scales. The study utilizes the Real-Time Evacuation Planning Model (RtePM) to model two different scenarios: a terrorist attack in Washington D.C and a hurricane at Virginia Beach. By utilizing these types of simulations ahead of and leading up to an impending event, key decision makers can be more informed and prepared when it comes time to issue an evacuation order. Although these types of simulations are useful on their own, when combined with additional information such as flow rate, population, and information from previous incidents they can become even more useful and informative. In [2], the authors discuss the implementation of a generic traffic management framework for solving large-scale constraint optimization problems. In the paper the system is discussed in regards to both emergency evacuation and congestion pricing. For the implementation of this system, an HPC cluster located at the University of Toronto was utilized to enable the parallelization of the two usecases.

This parallelization allows for the execution of both types of simulations concurrently allowing key decision makers the ability to have more information in front of them. This same methodology can also be utilized for general traffic understanding outside of emergency situations such as for large cities or regions over multiple day, large scale traffic studies.

2.7.3 Creating Generalized Definition of Urgent Computing

The last portion of research that we discuss in terms of urgent computing is the work performed on creating a generalized definition of urgent computing. This research brings together a number of different topics into a widely accepted and generalized definition of urgent computing, as previously there were a large number of variations on the definition of urgent computing. In [48] the authors explore the related urgent computing paradigms and provide a comprehensive general version of the urgent computing definition and works to clarify the differences between the different versions. The work defines an updated definition that helps to clarify common terms, requirements, pre- and post-computation characteristics, deadlines, and costs.

Chapter 3

Evaluation of High Performance Computing Workloads In The Commercial Cloud

Our original work begins with an effort to design and architect a dynamic HPC environment that researchers with little cloud knowledge could launch within Amazon Web Services (AWS). During the course of this work, we find that although some applications are not well suited to be executed on AWS, there are some that performed just as well in the cloud as they did on premise.

3.1 Introduction

In the last few years the use of commercial clouds, such as Amazon Web Services (AWS) and Google Cloud Platform (GCP), in industry has grown very quickly. However, the use of these commercial clouds for high performance computing (HPC) by academic researchers and institutions has not grown as quickly. One reason for this is that academic researchers have a certain process that they utilize for their computational needs and tend not to change that process unless they absolutely have to. Many researchers simply do not have the time to take away from their research to learn a new technology such as the cloud.

In order to help facilitate the adoption of cloud, researchers need a simple and effective way to access an HPC environment that looks and behaves similarly to a traditional on premise HPC environment. By giving researchers an HPC environment on the cloud that closely resembles the traditional HPC environment they are currently utilizing, researchers will be more open to utilizing cloud resources because they will not have to completely change their workflows/workloads.

Another reason for this lack of adoption is due to the performance differences between on-premise and cloud resources when executing HPC applications. While on premise HPC resources are generally engineered to utilize low latency networks, such as Infiniband, and execute HPC applications, the cloud environment is a more generalized environment and tends to use more traditional Ethernet based networks. However, although the cloud may not have the same network type as on premise, the cloud does typically allow access to a more diverse range of compute resources that may not be available to the user on premise. Examples of some of this diverse hardware includes access to the latest Graphical Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs), large memory machines, and access to newer chipsets and processors. Theoretically this means that when utilizing certain HPC applications, researchers can decrease their application run time and produce more results in the same period of time.

One of the main challenges that comes with this diverse range of hardware is determining which configurations of hardware will yield the best results with certain types of HPC applications. We aim to identify types of HPC applications that perform well on the cloud as well as determining the ones that do not perform as well. For the applications that do not perform well, we will investigate different hardware configurations and try to determine if there is any type of hardware configuration in the cloud that will perform well.

We describe and report on our initial HPC environment setup to enable more efficient access to researchers, identify which HPC applications perform well on the cloud, the configuration experiments on poor performing applications, and finally identify the goals of future research in this area.

3.2 Methodology

Academic researchers are accustom to utilizing a traditional on premise HPC environment that typically consists of a Login node, HPC Scheduler (Torque/SLURM/etc), a shared filesystem, and compute nodes. This HPC environment is usually maintained and configured by the University and system administrators. However, on the cloud the researcher becomes responsible for deploying, managing, and maintaining this infrastructure. In order to get comparable benchmarks, an infrastructure for deploying these HPC environments on the cloud was required because no matter what the performance comparisons show, researchers will not want to have to become a system administrator to execute on the cloud. This solution is developed utilizing Python, Botocore, and Javascript in order to create a web based application that allows researchers to launch these HPC environments through a simple point and click wizard [73].

After developing this tool, we are able to perform the benchmarks on both the cloud and on premise resources. For the benchmarks we utilize the HPC Challenge benchmark suite [49]. The HPC Challenge benchmark suite is developed out of the University of Tennessee and consists of a set of seven different benchmarks that test various aspects of an HPC environment. The seven included benchmarks are: HPL, DGEMM, STREAM, PTRANS, RandomAccess, FFT, and Communication bandwidth and latency. These benchmarks test the network, memory, and computational performance all from the same benchmark suite.

We conduct these HPCC experiments both on a generated HPC environment in AWS and an on premise HPC cluster. The on premise cluster consisted of 8 nodes that each had a 2.0Ghz Intel Xeon E5-2660v2 CPU and 128 GB of RAM. The generated HPC environment in AWS also consisted of 8 nodes of the r3.4xlarge AWS instance type that each contained a 2.5 Ghz Intel Xeon E5-2670 CPU and 122 GB of RAM. This is as close as we could get for comparison purposes due to AWS having rigid instance types so we cannot change the amount of RAM or the CPU used by the instance type. Both clusters ran CentOS 7, MPICH 3.0.4, BLAS, and Atlas SSE3 libraries. We also have dedicated access to the Torque/Maui scheduler queue so we do not have any contention from other jobs in the system.

After conducting the HPCC experiments we further analyze additional AWS configurations for the MPIFFT benchmark as it was the worst performing application in the HPCC benchmark suite. For these experiments we utilize multiple AWS instance types and options in order to find a configuration that would increase the performance. These experiments are executed on m4.10xlarge, c4.8xlarge, and c3.8xlarge instance types using both dedicated and non-dedicated options.

When executing the benchmarks in AWS we utilize Clustered Placement Groups which provide a way to control placement of instances to help minimize the latency between instances [86]. We also utilize Enhanced Networking which utilizes single root I/O virtualization (SR-IOV) in order to provide higher performance (in packets per second), lower latency, and lower jitter [86].

3.3 Results

To perform the experiments on each environment, we run the full HPCC benchmark suite ten times and then take the average of the results. After obtaining these results, we focus in on the MPIFFT benchmark to determine if there is any configuration on AWS that allows us to match the performance that we were seeing on the on premise cluster.

3.3.1 HPCC Benchmark Results

We first evaluate the execution of the full HPCC benchmark suite on both clusters. Fig. 3.1 illustrates the results. We find that as initially expected, most of the results show that the local cluster did perform better. As illustrated by the Randomly Ordered Ring Latency benchmark, much of this performance difference comes from the difference in network architectures and lower latency of the on premise clusters. In general we notice that many of the benchmarks that rely heavily on message passing such as and network communication such as MPIFFT and PTRANS perform poorly. While the StarSTREAM and HPL Calculated Teraflops benchmarks that do not rely as heavily on the network perform well in the cloud.

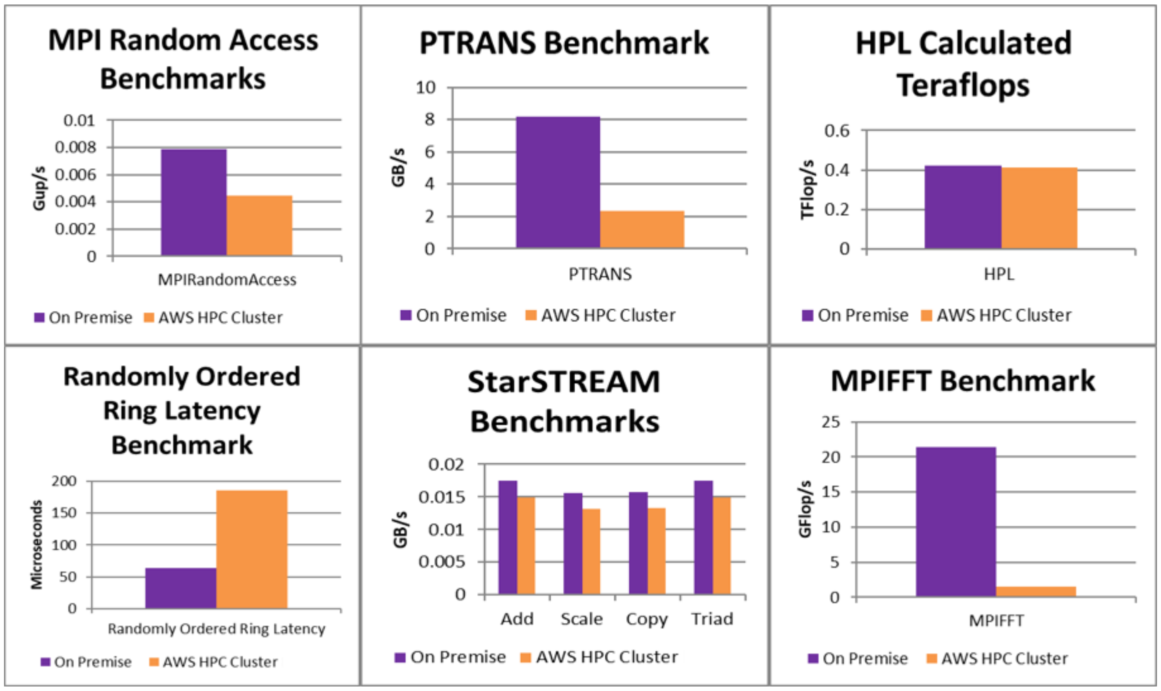


Figure 3.1: Results of individual HPCC benchmarks performed on an AWS HPC cluster and an on premise cluster.

The HPL benchmark is the best performing benchmark showing only a 2.1% decrease in performance between the two clusters. This is in stark contrast to the MPIFFT benchmark that shows a 20 GFlops/s decrease from the on premise environment. These results confirm our hypothesis about there being certain HPC application types that perform well in the cloud while there are other types that do not perform well. As well as confirming our initial hypothesis, the results also posed another question: Is there any AWS configuration that would yield better performance for the MPIFFT benchmark?

3.3.2 MPIFFT Configuration Testing

Due to the MPIFFT benchmark's poor performance in the initial experiments, we want to try and narrow down what caused the decrease in performance and determine if there is a configuration of instances on AWS that would yield better performance. For these experiments we try a number of different settings and combinations of AWS options in order to attempt to determine the issues. We start off by comparing the differences between AWS's dedicated instances versus non-dedicated instances. The results of this experiment are shown in Fig. 3.2. All the experiments run for this test utiliz the same AWS instance type (m3.10xlarge) with the only difference being the AWS option of dedicated instances being turned on for some executions and off for others.

In AWS dedicated instances are instances that run on hardware that is dedicated to a single customer meaning that no other user's VMs will be placed on that physical hardware. This can help to eliminate some of the variability in AWS regarding other VMs utilizing the network extensively. As we see in Fig. 3.2 when only utilizing one or two process per instance the performance is almost identical even when we use multiple instances. However, once we get to utilizing more than four processors per node we see that the dedicated instances start to out perform their non-dedicated counter parts. This discovery led us to investigate the use of dedicated instances more in depth.

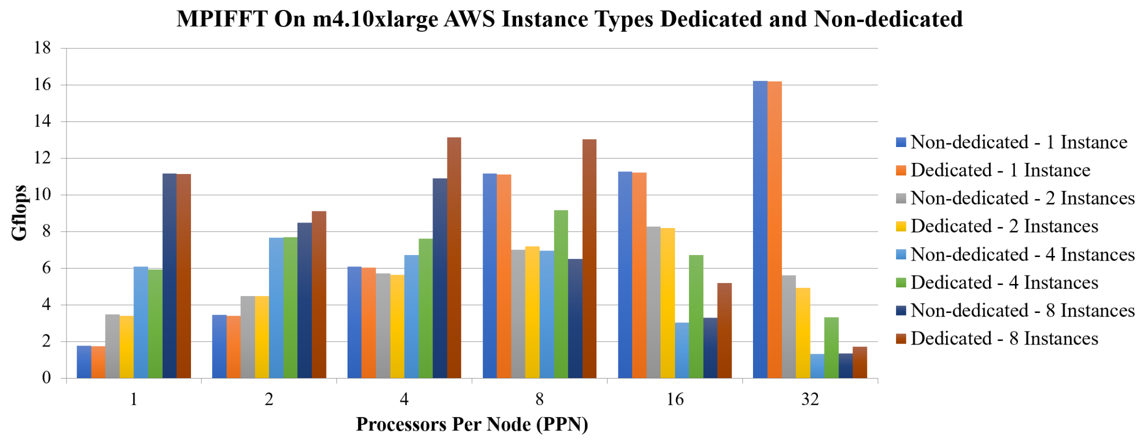


Figure 3.2: Results of the MPIFFT benchmark when run on a different configurations of the m4.10xlarge AWS Instance type.

After determining that utilizing the dedicated option in AWS did seem to help performance, we ran some more experiments utilizing multiple AWS instance types to see which combination yielded the best performance. The results of these experiments are shown in Fig. 3.3. As Fig. 3.3 shows, we varied the instance types, number of instances, and processors per node for each of our experiments. When running on a single dedicated instance type, the numbers were comparable and even a little better with that of the on premise (Holocron) cluster. This is likely due to the slightly newer processors utilized by the AWS instance types. A comparison of the instance types utilized in this experiment is shown in Table 3.1.

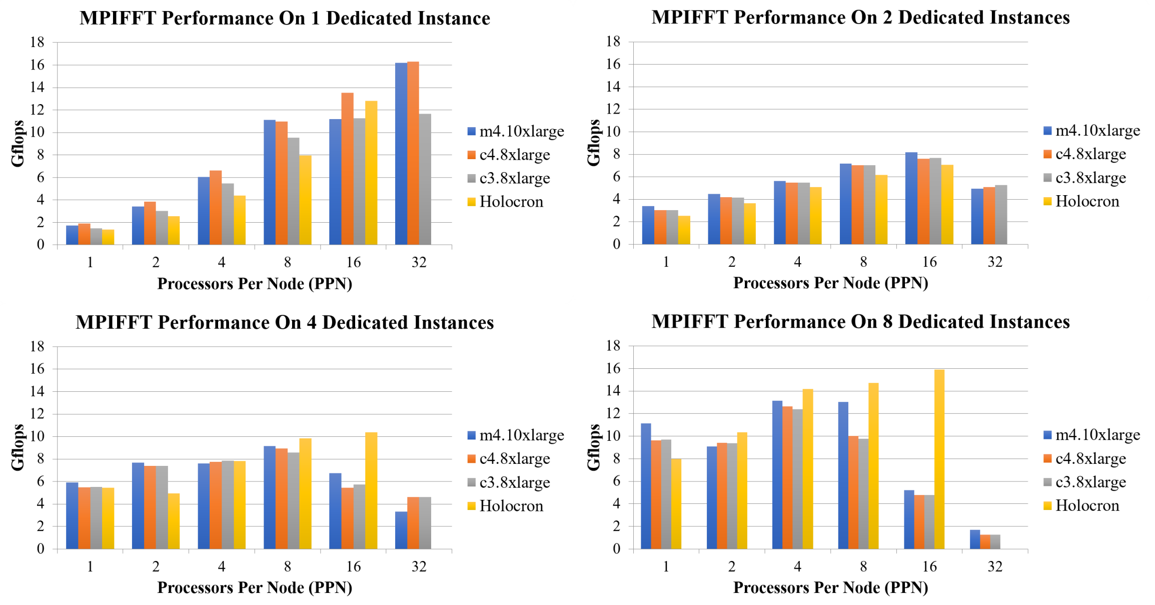


Figure 3.3: Results of MPIFFT benchmark performance across different configurations of AWS dedicated instance types, number of instances, and number of processors per node.

Table 3.1: Compute Node Configuration

Instance Type	Cores/vCPUs	Processor	Memory (GB)	Network
m4.10xlarge	40	Intel Xeon E5-2676v3	160	10GB Ethernet
c4.8xlarge	36	Intel Xeon E5-2666v3	60	10GB Ethernet
c3.8xlarge	32	Intel Xeon E5-2680v3	60	10GB Ethernet
Holocron	20	Intel Xeon E5-2660v2	128	10GB Ethernet

With our single instance experiments as a baseline, we begin increasing the number of instances utilized by the MPIFFT benchmark. The results show that the optimal configuration for the MPIFFT benchmark seems to be any combination of 32 processes divided between the number of instances and the number of processors per node. Any additional processes added above 32 causes a sharp decline in the MPIFFT performance which seems to indicate that there is a bottleneck in the network that is holding the execution back.

This stands in contrast to the local Holocron cluster where the addition of more processors keeps increasing the performance of the benchmark. This underscores the need for a different way of approaching running HPC applications within the cloud as some of the traditional thinking on how to improve performance may actually degraded performance on AWS. The results also show the fact that although the HPC environment on AWS is still slower then the on premise Holocron cluster, there are configurations available on the cloud that will perform decently.

3.4 Conclusions

Although we see that many of the HPCC benchmarks did not perform as well in the cloud as they did on the local environment, the results indicate that the cloud is still viable for executing certain types of workloads. Due to vast number of resources available in the cloud and the quick availability of new hardware, the cloud provides an option to researchers that may not have the required computational resources available to them locally. Although the performance was not as good as locally, for a researcher being able to execute their workload, even at a slower rate, is better than not being able to run at all. By utilizing different configurations and optimizations available within the cloud, performance of some HPC applications can be improved to an acceptable level.

Chapter 4

Dynamic Resource and Workflow Management

Our previous results have shown that it is viable to execute certain classes of HPC applications in the commercial cloud and we discussed a tool to help researchers deploy their own HPC environment within the AWS cloud. One issue with provisioning an HPC environment in the cloud is that the researcher then has to monitor the status of their jobs and ensure that the HPC environment is de-provisioned properly in order to stop incurring charges. This poses a problem for many researchers as some of their jobs may run for multiple hours, days, or even weeks. What researchers require is a tool that can help automate this process.

As previously discussed are a multitude of tools that exist to help manage either the deployment of cloud resources or the execution of workloads on resources. However, there is a noticeable gap regarding tools that can manage both of these aspects. This means that in order to execute workloads in the cloud, researchers either have to create their own customized solutions or take time to learn multiple new tools. In this chapter we discuss the design, architecture, and implementation of the automated Provisioning and Workflow Management Tool (PAW) that was presented at MTAGS17: 10th Workshop on Many-Task Computing on Clouds, Grids, and Supercomputers [75].

PAW is a comprehensive resource provisioning and workflow management tool that automates the steps of dynamically deploying a scalable HPC environment in the cloud, executing a set of jobs or custom workflow, and de-provisioning the HPC environment after all the jobs have completed in a single operation. The key contribution of PAW is that it separates the provisioning of cluster resources in the cloud from the management of scientific workflows on these resources, enabling fine-grained decisions about performance and cost tradeoffs to be made by the researcher. The modular and open source nature of PAW enables any user to create customizable workflow and resource plugins along with cluster environment templates that can be shared with other researchers.

4.1 Methodology

As mentioned there are numerous tools that handle either the provisioning of resources or the management of workflows but there are very few tools that do both effectively. Our research goal is to develop a scalable, easy to use, and extensible tool that can be utilized by researchers to help port their workflows to the cloud efficiently and effectively. The following principles are identified to be the core ideals for this tool:

- **Extensible** - The tool must be written to allow for plugins and extensions to be developed for other resource providers to prevent vendor lock in
- **Scalable** - The tool must be able to create large scale environments quickly and efficiently
- **Limited Dependencies**: To expedite the installation and usage of the tool, it should have a limited number of outside libraries and dependencies
- **Customizable** - Researchers must be able to customize different aspects of the tool in order to meet their research needs
- **Limited Cloud Knowledge** - Researchers must be able to operate the tool with minimal cloud knowledge

In its initial implementation PAW utilizes CloudyCluster which supports the dynamic provisioning and de-provisioning of cluster environments within commercial clouds [68]. CloudyCluster supports provisioning cluster environments that can include shared filesystems, NAT instances, compute nodes, a parallel filesystem, login nodes, and multiple HPC schedulers. CloudyCluster also provides a meta-scheduler called CloudyCluster Queue (CCQ) that provides job-driven autoscaling. The job-driven autoscaling is specified either through special directives within the job script or through parsing the CPU and memory requirements from supported HPC scheduler directives. CCQ allows PAW to submit a job and dynamically create the exact resource required to execute the workflow.

In order to demonstrate the ability of the tool, we present a case study that implements a PAW custom topic modeling workflow to perform a parameter sweep on the full text NIPS conference proceedings [28], and a set of abstracts extracted from computer science publications that were provided by Elsevier. For each of the experiments we chose to utilize AWS as the resource provider and utilize the c4.2xlarge and c4.xlarge AWS instance types which have 8 and 4 vCPUs respectively. These experiments ranged in size from clusters environments of 278 compute instances up to 5,000 compute instances.

4.2 Architecture

A simplified view of PAW's operational stages is shown in Fig. 4.1. PAW separates the operational stages into three parts: Control, Environment, and Workflow. The first operational stage of PAW is the Control stage. In this stage, the resources that are required in order to provision the cluster environment are created. This stage is an optional stage depending on the hardware provider and the provisioning software that is being utilized by PAW. The second stage is the Environment stage where the cluster environment is dynamically provisioned from the configuration defined in the PAW configuration file. The third stage is the Workflow stage where the compute instances are dynamically provisioned and the custom PAW workflows are executed on the provisioned compute instances. After the jobs and workflows are completed, PAW can then automatically begin to de-provision the resources if specified. Due to the decoupling of control planes, the user can also choose to leave the resources running so that they can execute more jobs or perform analysis if desired.

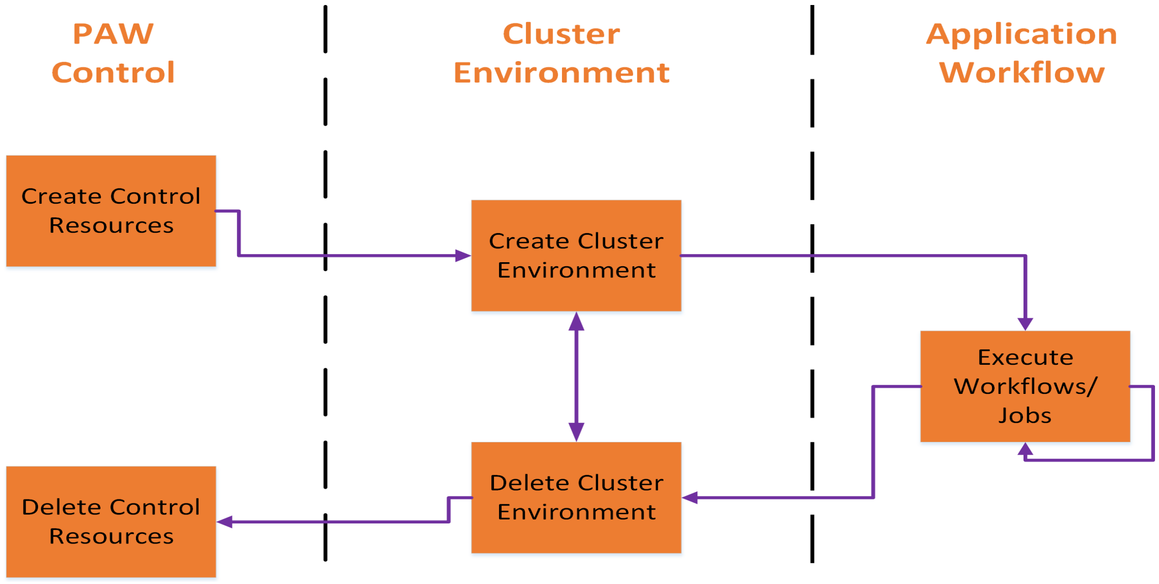


Figure 4.1: Simplified view of PAW's operational stages of execution

By performing the automated deployment and de-provisioning of cloud resources, PAW beings to address some of the billing challenges as described in [14]. Since cloud resources are billed by usage there are some inherent risks involved. If a researcher happens to forget and leave some resources running after their workflow has been completed they will be charged even though no research was being conducted. PAW helps to alleviate this risk through automated de-provisioning.

PAW utilizes a human readable *ini* formatted configuration file to define all the parameters required for execution. Several sample configuration files are included with PAW so that researchers can begin executing workflows quickly. The core of PAW is designed to provide a standardized, simplified, and pluggable interface that can be expanded to support a variety of hardware environments, scheduling software, workflows, and scientific applications.

PAW is written in Python and its core architecture is built from four components: Resources, Environments and Environment Templates, Schedulers, and Workflow Templates. Each of these components is defined by a Python base class that contains the base methods required for each. In order to implement a new type of component a user must create a Python class that implements the methods defined in the base class for that component.

4.3 Resources

The Resources classes of PAW are the components that interface directly with the underlying cloud or resource provisioner being utilized and is optional. The code within a Resource class creates the Control Resources that are required to interface with the cluster environment provisioning tools. However, if the cluster environment provisioning tool does not require the creation of Control Resources, this stage can be skipped. Examples of Resource class implementations can include commercial cloud providers such as AWS and GCP as well as traditional on premise hardware or private clouds like CloudLab [80]. In the initial implementation, the AWS Resource class has been defined and it creates the Control Resources required to utilize CloudyCluster within AWS.

4.3.1 Environments and Environment Templates

The Environment classes provide an interface to the chosen cluster environment provisioning tool and allow PAW to create, delete, and monitor cluster environments of a specific type. A cluster environment is defined within PAW to contain all of the computational resources required to execute the workflows or jobs specified by the user. This type of environment generally consists of a combination of a shared filesystem, a scheduler, NAT instance, compute instances, and a login instance. These combinations of cluster environments can be specified through the included environment template generator.

PAW includes a built-in environment template generator that allows researchers with little cloud knowledge to specify their computational requirements in general terms and have them translated into the parameters that the resource provisioning tool is expecting. This minimizes the knowledge that researchers need to know about the underlying resource provisioners and cluster environment tool. The environment template generator is built to be modular and pluggable similar to PAW and can be extended to other resource provisioners. PAW comes bundled with a CloudyCluster environment template generator that allows users to generate customized CloudyCluster templates to help researchers deploy their environments faster.

4.3.2 Schedulers

The Scheduler classes allow PAW to interface with different HPC schedulers within the created environments. Each Scheduler class provides the required commands and code to submit, delete, monitor and modify jobs within that specific HPC scheduler. For the current iteration of PAW, only the CCQ scheduler is fully implemented. However, CCQ already supports two of the most common HPC schedulers, SLURM and Torque/Maui, so PAW supports these scheduler types through CCQ.

4.3.3 Workflows

PAW also provides a way for researchers to create a custom PAW workflow that can be shared with others. A custom PAW workflow is defined as a custom set of tasks or actions specified by the user that can then be submitted to an HPC scheduler to perform work. These customizable workflows are implemented as a single Python class that has two methods that must be implemented: *run* and *monitor*. A user puts the requisite code to generate or read a batch script file. This code will be called by PAW once the compute resources have been created and then will execute and submit the work to the HPC scheduler. The *monitor* function contains any code that is required to check for the completion of the work submitted by the *run* method. For example, the *monitor* function could include monitoring the number of jobs running with a certain name, or simply checking to see if an output file exists in a certain directory.

A PAW custom workflow is not designed to replace a stand alone workflow management tool such as SWIFT [104] or Pegasus [20], but instead it is designed to compliment them and make them more usable and accessible in the cloud. For example, a custom PAW workflow could include the use of SWIFT to manage the computational workflow while PAW manages the resource management and initial job submission. This allows researchers to utilize the same tools that they currently using which helps to minimize the learning curve and the changes required to their original workflows.

4.4 Case Study

To demonstrate the scalability, flexibility, and extensability of PAW we present a case study utilizing a topic modeling workflow that performs high throughput parameter sweeps. We choose this type of high throughput workflow because this type of workflow does not rely on the network and the processes operate separately from one another. As we discovered in previous work, these types of workflows tend to perform better on cloud resources. We implement this topic modeling workflow as a custom PAW workflow class that enables us to specify the parameters required to execute the job in the PAW configuration file. Once the custom workflow is complete, we utilize PAW to execute the workflow with a single command.

Our custom workflow begins with a human readable experiment descriptor file that contains the basic information about the experiments to be conducted. This information includes things such as the number of compute instances required, which dataset to utilize, and which experimental parameters to test. This configuration file is specified by the user through the PAW configuration file previously discussed.

The cluster environment is a basic cluster environment that consists of an HPC batch scheduler, a shared filesystem, a login node, and compute instances. For these experiments we utilize the c4.2xlarge and c4.xlarge AWS instance types. These instance types have 8 and 4 vCPUs respectively. These instance types were chosen because they fit the computational requirements of the workload and because they were not too expensive. The experiments ranged in size from 278 compute instances up to 5,000 compute instances in a single environment.

The results of the case study are shown in Fig. 4.2. Fig. 4.2 consists of six graphs of which graphs A, B, C, and D show the results from the CS Abstracts experiment while graphs E and F show the results of the NIPS experiments. Graphs A, B, C, and D are truncated on the right hand side for space reasons as these jobs are longer running while graphs E and F show the full execution cycle of PAW as these jobs took less time to execute.

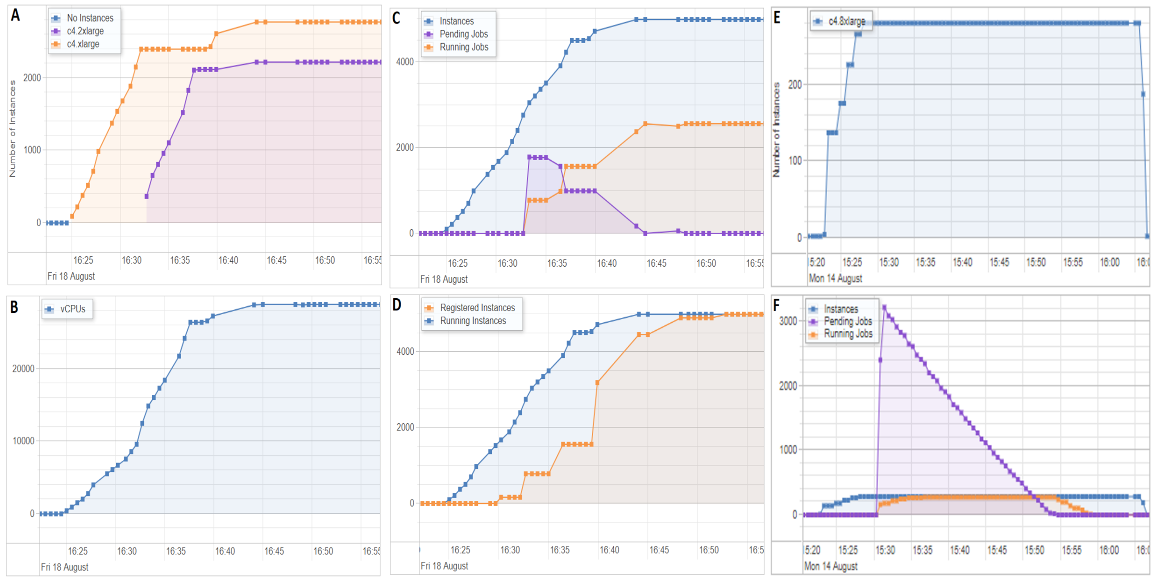


Figure 4.2: Experimental executions of the topic modeling workflow utilizing PAW.

Graph A shows the timeline of the number of each AWS instance type that was launched during the CS Abstracts experiment. We utilize PAW to dynamically launch the instances after the initial workflow had been submitted and were able to provision 5,000 instances consisting of 2,788 c4.xlarge instances and 2,222 c4.2xlarge instances in about 25 minutes. As graph B shows, during this experiment our 5,000 instances had a grand total of 28,832 vCPUs available for computation.

Graph C depicts a timeline of the number of batch jobs submitted by our custom Workflow along with the total number of compute instances provisioned. As mentioned, our custom workflow first submits a single batch job that generates and submits the rest of the batch jobs for the experiment. Graph C shows that the first job is submitted and begins running within 5 minutes of submitting the workflow to PAW and finishes generating the rest of the experiments and submitting them within 8 minutes of launch. As shown, at this point the number of pending jobs within the environment rises quickly to almost 2,000 and then starts to decrease. This is because as more instances are provisioned more jobs start until all the generated jobs are running. Graph D shows an illustration of how quickly instances go from initially being provisioned to being running and ready to be assigned for computation. The first instance registers in within 5 minutes and then 95% of the rest of the instances register within 20 minutes.

Graphs E and F illustrate PAW's ability to not only create resources quickly but also how it can de-provision them quickly as well after the workflow has completed. Graph E shows a timeline of all the instances that were provisioned during these experiments. The graph shows that within two minutes after PAW detected that the workflow had finished, all the computational resources had been deleted. This quick recognition of completion and resource termination is critical for managing costs when running in a cloud environment that is charged by the second.

Graph F depicts the number of pending and running jobs in the provisioned environment as well as the number of instances provisioned. It shows that the first job was submitted within five minutes and that 3,000 jobs were generated and submitted in 30 minutes. The graph also shows that the 278 instances that were provisioned by PAW completed 3,210 jobs in about 30 minutes as well as the detection of workflow completion and subsequent de-provisioning of the environment.

As with running anything in the cloud, there is a cost to execute these environments. For these experiments we utilized the AWS Spot Market to help us obtain the lowest possible cost. The estimated cost for the CS Abstracts workflow is about $777.80 an hour for over

28,000 vCPUs and 5,000 instances. This equates to around $0.028 per vCPU hour, however due to fluctuations in the Spot Market the price could be more or less depending on the market at run time.

4.5 Conclusions

Building off of our previous work in validating and identifying that certain parallel scientific applications can run well on the cloud, we built the Provisioning And Workflow (PAW) Management Tool for parallel scientific workflows. PAW provides a modular base that can be extended to other clouds or resource providers in order to add even more capabilities in the future. We showcase the scalability of PAW by utilizing a custom topic modeling workflow that utilizing the current version of PAW with CloudyCluster and AWS was able to provision a workflow that utilized 5,000 instances and over 28,000 cores in 25 minutes.

PAW was designed to help enable researchers with minimal commercial cloud knowledge to be able to take advantage of some of the advantages that the cloud can offer. By utilizing PAW, researchers can run custom defined parallel scientific workflows within AWS just as they would on a traditional HPC cluster and without any in depth knowledge of how AWS works. Unlike a traditional HPC cluster though, researchers have exclusive access to the cluster environment created by PAW allowing them to not have to contend for resources with other users.

Chapter 5

Considerations When Running At Scale

Now that we know that executing certain parallel scientific workflows on the commercial cloud is feasible and have built a tool to help automate the process, we take a deeper look at the tradeoffs, challenges, and solutions associated with running a very large scale cluster with commercial cloud resources. Due to size of most academic resources as well as the contention by other users of these systems, the actual availability of these systems to individual researchers is limited. There are times where due to other researchers utilizing the system, a researcher's submitted job can sit in the queue for hours or even days depending on its size. This limitation can slow down the time-to-science for researchers who have an urgent need for access to large scale resources (e.g hurricane forecasting). It is in these situations where the commercial cloud could be utilized to allow the researcher access to the resources required to execute their workload.

An example of this is a high throughput computational (HTC) workload which generally consists of independent programs that rely less on low-latency messaging that our previous research has shown can be a bottleneck for HPC applications. HTC workflows can contain a large number of individual jobs which can make acquiring the resources to execute them in a fixed turn-around time on traditional HPC resources difficult. Limitations for this could be the physical size or availability of the networking, computational, and storage resources on site or how many jobs a user can submit to the system at once. When there are not enough resources available, the number of HTC jobs that can run simultaneously is decreased which increases the amount of time required to execute the entire suite of jobs. If the HTC suite is massive, say consisting of tens of thousands of jobs, researchers may have to wait for weeks to get enough jobs to execute for a usable results.

The commercial cloud has the allure of “unlimited” resources, the ability to have complete control over administrative policies of the environment, and an increasing variety of hardware that can help solve many of the issues faced when attempting to run a massive HTC workflow. However the challenges with dynamically provisioning traditional HPC-type environments at a massive scale on the commercial cloud is not well studied. There have been a number of studies that have looked at optimizing various HPC applications in the cloud, HPC application performance studies, and studies about which applications and programming models are best suited for cloud [83, 36, 93, 32, 34, 52, 29]. However, the scale of these experiments has been modest by our scale ranging from a few nodes to a few thousand instances in total.

At the time of this research, the largest scale HTC or HPC job that had been executed on a commercial cloud and publicized was an HTC job that was designed to execute a single workflow on GCP [10]. At its peak, the job utilized 580,000 cores simultaneously. However, this suite of jobs executed utilizing a work queue model in which compute resources receive work from a queue master similar to [84]. All of the instances utilized operated as separate entities and were spread over multiple GCP regions and the submission tools were a different from those utilized by traditional scientific computing environments as there was no HPC batch scheduler utilized.

Our goals for this research are the following:

- Identify the challenges that come with provisioning a massive scale HPC environment on the commercial cloud

- Propose potential solutions to these challenges
- Design an architecture that will allow us to execute an environment consisting of at least 1,000,000 cores
- Deploy a Topic Modeling Parameter HTC workflow on the massive scale environment

5.1 Methodology

For this study we consider the three largest commercial cloud providers, Amazon Web Services (AWS), Google Cloud Platform (GCP) and Microsoft Azure which each offer a number of different services and each have their own advantages and disadvantages. But after doing to the comparison, we chose to utilize AWS as it offered the most effective cost saving options via the AWS Spot Market and also provides the largest selection of off-the-shelf provisioning tools. We also had previously developed an automated Provisioning And Workflow management tool [75] to help deploy HPC environments on AWS which we utilize for this experiment.

AWS also has a concept called an AWS Spot Fleet which can utilize multiple AWS Instance Types within a single Autoscaling group based upon a specified "weight". This is important because in order to reach our goal of provisioning over 1,000,000 cores we need to ensure that if a single instance type has a price spike during the experiment we have another instance type that can be launched instead.

In order to test our scaling limitations we develop a test plan that calls for experimental testing at increasingly larger scales. We first start out with a modest test of just 10,000 vCPU or about 1% of the goal and then move to a medium sized test of about 5,000 instances which is designed to test the scheduler software and provisioning software limitations as well. Successful resolution of these limitations within a single 5,000 instance environment laid the groundwork for executing multiple of these environments in our final execution. It should be noted our medium sized cluster is comparable to many campus-scale computing clusters.

Table 5.1: 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

5.2 Limitations To Scaling

This research identifies and resolves a number of different limitations to massive scaling on the commercial cloud. Of these limitations, several of these are common to execution on all commercial clouds, including those of a shared filesystem, network limitations such as a NAT instance, launching heterogeneous instance types to control costs, HPC scheduler stability, and cloud vender user and API limits. While some limitations such as the dynamic pricing effect on the Spot Market are specific to AWS. Table 5.1 lists a summary of the limitations we identified during this research and our proposed solutions for each limitation.

5.2.1 Shared Filesystem

The scientific HTC workflow we are utilzing was originally designed to generate the analysis jobs based on a specified configuration file, and each of these generated jobs required specific data that was located on a shared filesystem. Without access to this data on the shared filesystem, the workflow could not execute successfully. However, our massive scale environment will contain tens of thousands of instances and it is known that globally shared filesystems, both in the cloud and on premise do not scale well in general [4]. Even a robust shared filesystem would have issues when all jobs require access to the filesystem at the same time.

To address this issue we attempt a few different solutions. The first solution is to create a version of the workflow that submits a parent job that copies the dataset to a local scratch filesystem, generates the jobs for execution, and uploads a tar file of the generated jobs to Amazon S3. Amazon S3 is an object store service that is designed to be highly scalable and available globally.

In this scenario, when each compute instance is assigned a job, it will download the tar file from S3 to the local instance, extract it, and then execute the application. This avoids using a shared filesystem but still allows each instance to have access to the data it requires to execute and maintains the dynamic and flexible nature of the workflow. This solution worked well for the small tests, but issues accessing the same file began to occur during our medium sized tests. As multiple thousands of instances attempted to download the same file we began to see timeouts and the jobs would fail. In order to combat this, we attempt a second solution which involves uploading multiple copies of the tar file to S3 in order to decrease the load on a single object. This solution decreased the number of timeouts but did not eliminate them, which led us to another limitation within the network infrastructure utilized to access S3 at a massive scale.

5.2.2 NAT Instance

Our cluster environments utilize a Network Address Translation (NAT) instance in order to allow compute instances to download files and communicate with particular AWS APIs. This is required because the networks that AWS utilizes for EC2 instances, Virtual Private Clouds (VPCs), are private by default. In order to communicate with servers outside of the VPC or other AWS APIs the NAT process must be utilized. By utilizing PAW we were automatically provisioning a NAT instance, however, we were exceeding the throughput of our provisioned instance which was causing the timeouts we were seeing when downloading the dataset.

To solve this problem, we identify two potential solutions: NAT Gateways or VPC

Endpoints. A NAT Gateway is a managed AWS service that handles NAT operations, however you are charged for the amount of data processed by the NAT Gateway which can add up quickly if you are downloading and uploading lots of data. So in order to avoid this extra charge, we instead utilize VPC Endpoints which allow instances inside a VPC to communicate directly with supported AWS services, such as S3, without having to go through the NAT process. However, due to time constraints we did not have time to fully test the VPC Endpoint solution so we attempt one final solution. For the final execution we pre-generate the required datasets and experiment files and build them into the machine images that the compute instances utilize. This solved the network limitation of the copy and reduces the chance of failure when running at a massive scale. However, this does drastically limit the flexibility and dynamic nature of the workflow.

5.2.3 Dynamic Pricing Effects On The Spot Market

When conducting our initial medium scale experiments, we utilize the AWS default *Lowest Price* allocation strategy. In this mode, an AWS Spot Fleet (collection of compute instances) launches Spot Instances into the Spot Pool with the lowest Spot Price [86]. But during testing we observed long periods where the launching of instances leveled off for long periods before beginning to launch more instances. Further investigation into this issue revealed the cause.

Amazon Spot pricing changes based upon the supply and demand for different instance types and by requesting all of our Spot Instances in the same Spot Pool we were actually driving up the price for the instances we wanted to use. Once the Spot Price reached our maximum bid price (i.e., the max amount we are willing to pay for a Spot Instance), the new launches would fail. However, utilizing the *Lowest Price* allocation strategy the Spot Fleet would continue to attempt to launch instances into that Spot Pool for a few minutes before it reached a maximum number of failures and moved to the next Spot Pool.

To remove this delay, we utilize the *Diversified* allocation method. This allocation method attempts to spread out the launching of the requested Spot Instances between multiple Spot Pools which prevents the price from spiking in a single Spot Pool. This also decreased the cost of the experiment as instead of paying the maximum amount in particular Spot Pool now the prices would be more consistent.

5.2.4 Heterogeneous Instance Types With Spot Allocations

An Amazon 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 limits the target capacity for each Spot Fleet Group to no more than 3,000. This posed a problem as this limitation requires the management of an excessive number of Spot Fleets to obtain the desired size and is a limitation to scalability.

Spot Weight parameters define the number of capacity units represented by a single Spot Instance type [86]. There are two default modes for specifying the Spot Weight parameters for each Spot Instance type: Instance mode and vCPU mode. In Instance mode, each Spot Instance counts as a single instance so the Spot Weight is set to 1 whereas in vCPU mode the Spot Weight is set to the number of vCPUs each Spot Instance has. This is useful because since different Spot Instance types have different numbers of vCPUs some instances should count for more then other instances. However, neither of these solutions fit our needs exactly.

A key observation is that the Spot Weight parameter can be a fractional value less than 1.0, the effects of this is shown in Table 5.2. As Table 5.2 shows, using vCPU mode and a capacity of 3,000 the total instances per fleet is only 83. However, a Spot Fleet Weight of 0.05 provides 20 instances for each capacity unit yielding a total of 60,000 instances with a capacity set to 3,000. This is how we accomplish launching more than 3,000 instances in a single Spot Fleet.

Determining the optimal values for these Spot Weights is a process of trial and error because as the Spot Weights are modified so are the types of instances that will be launched. The adjustments that we make centered on attempting to ensure that the Spot

Table 5.2: 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 5.3: 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	$1.570	0.064	320	1
	m4.16xlarge	$1.670	0.064		
Large	c4.8xlarge	$0.800	0.036	160	4
	c3.8xlarge	$0.876	0.032		
	r4.8xlarge	$0.700	0.032		
Medium	c4.4xlarge	$0.370	0.014	70	3
	hi1.4xlarge	$0.370	0.014		
	i3.4xlarge	$0.418	0.015		
	r4.4xlarge	$0.470	0.015		
	m4.4xlarge	$0.514	0.015		
Small	m4.2xlarge	$0.258	0.007	32	5
	m3.2xlarge	$0.236	0.007		
	c4.2xlarge	$0.190	0.006		
Tiny	c4.xlarge	$0.100	0.025	100	1

Fleets launch instances with higher vCPU counts over small vCPU instances. This type of adjustment is required at the massive scale we are running at because if all the instances launched were smaller 4 vCPU machines we would need 250,000 instances in order to reach our goal of one million vCPUs whereas if we launched all 36 vCPU machines we would only need 27,778 instances total to reach our goal.

After experimenting with the different configurations, we find that we need to create "workflow classes" that only contain instances that have similar characteristics due to the difficulty in getting a Spot Fleet to launch larger expensive instances over smaller cheaper instances. By creating these workflow classes we ensure that each Spot Fleet will launch the proper combination of instances and we will get a certain amount of capacity from each Spot Fleet. The workflow classes are detailed in Table 5.3.

5.2.5 Scheduler Scalability

Another bottleneck is revealed by the medium scale tests of a few thousand compute instances: the scalabilty of Salt-Stack and SLURM. SLURM is an HPC batch scheduler through which users submit batch jobs for processing. SLURM schedules the batch jobs on the available resources and according to the SLURM website, the largest SLURM cluster contained 98,304 nodes which is way less than our theoretical maximum of 250,000 [85]. This is our first limit, keep the size of the environment under 98,304 instances to avoid issues.

SLURM, like most HPC schedulers, is built to be statically configured where the node configurations are known ahead of time and are coded into the SLURM configuration file. This works well in a traditional cluster environment where the IP addresses, hardware configurations, and total number of nodes is know but in the cloud all of this information isn't decided until execution which makes configuring the scheduler more difficult. All of the configuration information has to be added to the configuration file and then the configuration file has to be pushed to all the instances.

Within PAW, CloudyCluster utilizes SaltStack to push out the configuration file to each of the compute instances from a single Salt Master. In our massive configuration, this becomes a bottleneck in our deployment. Although there is no technical limit to the number of Salt Minions (i.e. compute instances) supported by a single Salt Master, the amount of resources required to operate the Salt Master increase with the number of Salt Minions. Typically this bottleneck is solved by utilizing *salt-syndic* but that feature is not yet implemented in CloudyCluster. To avoid this bottleneck we decide that instead of creating one massive environment we will create multiple environments that each will contain a maximum of 5,000 instances in order to ensure we can stay within the limits of SLURM and SaltStack.

The creation of multiple cluster environments is also a strategic optimization of the management of of the massive number of resources while not detracting from the execution of our HTC workload. By creating multiple environments we are able to minimize the effects from a failure, stay within the scaling limits of SLURM and SaltStack, and maintain the dynamic nature of our workflow. In the case of a catastrophic failure in a single environment the experiments executing in the other environments will remain unaffected ensuring that we get some usable results. This also allows us to partition the workflow and run different analysis on different environments.

5.2.6 User Limits

As is standard practice with all commercial cloud providers, AWS imposes certain limits on the number of resources that a user can create. This is a safety feature for the user and the cloud provider as it ensures that the cloud provider can support the requests of all users and it prevents users from having an unexpectedly large bill. There are mechanisms provided in order to increase these limits which at the scale that we want to run at will be required.

The first limits that we need to raised are the EC2 On-Demand Instances, VPC, VPC Endpoints, and EC2 Spot Instance limits. These limits prevent the launching of a large number of instances and a large number of networks which we will be requiring for our multiple environment setup. Typically this is done by filling out a request form, but for the large requests that we required we worked directly with AWS associates to help get these limits raised.

Another limit that was crucial to the execution of the experiment is the Amazon Elastic Block Storage (EBS) limits. All of the instances that we launch will utilize an EBS volume to store the OS and other data for the machine. A default AWS account is limited to 20TB of EBS storage, but each compute instance in our experiment requires a 40GB EBS volume which means that 20TB would only get us to 500 instances. In the most extreme case utilizing 250,000 4 vCPU instances would require 10PB of EBS storage. We were able to get our limit raised to 8PB as we took precautions to make sure we did not end up with the worst case.

5.2.7 API Limits

Cloud Providers also implement limits on the number of API calls that can be issued by a certain account within a certain period of time. This helps to mitigate API misuse and abuse while helping to keep the system from becoming overloaded. This API throttling is also a preventative measure to make sure one rouge customer doesn't affect other customers. While running at a massive scale with tens of thousands of instances making API calls at the same time from the same account generates a large number of legitimate API requests in a short amount of time which can look similar to a denial of service attack.

This is an issue that we encountered during our execution. As shown in Fig. 5.1, around the 600,000 vCPU mark the graph slows down and doesn't increase as quickly. At this point in the experiment we noticed that our calls to launch new instances were consistently failing, this was due to the large number of API calls generated in a short period of time. So in order to reduce the throttling, certain monitoring tools were turned off and we decreased the launch rate of new Spot Fleets.

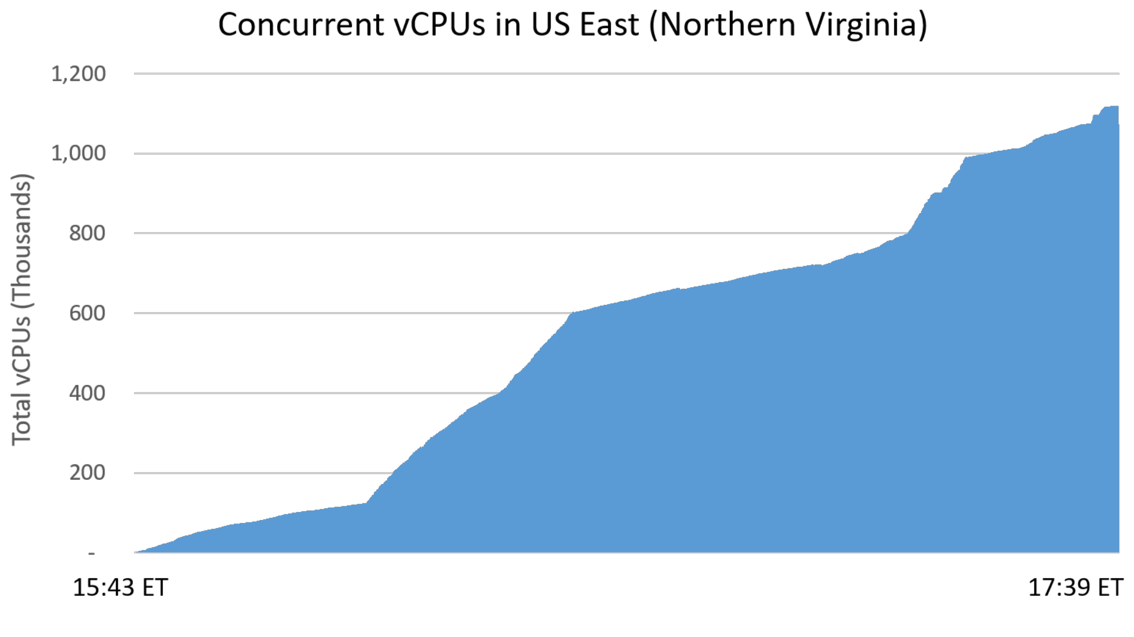


Figure 5.1: Timeline of vCPU count from beginning to peak vCPU use count.

It was later determined during post execution analysis that the throttling came from our use of the *describeInstances* API call. This call is utilized to obtain statistics about our AWS account, the running instances, and how we observe the execution progress. Another solution that was discussed to replace this call in future experiments was using Amazon Cloud Watch which provides similar information but requires less API calls.

5.3 Execution and Evaluation

The HTC workload of choice for the experiment is a topic modeling application based on Latent Dirichlet Allocation. While these models are widely used, the evaluation of their outputs and sensitivity to the various input parameters is an active area of research [13].

This scientific workload examines the impact of the alpha, beta, and topic count parameters which are required input by the topic modeling algorithm. This workflow examines two different datasets and performs a wide sweep of hundreds of thousands of parameter combinations with each job within the workflow executes Parallel Latent Dirichlet Allocation (PLDA) [102]. The two datasets are the full text conference proceedings from Advances in Neural Information Processing Systems (NIPS) [28] and seventeen years of abstracts from a wide variety of computer science publications provided by Elsevier Scopus.

5.3.1 Execution

The workload executed across a number of identically configured environments with the only differences being the selection of instance types utilized for the compute instances. In preparation for the launching of the compute instances, we launched the core components only of forty cluster environments, which provided us with extra environments in case that some failed during deployment. The environments were launched in two different stages: minimal set of cluster environmental components and the launching of the compute instances. During the first phase, PAW launched the Control, Scheduler, NAT, and Login instances along with creating the VPC and other network components required. The configurations were all defined via a customized PAW environment template.

The second phase involved the launching of all the compute instances within the different created environments. This was accomplished through a customized PAW workflow class that allowed PAW to process the topic modeling workflows and automatically submit the jobs to the local environments utilizing CCQ which then handles the provisioning of the compute instances utilizing AWS Spot Fleet instances. A single "workflow class" was launched on each created environment, these workflow classes are described in Table 5.3.

The first workflow class was launched 3:43PM EST and 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 total. Although we had 40 environments launched and ready, we only ended up requiring 14 of them to reach our goal of one million vCPUs. Due to the previously mentioned API throttling, we had to slow down the launching of our workflow classes around the 600,000 vCPU mark.

At 5:39PM EST, we reached the peak vCPU count of 1,119,196 vCPUs running within 49,925 Spot Instances spread across 12 different AWS Instance types. A breakdown of the toal number of vCPUs per instance type at the peak time is show in Fig. 5.2. In total 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 even weeks.

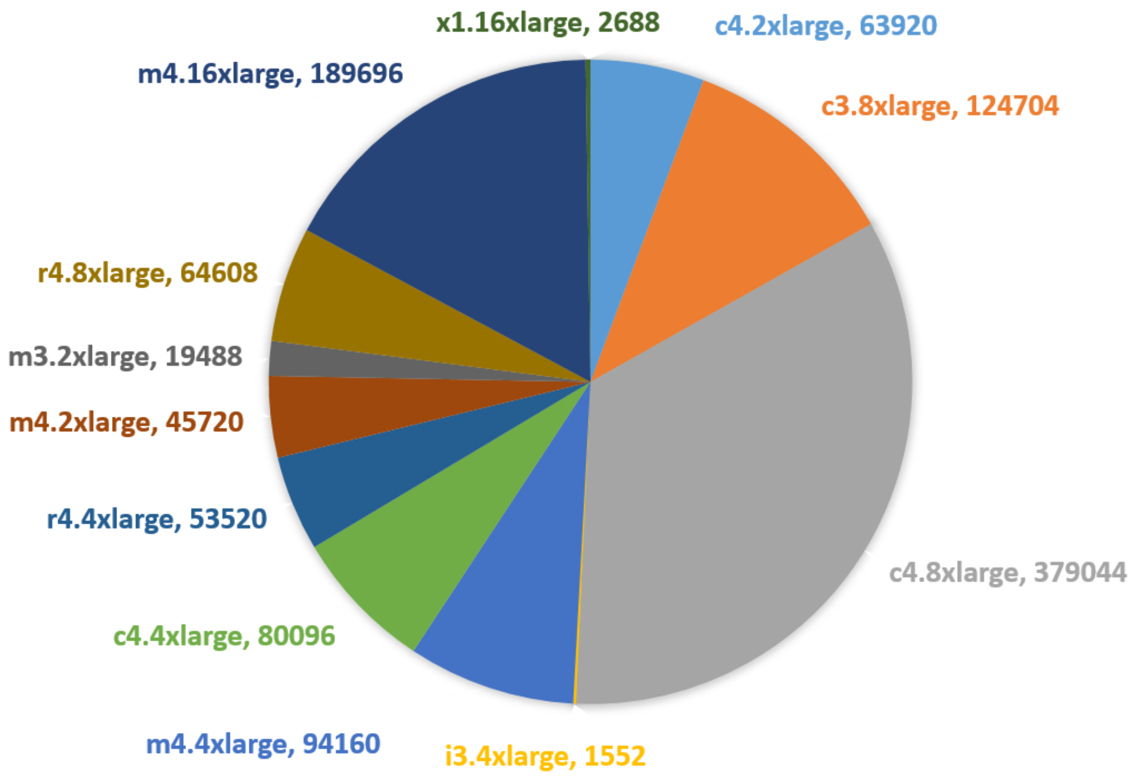


Figure 5.2: Distribution of vCPUs among the instance types utilized at peak.

5.3.2 Cost

When utilizing commercial cloud resources one important factor is always cost. However, the cost comparisons between local resources and commercial clouds are complex and depend on a variety of factors such as size, maintenance costs, characteristics of the workflow, and the utilization of the resources. Cost estimation done at a representative university shows on premise computational costs to be well under US \$0.02 per core hour regardless of job type. This estimate includes costs for hardware, software, space, power, cooling, labor, network, etc but does not include the user-facing services such as user support and research computing facilitation [65].

We set our maximum Spot Bid Price per instance type at the price that would keep us close to the \$0.02 per vCPU hour. This helped us to narrow down the instance types to use for the experiment and keep costs low. We also utilized the Spot Weight feature to drive our Spot Fleets to launch certain more desirable instances instead of the less desirable instances. A breakdown of our Spot Fleet Weights per instance type is shown in Table 5.3.

Another issue encountered when executing at scale is that when running a massive number of resources users start to compete against themselves which drives up the Spot Price and the overall costs. We find that launching many instances of the same type quickly raised the Spot Price up to our maximum bid price. This fluctuation in the Spot Price is illustrated in Fig. 5.3. As shown in the figure, at the beginning of the experiment, the prices were near US $0.40 for all Availability Zones and that the prices rose by as much as a factor of two during the experiment.

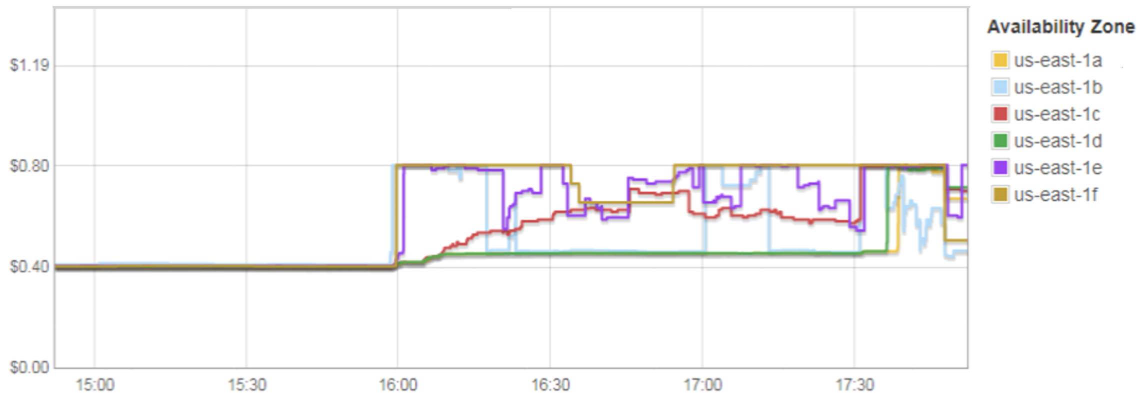


Figure 5.3: The effect on the Spot Prices of the c4.8xlarge instances in each Availability Zone.

During the experiment we utilized a total of 1,832,923 vCPU hours at an average cost of US $0.0172 per core hour. While there are other costs that need to be considered when running workflows in the AWS cloud, the costs for network egress and data storage were negligible. The total cost of our computational resources for the two hour experiment was US $32,423. A summary of the cost per vCPU hour broken down by instance type and workflow class is shown in Fig. 5.4 while a graph showing the cost per vCPU hour, target price, and average price for all instance types is shown in Table 5.4.

Table 5.4: 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

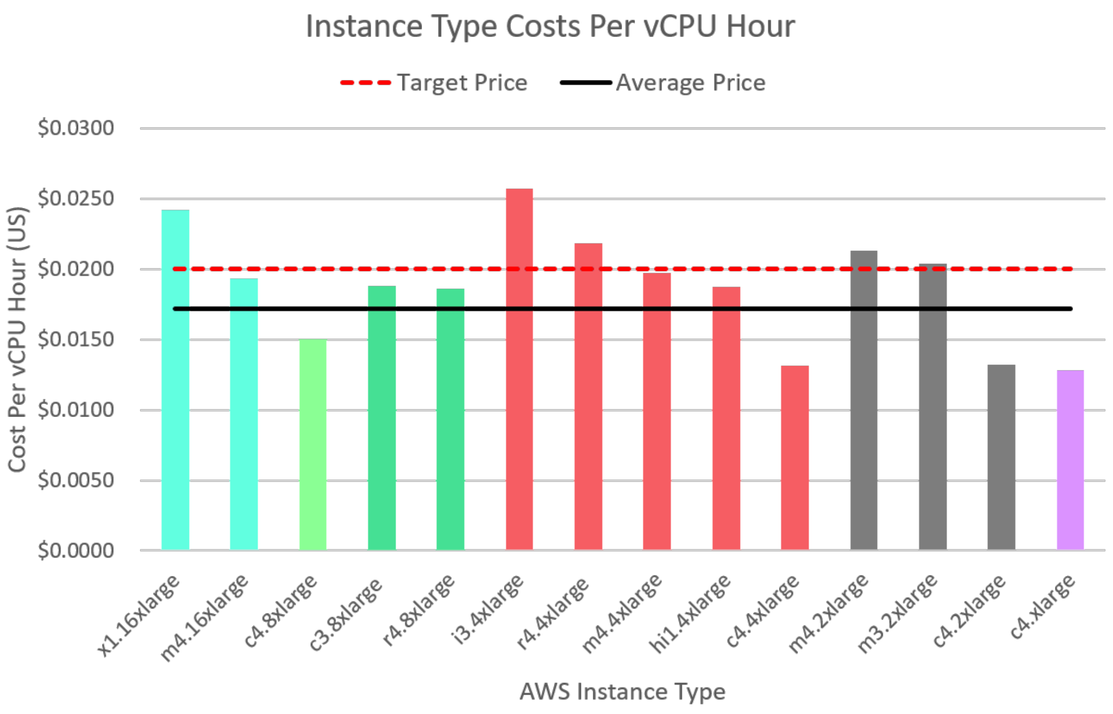


Figure 5.4: Costs per vCPU hour of each AWS Instance Type used in the experiments. The solid black line is the average price per vCPU hour across all instance types and the red dotted line is the target price per vCPU hour.

5.4 Analysis

This research identifies and resolves a number of limitations to massive scaling on the commercial cloud. Several of the limitations that are identified and resolved are common to execution on all commercial clouds including those of a shared filesystem, network limitations, launching heterogeneous instance types to control costs, HPC scheduler scalability, user limits, and API limits. We also provide a detailed cost analysis for running on AWS and discuss how costs may be lower with smaller workflows utilizing the same technologies.

We are able to utilize the automated Provision And Workflow management tool (PAW) [75] for the lifecycle tasks of cluster provisioning, workflow execution, and cluster de-provisioning. By utilizing PAW, we are able to execute our custom topic modeling workflow on 1,119,196 vCPUs simultaneously with minimal user input at an average cost of US \$0.0172. Utilizing these resources we are able to execute just under a half a million jobs in two hours which would have taken days or even weeks on a shared local resource. The execution of a massive HTC application in the commercial cloud is a promising demonstration of how the commercial cloud can be utilized for scientific applications.

Chapter 6

Commercial Cloud Infrastructure for Transportation Cyber Physical Systems

We have shown in our previous work that the commercial cloud has clear benefits for dynamically deploying HPC environments and workloads on-demand. We now dive deeper into how these same deployment patterns, constructs, and tools utilized for our previous work can be applied to the infrastructure supporting Transportation Cyber Physical Systems (TCPS). We will discuss the current state of TCPS and what unique challenges these types of systems face. One of the major patterns that we will discuss is how utilizing the infrastructure as code (IaC) paradigm within the commercial cloud can help simplify the creation and maintenance tasks of deploying TCPS infrastructure within the cloud.

6.1 Introduction

Transportation Cyber Physical Systems are systems that tie together both the physical transportation system (including automotive, aviation, and rail systems) with the cyber systems that help to detect accidents and other malfunctions within the physical transportation systems. TCPS systems collect data from a wide variety of sources including multiple transportation modes and a wide variety of data collection devices. The data within these systems can be described utilizing the “5Vs of Big Data”: (1) volume, (2) variety, (3) velocity, and (5) value [21].

In this context, volume refers to the amount of raw data that needs to be stored. Velocity refers to the rate at which data is being collected, transferred to the storage infrastructure, and how quickly the back-end services are expected to process the data. Variety refers to the numerous types of data that come from a large number of datasources and data formats that will be utilized and processed by the system. The veracity aspect comes from the reduced reliability of the data after all of the data transformation has been completed. While value refers to the desired outcome for the processing of the TCPS data.

These characteristics make traditional data processing and delivery systems inadequate for many types of TCPS data analysis and decision support tasks as within these traditional systems large amounts of heterogeneous data cannot be processed in real-time. In our work, we introduce modern data infrastructures that are needed to support TCPS. We also discuss the leveraging of the infrastructure as code paradigm that can utilize commercial cloud facilities. Utilizing infrastructure as code (IaC), system administrators can better track changes to their infrastructure allowing them to quickly and efficiently rollback or upgrade their systems via traditional version management systems [76].

6.2 TCPS Infrastructure

In this section we discuss the overall TCPS infrastructure which includes both the networking challenges as well as two different data processing architectures: the LAMP architecture and the Lambda architecture. We discuss how the different types of network capabilities available can influence the overall design of the TCPS as well as which components are located at the edge or in the commercial cloud. We also discuss how the traditional LAMP architecture does not fit the profile of the TCPS data and how the proposed Lambda architecture addresses some of the gaps within the LAMP architecture that make it more suitable for TCPS data.

6.2.1 TCPS Networking

In a TCPS there are a large number of actors within the system that all need to cooperate and communicate with each other quickly and efficiently. Within TCPS infrastructure, people, and vehicles all have to collaborate and coordinate to support the application requirements of both end-users and other stakeholders such as motorists and automotive-related industries. The architecture for the data infrastructure of TCPS can be either centralized, distributed, or centralized-distributed. However, regardless of the architecture the data infrastructure must be able to communicate with data senders, data receivers, and other data infrastructure components. Typically these communications will take place through either a wireless or wired mediums. However the exact medium to utilize for communication will depend on a number of factors such as the application requirements and the availability, range, delay, and bandwidth of the medium itself. Depending upon the application, there may be multiple layers of data storage and processors that all need to communicate with each other. The number of data infrastructure layers will be selected based upon how many are needed to reduce data delivery delay, bandwidth, and data loss rate [79]. When deciding on whether not to utilize the commercial cloud for these applications one must take these factors into consideration.

Although the processing power that is available within the commercial cloud makes it great for scalability, there are certain aspects such as data movement and network bandwidth that can hinder its usefulness in certain scenarios. This is especially true for safety critical information that has specific real-time processing requirements. For these types of applications, the commercial cloud may not be the best solution as the latency introduced when sending the data to the cloud and back to the device or person could exceed the required threshold. However, for an analysis that takes into account historical data or for running simulations based upon the collected data without real-time constraints the commercial cloud could be a useful tool within a TCPS system.

6.2.2 LAMP Architecture

Traditional data processing or web service environments typically rely on standard web service stacks, which typically rely on a LAMP architecture [46]. The LAMP acronym stands for the **L**inux operating system, **A**pache HTTP Server, **M**ySQL relational database management systems (RDBMS), and the **P**hp programming language. A typical LAMP architecture deployment is shown in Fig. 6.1.

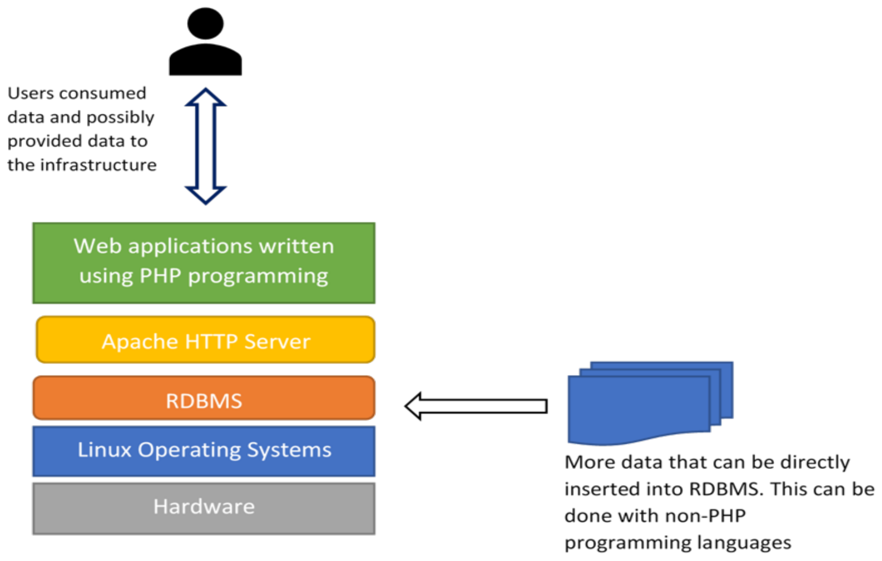

Users consumed data and possibly provided data to the infrastructure
Web applications written using PHP programming
Apache HTTP Server
RDBMS
Linux Operating Systems
Hardware
More data that can be directly inserted into RDBMS. This can be done with non-PHP programming languages

Figure 6.1: Overview of LAMP architecture for traditional data processing or web service environments.

In this architecture each layer serves a specific and integral purpose. Linux is responsible for providing a reliable and open source operating system running on top of the hardware resources. The RDBMS system is where all of the data is stored and accessed and is thus the core data infrastructure. The Apache HTTP server provides users and interface through which users can access applications that have been written using PHP or other programming languages. In the LAMP architecture, the RDBMS and the applications running on the Apache HTTP server are typically responsible for all the data collection, insertion, and presentation. The LAMP architecture works well for smaller web based services and small scale data processing but when scaling up to larger datasets with multiple sources the architecture can be strained which can limit the ability to analyze and access the data

6.2.3 Lambda Architecture

While the LAMP architecture works well for many scenarios, TCPS have a unique set of complex data challenges that do not fit well into this architecture and therefore a new architecture is required. This new architecture must be able to accommodate the "5vs of Big Data" that we have previously discussed. One architecture that has been proposed to help tackle this characteristics is the *Lambda Architecture* (LA) [55]. The Lambda architecture is broken down into four primary components: the *data layer*, the *batch layer*, the *stream layer*, and the *serving layer*. These four primary components of the LA architecture are shown in Fig. 6.2

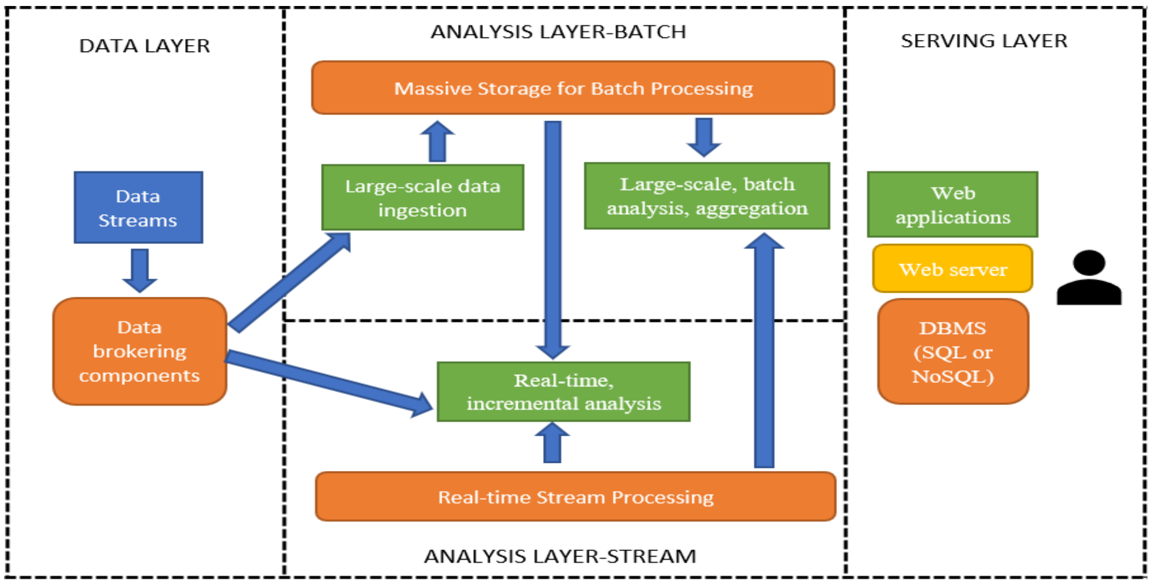

DATA LAYER
ANALYSIS LAYER-BATCH
SERVING LAYER
Massive Storage for Batch Processing
Data Streams
Large-scale data ingestion
Large-scale, batch analysis, aggregation
Web applications
Web server
Data brokering components
DBMS (SQL or NoSQL)
Real-time, incremental analysis
Real-time Stream Processing
ANALYSIS LAYER-STREAM

Figure 6.2: Overview of the proposed Lambda (LA) architecture for data infrastructure.

In the following subsections we will discuss each primary component in detail along with how it fits within a TCPS. We will also discuss how each of these components could possibly be implemented or migrated to the commercial cloud to improve the scalability and flexibility of the architecture and TCPS.

6.2.3.1 Data Layer: Collection and Brokering

One of the major challenges of TCPS data is that it is coming from a wide variety of data sources. These sources can include devices such as sensors and cameras connected to traffic lights, connected vehicles, satellites, and even motorists. Each individual source produces different types of data that can pertain to different aspects of the TCPS such as location, weather, route of travel, accident avoidance, and overall infrastructure performance. Most of the time this data is being sent in a variety of different formats which all require a different type of processing to make use of the data. Some of the data requires no processing, such as the count of vehicles that pass a specific point over a period of time. While other types of data need additional processing before it can be utilized, for example a video feed from a CCTV camera. This wide variety of data and processing power required greatly shapes the type of data infrastructure that a TCPS utilizes.

The data layer is responsible for the storage of this wide variety of data which for TCPS can be on the scale of petabytes. This scale can be problematic for traditional data storage systems such as those found within the LAMP architecture. There are also additional unique requirements that are placed on TCPS data from the users of the data. For instance, a connected vehicle needs to be able to make real-time decisions based upon data coming from various sensors to determine if it needs to apply the brakes or not. If the data infrastructure cannot handle the volume of data or becomes unavailable the vehicle could crash and cause both physical and monetary damages. This also leads to the issue that the data layer needs to be not only fast and reliable but also secure and scalable as well. As the number of connected vehicles grows the data infrastructure must scale accordingly.

Another unique aspect of TCPS data and the corresponding data layer is the way that the data is transmitted to the data layer. Some of the data that is collected by TCPS comes from sensors and other devices that are mounted on moving vehicles which poses another unique challenge. Due to the constant movement of the vehicles maintaining a constant connection that can be used to transmit the data can be difficult. This means that generally some of the data will end up lost in transit due to the constant movement. This requires a robust data ingestion system that can handle this scenario. This also leads to the question of where to host the data infrastructure: locally or in the cloud? There are different advantages to each solution. Keeping the data infrastructure local will help to cut down on latency in the transmission of the data and can allow for more real-time decision making but at the expense of maintainability and scalability. Whereas locating the data processing infrastructure in the cloud allows for greater scalability and maintainability as the infrastructure can grow and shrink on-demand but will add additional latency to the collection of the data.

Within the data layer, the data brokering component ties both the batch and stream layers together. The data brokering component provides several critical services to the entire infrastructure such as improving data stream accessibility by duplicating and partitioning the data as well as reducing the risk of data loss when services within the batch and streaming layers fail. With these additional services, a TCPS data infrastructure resembles a large-scale distributed system. This means that a TCPS data infrastructure faces similar challenges as a large-scale distributed system such as complex environments, heterogeneity in hardware and software components, dynamic flexible deployment, and high reliability, throughput and resiliency [18]. However, this also means that TCPS data infrastructures can utilize the same solution employed by these large scale distributed systems. This solution is to implement a message-orientated middleware (MOM) solution.

In this solution, messages are data elements being transferred from data sources to various data storage and processing components. Inside of this MOM architecture data is sent from *producers* (sources) to a *broker* (queue) the data can then be pulled by the *consumers* (destination processing entities) or pushed to the consumers by the broker. An acknowledgement may be sent back to the broker by the consumer which can then be pulled or pushed by the producers.

There are a number of MOM software frameworks that are able to provide data brokering services. Among these software frameworks is Apache Kafka which is the latest framework designed for use within an enterprise production environment [101].In Kafka, producers sent their data to the broker layer and data that is expected to be consumed by the same consumers is directed towards a specific and unique *topic*. Multiple producers and consumers can publish and retrieve data from the same topic or from different topics at the same time. The number of topic partitions and brokers can be scaled dynamically to cope with changing data demands. Kafka demonstrated a significant improvements in overall data throughputs over ActiveMQ and RabbitMQ, two other popular MOM frameworks [101].

An important and critical requirement of the TCPS data infrastructure is that it has to be setup so that processes from both the batch and streaming layers can have concurrent access to the data. This way the TCPS data infrastructure only has to ingest the data once and then both the batch and streaming consumers can pull the data at the same time. Kafka is built to handle receiving data streams from external sources and it allows for concurrent data access by multiple consumers making it a good choice for TCPS.

Due to the large-scale storage requirements and the requirement to scale up and down based on data volume, it would seem that the data layer would be a perfect candidate to move to the cloud. To make this prospect even more enticing all three major commercial cloud providers offer their own managed Kafta solutions. AWS has Managed Streaming for Apache Kafka (Amazon MSK) [86], GCP has Confluent Cloud on GCP [92], and Azure has Apache Kafka in Azure HDInsight [26]. These manage services allow users to dynamically scale and rollout their Kafka solutions easily and efficiently. However, depending on the source of the data and the processing requirements utilizing the commercial cloud may not always be possible. If there are real-time processing constraints on the data the latency added by sending the data to the cloud may not meet the requirements. In this case it makes more sense to have the data layer closer to the data. However, that doesn't mean other layers couldn't be located in the cloud.

6.2.3.2 Batch and Streaming Layer: Data Processing Engines

Once the data has been ingested and stored by the data layer, the next step is to utilize the data to perform analytics to get the required information utilized by TCPS. In this section we will discuss two different general processing engines that TCPS utilizes for data processing: batch processing engines and stream processing engines. Each of these classes perform different types of analysis with different time constraints and are critical for the functioning of the TCPS.

The batch layer consists of multiple components to assist in storing and processing massive amounts of data that do not have any real-time processing restrictions. Although the storage functionality is similar to that of the RDBMS systems found in the LAMP architecture, in addition to storage the batch layer must also support data access via query languages along with the ability to support complex manipulation and computational tasks on the data. Due to the network overhead of having to move large amounts of data around for processing, splitting up the storage and computation portions of the batch layer is not advisable. Therefore a software ecosystem that can handle both the data storage and processing must be utilized within the batch layer. One of the most popular software ecosystems being utilized by the batch layer is the Hadoop ecosystem. The Hadoop ecosystem includes both the Hadoop Distributed File System (HDFS) [88] for large-scale storage and the Hadoop MapReduce framework [91] for scalable data processing.

The Hadoop MapReduce framework and HDFS integrate seamlessly to abstract away all the underlying computing mechanisms so that all the users need to worry about is the implementation of Map and Reduce phases of the data operation. HDFS was designed to automatically handle hardware failure, provide support for large datasets, and be portable amongst systems. The Hadoop MapReduce framework meanwhile takes care of supporting parallelization and the automatic recovery of Map tasks due to hardware or system failures. These features make the Hadoop framework one of the most popular systems for use within the batch layer of the Lambda architecture.

The batch layer is meant to process data that does not have any real-time processing constraints. Usually this involves performing trend analysis or other historical analysis utilizing large amounts of data. Not having any real-time processing constraints makes this layer a great candidate to be moved into the commercial cloud. The reasons for this are two-fold, first the commercial cloud allows users to dynamically scale their storage capacity depending upon the amount of data that they have collected. When dealing with historical data, this could be quite a large amount of data that will only grow over time making the scalability of the cloud a major advantage. Secondly, all three of the main commercial cloud providers offer a managed Hadoop ecosystem as a service. AWS offers Hadoop running on their managed Elastic Map Reduce (EMR) service [86], GCP offers Cloud Dataproc [30], and Azure offers HDInsight [78]. Each of these services helps user to take advantage of the cloud specific features offered by each provider and enables users to get started and setup quickly. Utilizing these managed services also can allow for the management of the infrastructure as code which we will discuss in a later section.

In contrast to the batch layer, the streaming layer is meant to handle small to medium data sizes and to provide output and results in real-time or near-real-time. The streaming layer does not have any persistent storage components but instead relies on streaming data processing infrastructure that can ingest and process the data in real-time. Stream processing has been around since the the 1960s [89], but there are a few modern frameworks that work to support the level of data volume and velocity required for big data. Among these frameworks are Spark [107], Flink [44], and Storm [94] which are the most well-known and widely adopted streaming frameworks. Amongst these three, Spark stands out as the most widely adopted framework for the streaming layer amongst both academia and industry. The Apache Spark framework has been developed to address the shortcomings of the Hadoop MapReduce framework [107]. However, it should be noted that Apache Spark actually still utilizes the MapReduce programming paradigm. This means that many of the of the core data operations in Spark are based upon mapping and reducing. Spark however is designed to hold all data is maintained in memory and reusable/accessible across different stages of interactive jobs. With the introduction of the resource manager:

YARN (Yet Another Resource Negotiator) [97] in Hadoop 2.0, dynamic Spark clusters can be deployed and executed on the same hardware infrastructure that also supports HDFS and Hadoop MapReduce. The architecture of a Spark environment running on top of a traditional Hadoop infrastructure is shown in Fig. 6.3.

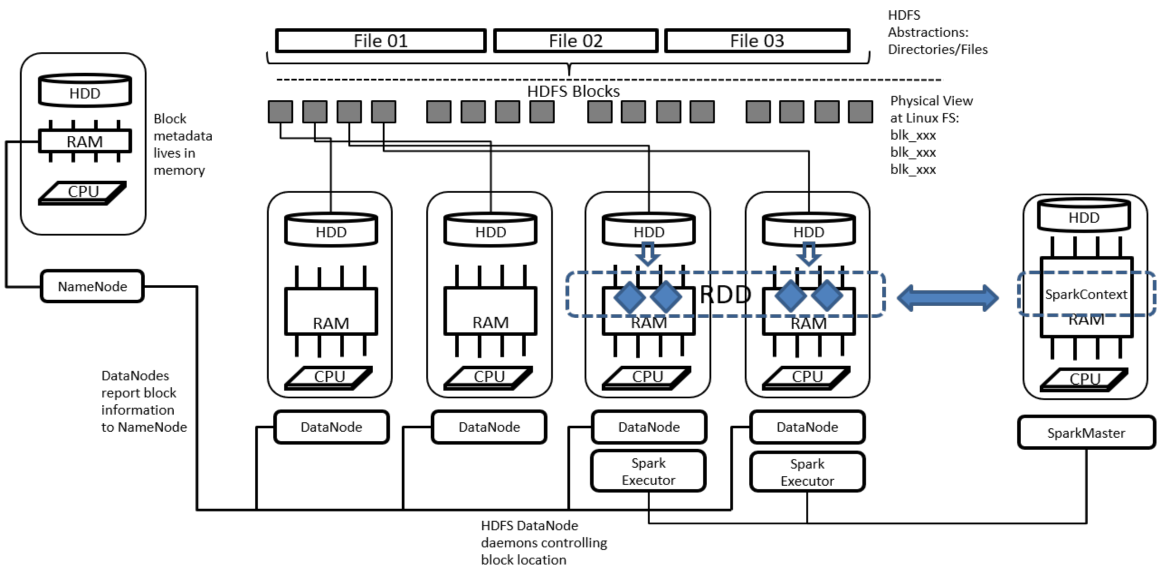

File 01
File 02
File 03
HDFS Abstractions: Directories/Files
HDFS Blocks
Physical View at Linux FS: blk_xxx blk_xxx blk_xxx
HDD
RAM
CPU
Block metadata lives in memory
NameNode
DataNodes report block information to NameNode
DataNode
Spark Executor
RDD
SparkContext
SparkMaster
HDFS DataNode daemons controlling block location

Figure 6.3: Spark deployment inside Hadoop infrastructure.

In Spark, the data is loaded into the system as resilient distributed datasets (RDD). This means that Spark will automatically read in the data files and convert them into read-only partitioned collection of records which by default one record is one line in the data files. Spark can support real-time streaming analytics by utilizing a StreamingContext that allows data to be continuously inserted into the Spark cluster's RDD environment over time. As Spark operates in memory space and reuses intermediate data, Spark can achieve a performance increase of up to two orders of magnitude over the traditional Hadoop MapReduce [107].

While the batch layer is a good candidate to move to the commercial cloud due to not having restrictions on latency and processing time, the candidacy of the streaming layer is a bit more muddled. There are good arguments to move the streaming layer to the cloud such as the scalability aspect, the ability to utilize the infrastructure as code principle, and the ability to utilize the same infrastructure for both the batch and streaming layers. However, there are also strong arguments against moving the streaming layer to the cloud. These arguments include the need real-time/near-real-time analytics and response, increased latency, and having to have duplicate infrastructure if the batch layer is located in the cloud. In the end the decision will be dependant on the characteristics of the data itself. If there is a high volume of data and very strict real-time processing requirements it may make sense to keep the streaming layer located locally and closer to the data sources. If this is the case it may be more cost-efficient to keep the batch layer local as well to eliminate the costs of additional infrastructure. However, if the streaming data has more relaxed time constraints the cloud can be a viable option especially if there are plans to scale up the infrastructure in the future.

6.2.3.3 Serving Layer

Within the Lambda architecture, the serving layer is where the end users can interact with the data and analytical results that are stored within the system. This interaction is accomplished through applications that can be written in different programming languages similar to the PHP layer in the LAMP architecture previously described. Many of these applications contained within this layer allow users to interact with them through a web browser, but there are additional applications that support direct interactions with the data at lower levels that are also included within this layer.

All of the tools that we have discussed in this chapter are based upon the Java programming language, although it should be noted that Spark's native language is Scala but it also runs within the Java Virtual Machine). However, there are additional Application Programming Interfaces (APIs) that can interact with the underlying frameworks utilizing various other programming languages such as Python and R. The way that these various applications interact with the underlying system depends on the layer that they are working at.

At the batch layer, the Hadoop MapReduce framework supports execution of both native Python and R codes that are driven by the map and reduce tasks. utilizing the HadoopStreaming library, Python and R programs can utilize Linux's standard I/O to read from HDFS's data blocks. There are also additional APIs and packages within Python and R that provide access to many more features of the Hadoop ecosystem. There are also database management systems that can be deployed as part of the batch layer. These database management systems include HBase [98], Cassandra [100], and MongoDB [9]. These database management systems are accessible via semantics that are similar to the Structured Query Language (SQL) which is the default syntax for all traditional relational database management systems. However, generally the semantics available for interfacing with the batch layer are not as expressive as those of traditional database management systems. Once the data has been transferred into a database management system, a number of third-party visualization software packages, such as Tableau [90] or PowerBI [59] can be utilized to help analyze and visualize the data produced by the batch layer.

Due to the batch layer favoring large-scale non-interactive jobs, many of the TCPS data tasks are often performed in the streaming layer. At the streaming layer the Spark framework is typically interfaced most often by the TCPS in order to provide end users with real-time/near-real-time insights. Although the native language of Spark is Scala, there are also up-to-date libraries that support most/all of Spark's features for the Java, Python, and R programming languages. This means that similarly to the batch layer, applications can be written in different languages depending on the analysis and data processing required and still take advantage of the features of Spark.

The migration of the serving layer to the cloud, as with the other layers of the Lambda Architecture, depends upon the location of the data and how the application is interacting with the data. If the serving layer is accessing large amounts of data to display then having the server and the data co-located would provide the most efficiency and least latency providing a better overall user experience. However, if the amount of data being exchanged is smaller, such as a JSON document with some result values, than having the serving layer located in the cloud is feasible. One benefit of having the serving layer be located in the cloud is that there are a large number of vendor provided solutions and images that are already pre-configured and ready to run out of the box on all the commercial clouds. This can limit the ramp-up and development time for the system administrators while allowing end users a wider variety of applications to analyze and view the data captured by the TCPS.

6.3 Infrastructure as Code (IaC)

The TCPS data infrastructure that is required to collect, store, distribute, and process the large volumes of data generated by TCPS can be quite complex. There are typically a large number of physical resources and software packages to be maintained and configured properly. Not to mention the tuning of the resources and infrastructure to ensure that it is working at maximum efficiency and effectiveness. Combine all of these factors and system administrators of TCPS systems face a very daunting and time-consuming task that may require many configuration changes, debugging, and architecture iterations to get things running perfectly. Just keeping track of all of the configuration and architecture iterations can be a full time job and if something is not documented properly the entire system can crash. Unfortunately, with the traditional method of creating infrastructure there is not a good way to keep track of these changes. However, the Infrastructure as Code (IaC) paradigm aims to change this by allowing users to create, modify, and remove infrastructure through code. IaC is an approach to utilizing cloud era technologies to dynamically build and manage infrastructure by treating infrastructure and the tools and services that manage the infrastructure itself as a software system and adapts software engineering practices to manage changes to the system in a structured and safe manner [62].

In addition to allowing better documentation of infrastructure and software changes, the IaC paradigm also enables the ability to iterate and change infrastructure more quickly and more efficiently. When utilizing the IaC paradigm, the infrastructure creation and configuration is all defined in code instead of being written down in user manuals or other documents. This means that any changes to the infrastructure or the software configuration can be tracked within standard software version control systems. This enables system administrators to ensure that the infrastructure is up to date and that all changes have been applied while also making it easier to debug the infrastructure if something is working properly as there is a well-defined history recorded in the version control system. This speed and agility is crucial for TCPS as the requirements and scale for the data storage and computation are rapidly changing and the infrastructure and system administrators need to be able to keep up. All three of the major cloud providers offer their own deployment engines that make utilizing the IaC paradigm easy to get started with.

6.3.1 Cloud IaC

As previously discussed the use of cloud computing has been increasing over the past few years due in part to its flexibility and scalability. By enabling users to scale their infrastructure depending upon an increase or decrease in demand users can save cost and create an overall better experience for their customers. It is this type of scaling that makes the cloud particularly useful and interesting in respect to TCPS as there are peak times where there is a lot of data processing to be done (ex: rush hour) and then off times where very little processing is required (ex: 2:00am). However, in order to take advantage of these scaling features some additional automation has to be performed during the processes of creating, provisioning, and removing the infrastructure to ensure the actions are performed correctly and in a consistent and trackable manner. This is where the IaC paradigm can be utilized to increase effectiveness and efficiency.

All three of the major cloud providers have their own IaC solutions that are designed to help tackle this exact problem within the cloud environment. These solutions are designed to help users create, modify, and delete their infrastructure within the cloud while maintaining a papertrail of the changes that have been made to the infrastructure. AWS has a service called *Cloud Formation* that allows users to use a simple JSON or YAML formatted file to model and provision all the infrastructure resources in a secure and automated manner. This file serves as the single source of truth for the user's cloud infrastructure making it easy to track and keep within standard version control systems [86]. A sample Cloud Formation Template (CFT) that launches a single EC2 instance is shown in Fig. 6.4

```
{
    "Description" : "Create an EC2 instance running the Amazon Linux 32 bit AMI.",
    "Parameters" : {
        "KeyPair" : {
            "Description" : "The EC2 Key Pair to allow SSH access to the instance",
            "Type" : "String"
        }
    },
    "Resources" : {
        "Ec2Instance" : {
            "Type" : "AWS::EC2::Instance",
            "Properties" : {
                "KeyName" : { "Ref" : "KeyPair" },
                "ImageId" : "ami-3b355a52"
            }
        }
    },
    "Outputs" : {
        "InstanceId" : {
            "Description" : "The InstanceId of the newly created EC2 instance",
            "Value" : {
                "Ref" : "Ec2Instance"
            }
        }
    },
    "AWSTemplateFormatVersion" : "2010-09-09"
}
```

Figure 6.4: Sample CloudFormation Template to launch an AWS EC2 Instance [86]

Microsoft Azure has its own IaC solution as well that is similar to AWS's CloudFormation concept called Azure Resource Management Templates. These templates are written in JSON formatted text files and define the infrastructure and configuration of the specified Azure infrastructure [58]. GCP also has a similar IaC solution available to users that is called the Cloud Deployment Manager that allows GCP infrastructure to be deployed utilizing a YAML or Python template file [30]. These tools are the cornerstones of a total IaC infrastructure in the cloud, but they are still just beginning to scratch the surface of the IaC paradigm.

As previously mentioned, utilizing IaC for TCPS has a number of advantages. We briefly mentioned the scalability of these solutions which can be extremely useful in TCPS.

With TCPS there are peak times, such as rush hour and during the holidays where additional processing power is required to handle all of the data that is being generated and sent to the system. By using IaC, system administrators can create policies that can have the infrastructure automatically create new infrastructure resources based upon the demand and then remove them when the demand decreases all without any intervention by the system administrator. These polices can be defined in code and therefore are easy to track and explain to other admins. This helps to minimize the number of configuration errors that can be caused by having a human in the process who can hit a wrong key or forget a critical step. This automation can also help ensure that servers are patched in a timely fashion which can minimize the attack surface for TCPS.

Another advantage offered by IaC is the ability to iterate and change infrastructure much quicker than utilizing traditional methods. By enabling the tracking of changes to the infrastructure through traditional version control systems, it is easier to ensure that the proper changes have been applied to the infrastructure. This also makes it easier to debug the infrastructure and even possibly rollback any changes that have unintended consequences as the previous configuration is located in the version control system. This speed and agility is crucial for TCPS as the requirements are always changing and the amount of downtime must be minimized in order to ensure the saftey of the end users.

6.3.2 Internet of Things (IoT) IaC

Along with having the ability to have the lower level compute infrastructure as code, the cloud also allows for Internet of Things (IoT) devices and sensor infrastructure to be created, modified, and removed through code. Although similar in concept, this process is a little different than the compute infrastructure process as this process does not involve deploying hardware resources, such as sensors, but rather refers to the ability to manage and update the infrastructure. One such cloud based solution for this type of process is Amazon Greengrass. Greengrass is a software that enables users to run local compute, messaging, data caching, sync, and ML inference capabilities on connected devices in a secure way. By utilizing familiar languages and programming models, users can create and test device software in the cloud and then deploy it directly to the physical devices [86]. This enables quick and efficient code and device prototyping which is critical within TCPS as there are always new devices and sensors that need to be integrated into the infrastructure. Greengrass also helps with helping to keep track of any changes to the software running on the sensors and devices deployed out in the field. These types of IaC services are still in their infancy but they are already starting to cause a shift in the traditional way of thinking how both datacenters and field devices are deployed. As the technology matures these types of solutions will become more prevalent as users begin to realize the power that they have and the additional benefits they provide.

6.4 Conclusions

In this chapter we have discussed the components and architectures associated with TCPS. We discussed in detail the four different layers of the Lambda architecture: data, batch, streaming, and serving layers. For each one we discussed the available software framework and infrastructure requirements along with the arguments for and against migrating the layers to the cloud. Our analysis shows that although it is possible to migrate all the layers to the cloud this may not be feasible as the data volume or real-time/near-real-time data processing requirements may be unable to be met. We also discussed the cloud native solutions for each layer and how they could be integrated into the layers to increase performance within each commercial cloud.

We note that of all the layers within Lambda architecture for TCPS, the batch layer is the most promising layer to migrate to the commercial cloud as there are no real-time processing constraints and it is designed for non-interactive jobs. However as the batch and streaming layers can execute on the same hardware infrastructure if the streaming layer cannot be moved to cloud the user may end up with duplicate infrastructure. This increases the operating cost and may not be an ideal solution.

Lastly we discussed the Infrastructure as Code (IaC) paradigm and how it can provide better change tracking and easier maintenance for TCPS infrastructure. By beginning to migrate and utilize the IaC paradigm the complex TCPS infrastructure can be tracked by traditional version control systems making documentation, maintenance, and changes to the infrastructure simpler and more efficient. This allows TCPS to be more flexible as the infrastructure can be upgraded and rolled back faster as the changes are well documented in a version control system. The IaC paradigm can also help to assist TCPS in being more efficient and effective by allowing the systems to automatically adapt to changes in demand by dynamically creating and removing resources. With the IaC paradigm only in its infancy, the power and scale of the available IaC tools will only get better with time. When looking to migrate portions of TCPS infrastructure to the cloud, IaC should be utilized as much as possible to ensure the reliability and flexibility of the system.

Chapter 7

On-Demand Urgent High Performance Computing Utilizing the Google Cloud Platform

We have previously presented our work on identifying how the commercial cloud could help provide additional architecture options and capabilities for data intensive workflows found within Transportation Cyber Physical Systems. We now showcase the large scale execution of a data intensive workload on the Google Cloud Platform. This workload utilizes the traffic monitoring software TrafficVision to process 211TB of video with 1.5 million vCPUs. The execution spanned 6,227,593 core hours and was executed from initial infrastructure provisioning to de-provisioning in a span of just 8 hours. By being able to provision infrastructure through code quickly and efficiently, these types of environments can be created on-demand to help prepare for an incoming disaster such as a hurricane.

7.1 Introduction

Evacuations of major cities before an impending natural disaster such as a hurricane can be complicated and even deadly. Although evacuation warnings are usually put out well in advanced, many residents do not evacuate until the last minute. This can cause severe gridlock and leave many motorists stranded due to running out of gasoline. Combined with excessive heat or flash flooding this situation can lead to a large number of fatalities even before the natural disaster hits. Such a situation occurred in 2005 in Houston, TX where more than 100 people died during the evacuation from Hurricane Rita. In our research we demonstrate how high performance computing (HPC) in the commercial cloud can be utilized in these types of situations to help aid with the disaster management efforts.

Utilizing HPC on the commercial cloud, first responders and disaster management coordinators can process large amounts of traffic data efficiently and on demand which can aid them make quicker decisions and monitor the situations in real-time. Our solution uses Google Cloud Platform (GCP) to launch a massive parallel processing environment that is the size of a Top 5 supercomputer [1]. Our approach to utilizing the commercial cloud for urgent computation has wide applicability. For example, many cities in the southeast region of the US and Texas are regularly evacuated during hurricane seasons.

One of the main hurdles to efficient emergency evacuations is traffic management. Traffic can begin to back up days before a hurricane is scheduled to make landfall which can drastically limit the number of people that can be moved in a timely manner. By utilizing the commercial cloud, emergency planners can dynamically and efficiently create a temporary large scale environment to help evaluate when and how to begin evacuations. By recording and capturing the traffic data during a hurricane, emergency planners can run models to evaluate certain what-if scenarios as the projected path of the oncoming hurricane changes.

The amount of data required to manage a region as the projected hurricane path changes is quite large. An evacuation that includes evacuation, clean up, and return taking up to 10 days and utilizing 8500 cameras across an evacuation region generates around 2M hours of recorded video and requires 2M hours of compute time to process. Along with this processing time, the same algorithms utilized to monitor emergency evacuations and give real time status can also utilize models built from previous evacuation studies which adds additional computational complexity and time. We utilize GCP to process 211TB of video in 2,005,170 compute hours. In our execution, each process reads its own input file and writes its own output data which causes a large strain to the storage, network, compute, HPC scheduling subsystems, and the cloud management infrastructure.

7.2 User Application Requirements

For this research, there were a number of different requirements that we take into consideration. The characteristics of the traffic monitoring application constrained the available options and form the basis for the workflow specifications. Here we will define the traffic monitoring application and its requirements and the considerations that went into our decisions.

7.2.1 Definition

The monitoring of different evacuation routes for accidents and tracking the movement of vehicles along the route requires a specialized software. This software must interface with video recorded from existing traditional cameras that have already been placed along different public evacuation routes. The application that we utilize for our execution is a commercial traffic analytics software, TrafficVision [95, 43, 41, 42]. The TrafficVision software package provides incident and anomaly detection in traffic patterns from existing highway camera infrastructure. The low-resolution of cameras preserves the privacy of the motorists while still providing the level of detail required for the monitoring of the overall traffic flow. TrafficVision's AutoLearn feature helps to handle the size and scale of this project as it helps to handle the real-world environmental/operational factors such as camera motion, varying light levels, or vision compression artifacts. TrafficVision also offers the flexibility to process real time video streams as well as batch processing of pre-recorded video data.

The processing of the video streams is an embarrassingly parallel task meaning that the processing can be scaled up to an extreme scale while still being able to process the clips simultaneously. In our execution, each vCPU or core can process a single 15FPS video stream, perform the required detection and issue alerts to the user. TrafficVision is a CPU bound application that does not have a large memory footprint. This is a key feature for scaling the application as our VMs can have a large CPU to memory ratio allowing us to select technologies that will save costs during execution.

7.2.2 Requirements

The user requirements that guide our design and selection of tools are the ability to execute at a large scale and to control the costs. Controlling costs is an important part of any experiment, however when working at the large scale that we are it becomes even more important. The two major decisions that we have to make to execute our TrafficVision workflow are what commercial cloud and what type of instance (VM) to utilize.

The features and aspects of the different commercial cloud providers are changing rapidly and as such there are always new features and services being released. At the time of this study there were three main commercial cloud providers that could handle such a large scale execution: Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure. Each provider offers a wide range of specialized services that can be utilized by end users. All of these providers offer a set of "core" services such as compute, database, and object storage but differentiate themselves on the additional services that are built on top these "core" services. For example, all three commercial cloud providers give users a list of predefined instance types for them to utilize. An instance type is simply a combination of CPU, memory, and GPU. Within AWS and Azure these instance types are defined by the provider and cannot be customized by the end user [86, 58]. However, GCP allows users to specify a custom instance type that can be tailored to fit a specific workload helping to keep costs down [30]. GCP custom instance types allow users to specify the number of vCPUs and the amount of RAM allocated with their instances. This aspect is useful for our TrafficVision workload as it does not have a large memory footprint and therefore we can allocate an instance with a minimum amount of RAM and a large number

Table 7.1: Comparison of "Spare Capacity" Instance Functionality as of August 23, 2019

	AWS	**GCP**	**Azure**
Reference	[86]	[30]	[58]
Unlimited Run Time	✓	-	✓
Fixed Discount	-	✓	✓
User Bidding	✓	-	-
Available Within Standard Compute Service	✓	✓	-
Custom Instance Types	-	✓	-
Maximum Discount	90%	80%	80%

of vCPUs.

All three cloud providers also allow users to access their "spare" capacity in the form of a specialized instance type. These specialized instance types provide users with discounts off the regular instance pricing with the caveat that they can be shutdown with only a short notice. A summary of each commercial cloud provider's "spare" capacity instance types can be found in Table 7.1.

For our execution, we choose to utilize GCP as the capabilities and pricing are a good match for our TrafficVision based workflow. By utilizing the custom instance types that GCP offered we are able to provision instances with a large number of vCPUs and a minimum amount of RAM for the TrafficVision software to run effectively. Combined with the fixed preemptible pricing and discounts, the custom instance types allow us to have more control over the costs by having them be more stable and predictable during the execution of the workflow.

While preemptible instances will work well for the application instances, this is not the case for the rest of the supporting HPC environment. The HPC scheduler, Login instance, Control instance, and NAT instance all need to be available all the time as without these instances the compute instances will not be able to get new jobs. For these instances we utilized a standard on-demand instance types to ensure availability during the execution of our workflow. In addition to the different instance pricing, utilizing a custom instance type will also help further manage our costs while also ensuring that the entire resource is being utilized and we are not wasting CPU cycles or having unused RAM sitting idle.

7.3 System Design Considerations

After the initial determination of which commercial cloud and which instance types to utilize, we are able to start to designing the system and the way to execute the workflow the most efficiently. Here we discuss why we selected the technologies that we did for our execution.

7.3.1 HPC Lifecycle Technology Selection

There are several software infrastructure components that are required for the management of a large scale HPC environment and to keep the jobs running smoothly within it. We evaluate two different software options for this component: an off-the-shelf solution provided by GCP as well as our previous work the Provisioning And Workflow Management Tool (PAW). In our previous work we discuss in detail the alternative resources and workflow management tools for the cloud and developed a comprehensive tool to accomplish both tasks: PAW [75].

The off-the-shelf GCP tool was recently published by GCP and provides a standard HPC deployment solution that was developed in collaboration with SchedMD. This HPC deployment provisions a traditional looking HPC environment that utilizes the Slurm HPC scheduler [51]. This traditional looking HPC environment consists of a Login instance, an shared NFS filesystem, Scheduler instance, and Compute instances within either an existing or new Virtual Private Cloud (VPC) network. Many of the previously mentioned GCP features such as custom instances and preemptible instances are supported by this deployment. This solution also allows users to provision their environments from Google-provided disk images which can significantly help speed up the launching process of the environment as the user can pre-configure and install all the required packages for their workflow before launching the environment. Utilizing these images, the solution is able to launch a new environment of 5000 instances within 7 minutes [51].

As with any solution however, amongst all the positives there are some drawbacks.

One of these drawbacks is that users are only allowed to provision a single instance type for their Slurm environment. This leads users to over-provision their environments as they have to cater to the most resource intensive workloads. This can result in excess costs if the created resources are not properly utilized for all workload execution. Another drawback to this solution is that it is a vendor specific solution and the configuration does not easily transfer to another cloud provider. For the area of urgent HPC, this can be a major limitation as the computation needs to be able to be completed utilizing all of the resources possible regardless of which cloud provider the resources are located in. Without interoperability between clouds, users with a need for urgent HPC could be left without any means of computing if the cloud provider that they rely on suffers an outage when the processing needs to be completed. Also, although this solution does provide an HPC scheduler it does not provide any means of workflow management as that is left up to the user to decide how to implement. This means that users are expected to manage and submit their jobs either with a customized script, third party workflow management tool, or manually. While this tends to work in smaller HPC environments, submitting and tracking hundreds of thousands of jobs can be frustrating and difficult.

PAW on the other hand is designed to be cloud provider agnostic and manage both the resources and the workflows in the same interface. PAW automates the dynamic resource provisioning for the cluster, executing user defined workflows, and then de-provisioning the cluster when the workflows have completed. In addition, PAW is built to automate all of the key management tasks that are typically the pain points when managing these large scale environments with a single command. PAW is not built to replace traditional workflow management tools, but rather to compliment and work in conjunction with them. PAW does not attempt to replicate the functionality of traditional off-the-shelf workflow management tools such as SWIFT [104], Tigres [35], and Pegasus [20] but instead provides an interface for users to create a custom workflow that can feed information to and from these tools.

The current iteration of PAW utilizes CloudyCluster [73, 68] APIs and included metascheduler CCQ [67] to perform the key management tasks for the HPC environment.

This provides PAW with the ability to execute the same workflows across multiple commercial clouds through the same interface. PAW has also already been tested at a massive scale. In our previous work, we utilized PAW to create a 1.1M vCPU HPC cluster on AWS to perform topic modeling research [72]. By utilizing a solution that has already been tested at a massive scale on another commercial cloud provider gives us the additional benefit of unlocking more resources for use in urgent computing scenarios and limits the amount of additional testing stress testing required.

For this experiment, we chose to update and extend PAW to accommodate our application's requirements as we were already familiar with the solution and it provided us with the most flexibility.

7.3.2 System Design

The design of our virtual HPC environment resembles that of a traditional HPC environment, however instead of the resources being provisioned ahead of time they are provisioned on demand and when the workflow is submitted to PAW. This deployment is done with the specification of a single PAW configuration file that contains all the specifications for both the HPC environment and the workflow to be processed. More detail about the composition of this configuration file and the provisioning/de-provisioning process is describe in [75]. Once this configuration file is submitted to PAW the creation of the resources begins. The first stage of this process creates what is referred to as a "base environment" which simply contains the minimum resources that are required for the HPC environment to operate. For most workflows these minimum resources include a Login, NAT, and Scheduler instance but depending on the workflow some of these may not be required.

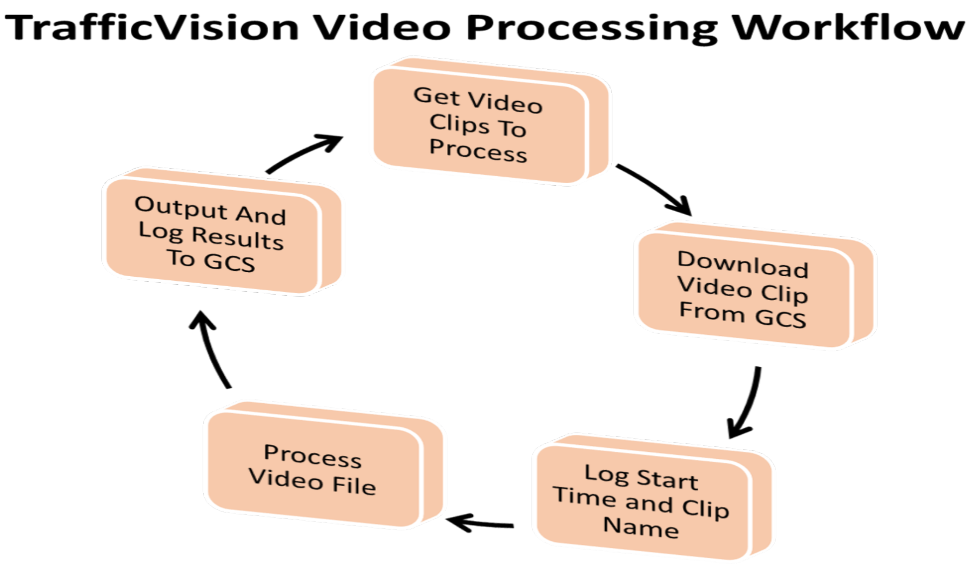

TrafficVision Video Processing Workflow
Get Video Clips To Process
Download Video Clip From GCS
Log Start Time and Clip Name
Process Video File
Output And Log Results To GCS

Figure 7.1: Overview of the TrafficVision application workflow.

For our experiment, our "base environment" consists of a Login instance, NAT, and HPC scheduler. Our TrafficVision workflow does not require a shared filesystem as we choose to utilize Google Cloud Storage (GCS) for both downloading our input files as well as writing out the output files after processing. Fig 7.1 illustrates the TrafficVision application workflow.

In order to achieve the processing scale required to manage vehicle evacuation at hurricane proportions, we utilize multiple of these "base environments" that, in aggregate, provide a massive scale parallel processing environment. There are a number reasons for utilizing multiple "base environments". One reason is that the utilization of multiple environments prevents a single point of failure within a single environment from causing the entire application and workflow to fail catastrophically. For example, if the HPC scheduler were to fail during the execution then any submitted jobs would fail to execute and we would be paying for compute power that we are not utilizing. By creating multiple smaller environments, if one environment goes down, the other environments can continue processing. Another reason for having multiple smaller environments is that these environments can be spread across different geographical regions to handle the processing. This ensures that if a particular cloud region fails that the other regions can continue computation. Although it may seem far-fetched that an entire processing region goes down, it is very much a possibility in the face of an impending natural disaster such as a hurricane.

Once these “base environments” are in place, PAW submits the user-defined workflow to the execution for processing. During this phase, PAW performs any actions defined by the configuration file and then submits the corresponding jobs to the HPC scheduler just like a traditional job would be submitted. PAW then monitors the scheduler queue for job completion and when there are no more jobs executing, PAW de-provisions the environment so that the user is no longer charged for the resources. This deletion is optional and not required if the user wants to maintain the data on a shared filesystem.

We have two options when designing the application in regards to accessing the traffic data required. The first option is to receive the live video feeds from a number of cameras to process the data in real time to showcase the ability of the system. However, as ordinary citizens we do not have access to the full scale of cameras or the proper network access for the live video feeds that may be utilized in a real hurricane scenario. The second option is to record a number of publicly available video feeds for a few weeks and store this data in Google Cloud Storage (GCS). This ensures that we have access to all the data that we need during the execution of the experiment and that we can emulate the size of a real life hurricane scenario and would not get blocked by trying to access a large number of camera feeds simultaneously. While we performed the experiment on recorded video, switching to live video requires just a simple changing of arguments within our script.

The workflow is implemented as a PAW user-defined workflow which simply generated a batch script for PAW to submit to the HPC scheduler. This batch script creates a simple work queue on each of the compute instances within the HPC environment. The required pre-recorded video clip is then copied from GCS to local storage for processing and each vidoe clip is processed as a single “job” on each instance. When that “job” has been processed the next clip is pulled from GCS and the process begins again. Our clips are formatted into one hour chunks and each clip is processed as though it is in real-time and as such takes about an hour to process. In the case of switching to real-time video instead of recorded a URL pointing to the video stream would be utilized instead of a path to a clip on GCS and the same video stream would be processed by a single vCPU until some stopping criteria is reached.

7.4 Implementation and Scalability Evaluation

Overall we did not experience a large number of unexpected challenges as in our previous work we have already identified a number of challenges and made sure to avoid the same pitfalls during this experiment. However, as expected we did find and identify two challenges that still surprised us during the execution. One of these challenges was an API rate limit per GCP Project while the second challenge was the very rapid provisioning of instances on GCP that created some additional complications with the infrastructure provisioning software.

We identified and resolved these issues utilzing a testing plan that allowed us to evaluate our workflow and design at different scales ranging from around 100 instances and 15,000 vCPUs to 5,000 instances and 80,000 vCPUs per environment. This medium scale limit tests the scalability of the HPC scheduler and underlying infrastructure provisioning software. Achieving success with a single 5,000 instance environment allowed us to move forward with the execution of multiple 5,000 instance environments that were utilized in our final run.

7.4.1 API Limitations Per Project

All commercial cloud providers have certain limitations placed upon a user's account to ensure the availability of the service for everyone and that a single person does not abuse the system. The first challenge that we encountered during the execution was a limitation on the number of certain API calls within a single GCP Project. A GCP Project can be thought of as an overarching "container" for all of a user's resources. Resources within a single GCP Project can be launched within any GCP regions as one of the features of GCP is that their VPC networks are global and can contain resources running in multiple regions. This is in contrast to both AWS and Azure where resources within a VPC are limited to a single region. We originally thought that we could utilize this feature to launch multiple environments in different regions utilizing the same GCP Project. However this turned out not to be the case.

Within a GCP Project there are a number of quotas (limits) on how many resources and API calls that a user can make within a certain time period within that project. Some of the quotas are regional while others are global and apply to all resources and regions within the same project. This is where we encountered our challenge. When we started to spin up multiple environments within a single GCP Project we found that we were hitting a number of GCP Compute Engine quotas with only one or two environments. The quotas that we were hitting included: *List requests per 100 seconds*, *Read requests per 100 seconds*, and *Heavy-weight read requests per 100 seconds*. These are the API calls that we are utilizing to create, monitor, and delete instances during the execution and upon scaling up we quickly hit these quotas which cause throttling from GCP and not allow us to get to our target threshold.

We identify a two-fold solution to this challenge. First we attempt the easy way: submit a request to have these quota limits increased to allow for more API calls to be made. We were granted an increase from 2,000 to 6,000 API calls per 100 seconds for all of these quotas. However we still needed to launch 93,750 instances in order to get the scale we required and by only allowing for 6,000 API calls per 100 seconds it would take 26 minutes of just launching instances to get us to our target. This does not even take into account any additional API calls that may be made during that 26 minute period for monitoring of the instances that had already been launched. Since these limits are global for the entire project, they would negatively impact all of our environments even if they were in a different region.

The second part of the solution is to move to a multiple GCP Project setup where we have a single GCP project for each region that we want to utilize during execution. This way we can spread the API calls across multiple projects and increase the quotas for each one. An additional benefit of moving to the multiple projects is that we are able to launch more environments simultaneously instead of having to wait a period for the quotas to clear.

During the execution we attempted to launch environments located in different regions within a single project. However, we find that by doing this we are limiting the number of environments that we can run simultaneously. Since we are utilizing preemptible instances which utilize GCP's spare capacity, the availability can vary greatly between the different GCP zones and regions. This variability means that we have to be flexible as to when and where we can launch instances as if we depend on just a single zone/region it may run out of capacity and we could not meet our target. By attempting to launch environments in different regions from the same project we find that our flexibility to utilize the available capacity is limited as sometimes the most capacity is be in a region that we set up in a project that has already maxed out the API quota. To fix this issue, we end up limiting to a single region per GCP project to ensure that we can always take advantage of available capacity.

Another similar API quota issue that we encountered was getting throttled when we attempted to shut down the instances after execution. As GCP is a "pay-as-you-go" cloud provider meaning that as soon as the instances are no longer doing work the instances should be shutdown as soon as possible to avoid extra charges. However, we find that we were still maxing out the API calls because we are attempting to delete all the instances at the same time. We do not want to have wait for the 100 second period for the quotas to clear as we do not want to pay for the instances during that time. Our solution to this is to utilize the pre-existing Salt master-minion setup within CloudyCluster to issue the "shutdown" command to all the instances within the environment. This will put the instances into a "Stopped" state within GCP where we are no longer paying for the run time, just the storage costs. This reduces the cost for the instances while we wait for the delete calls to cycle through.

7.4.2 Rapid Provisioning of Instances

The second unexpected challenge dealt with the rapid provisioning of instances by GCP. We also thought that this would be a feature that would help us get to processing faster, however that turns out not to be the case. This works at smaller scales, but when moving to a larger number of instances the large number of instances launching at the same time can cause issues. When the compute instances come up for the first time, they attempt to communicate with the HPC scheduler to register and start obtaining jobs. However if there is a large number of instances coming up at the same time and they all attempt to communicate with the HPC scheduler at the same time it starts to look like a denial of service attack on the HPC scheduler.

We encountered this issue when moving up to the 5,000 instance test. Things worked perfectly as the first few instances came up but as more and more instances were launched the scheduler quickly got less responsive until it crashed. This was not something that we expected and it took us a while to figure out that the rapid provisioning of instances was the cause. Once we identified the issue, we developed a solution to reduce the number of instances launched at a time by utilizing the GCP Batch request API to stagger the instance launches. This allows us to launch instances in more manageable batches with randomized amounts of time in-between. Although this did help to shrink the problem, it did not solve it.

We find that although the instances are registering with the scheduler, it is still taking a large amount of time for them to begin work. We find out that this is due to the configuration of Salt and the Slurm HPC scheduler within the CloudyCluster provisioning software utilized by PAW. Although we mitigated the denial of service attack on the scheduler, both Salt and Slurm were having issues trying to add and authenticate the larger number of instances. To fix this we look into two solutions. The first one is provided by the GCP SchedMD collaboration and involves putting the Slurm configuration file on a shared filesystem that all the instances could access. This eliminates the need to push the file to each instance as was the case with the CloudyCluster software it is not really scalable as there are known limitations to the scalability of shared filesystems [4]. This also adds complexity to our environment as we now need to create and maintain a shared filesytem.

The second solution involves modifying the actual configurations of Slurm and Salt to limit the number of instances that can authenticate at a time. This approach does not yield the same drastic drop in registration times as the first solution as the configuration file still needs to be pushed, however it does keep the underlying architecture scalable and eliminates the need for a shared filesytem. By performing some minor configuration edits, we are able to drop the instance registration time from 40 minutes to 20 minutes for all instances to be registered and computing.

7.5 Integration and System Evaluation

We executed our workflow across a set of HPC environments that were launched within different regions and GCP projects. We utilized a total of 4 different GCP projects and 6 different GCP regions for the execution of our TrafficVision workflow. Each GCP project was based in a different region depending upon which regions had the most spare capacity at the time of execution. We worked with our Google colleagues for guidence about which regions to utilize and ended up utilizing: *us-central1*, *europe-west4*, *us-east1*, *asia-east1*, *us-west1*, and *europe-west1*.

Our workflow utilizes a single custom GCP instance type: *custom-16-16384* which has 16vCPUs and 16GB of RAM. This custom instance type allows us to pay for the minimum amount of RAM which helps keep costs down significantly. This instance size is also chosen due to the fact that smaller to medium instance types are less likely to be preempted due to their ability to "fit" into more slots. The more resources that a preemptible instance requests, the more likely it is to get preempted. In order to reach our proposed goal, we need to launch a minimum of 93,750 instances so we wanted our instances

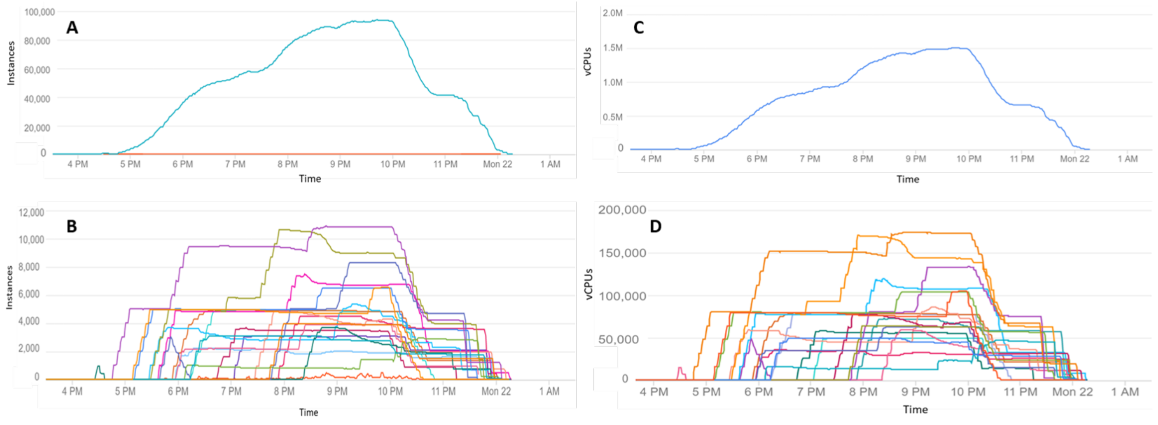


Figure 7.2: The overall view of workflow execution. In graphs **B** and **D** each line represents a single GCP Zone, names are not shown for space reasons. **A)** A timeline of the total number of instances in all regions/zones launched during workflow execution. **B)** A timeline of the total number of instances launched per Zone during workflow execution. **C)** A timeline of the total number of vCPUs launched in all regions/zones during workflow execution. **D)** A timeline of the total number of vCPUs per Zone during workflow execution.

to be able to "fit" in as many places as possible.

Fig. 7.2 showcases the ramp summary of the total number of instances and vCPUs. During the execution of the workflow, at our peak around 9:30 PM EST we had 93,905 instances running across 30 HPC environments totaling 1,502,480 vCPUs executing concurrently. A breakdown of the instance distribution by GCP region and zone is shown in Table 7.2. Although we launched almost 100,000 instances, our overall preemption rate during the execution remained relatively stable and low throughout with the highest peaks being 300 instances which is less than 1% of our total instances. A majority of the instances remained running until we shut them down at the end of the workflow. The overall preemption rate is shown in Fig. 7.3.

We did encounter some API throttling during the experiment. The effects of this can be seen in the graphs in Fig. 7.2 around the 6:00-7:30pm mark during creation along with the 10:00-11:00pm mark during deletion. During these periods the number of instances that were being launched or deleted slowed down or stopped. During these periods we created more HPC environments and utilized other projects to limit the number of API calls being

Table 7.2: Breakdown of the number of instances provisioned per GCP Region/Zone at peak

Region	Zone				
	A	B	C	D	F
us-central1	2,661	3,464	4,155	-	4,904
us-east1	-	6,708	1,915	3,066	-
us-west1	4,853	4,846	4,177	-	-
europe-west1	-	10,803	1,462	8,898	-
europe-west4	6,477	8,324	3,917	-	-
asia-east1	6,509	2,780	3,904	-	-

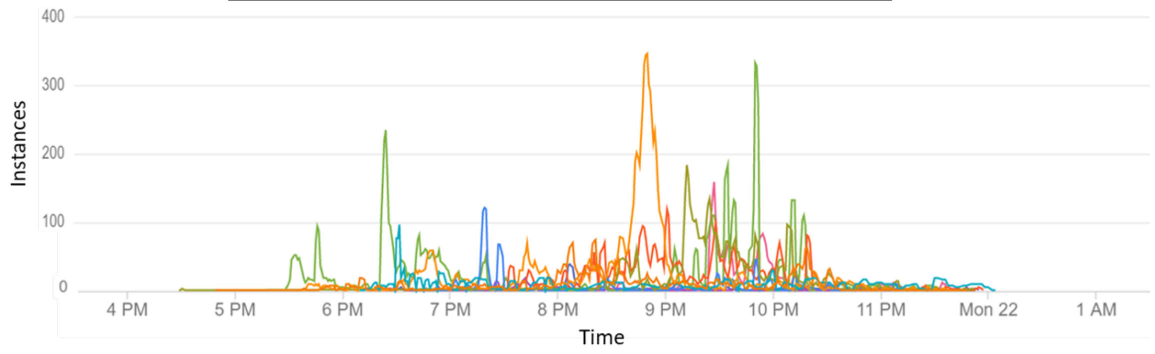


Figure 7.3: A timeline of the rate of instance preemption throughout the execution of the workload. Each line represents a single GCP Zone, names are not shown for space reasons.

used in each project.

We also note the performance of GCS during our experiment. To increase the throughput to GCS, we implement the concept of GCP *Private Routes* which allows GCS traffic to bypass the NAT instance eliminating a potential bottleneck. By utilizing this solution, we did not hit any throttling or performance issues with accessing the pre-recorded video clips or uploading the results. We hit a maximum rate of 52GiB/s read and 768MiB/s write to GCS during our execution. The overall throughput rates to GCS are shown in Fig. 7.4 A and B.

Over the 8 hours that our workflow executed we processed 2,006,170 hours (211TB) of video. The processing rate of completion is shown in Fig. 7.5. We note that there are not a large number of completions in the first hour because our pre-recorded video is segmented into one hour chunks and the results are not uploaded until the full clip has been processed. When running in real-time with a standard video stream the results would be uploaded in

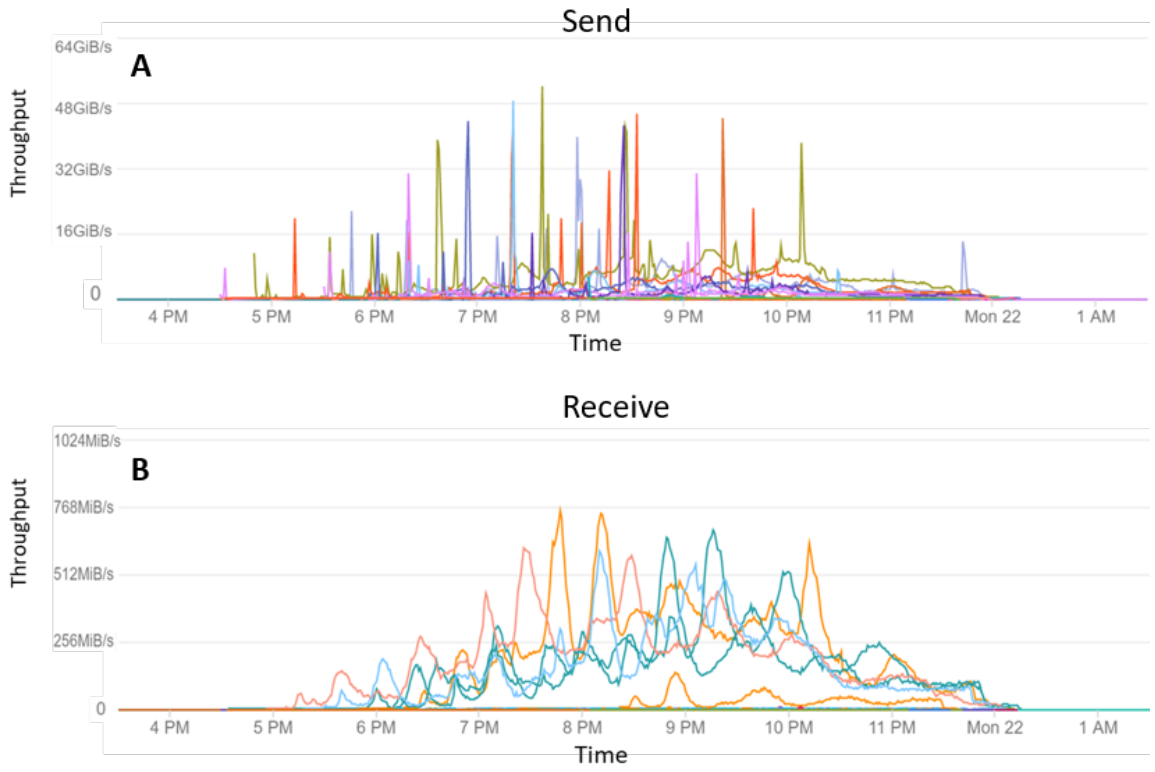


Figure 7.4: The overall view of GCS performance throughout workflow execution. In both graphs each line represents a single GCS Bucket, the names are not shown for space reasons. **A)** A timeline of the throughput of data sent to the running instances during the workflow execution. **B)** A timeline of the throughput of the data sent back to GCS after being processed by our workflow.

real time.

Another important evaluation factor for our workflow is the overall cost. By utilizing our custom machine type, we are able to help minimize our costs. At the time of the experiment a standard instance type comes with 16 vCPUs and 64GB RAM and costs $0.1600 USD per hour with preemptible pricing while our custom instance type costs $0.1264 USD per hour. This gives us a cost savings of $0.0336 USD per instance per hour which when multiplied with the roughly 94,000 instances required provides a cost savings of $3,158.40 USD per hour. This cost savings can be significant especially if the workflow needs to execute for an extended period of time.

We executed the workflow for approximately eight hours and utilized a total of

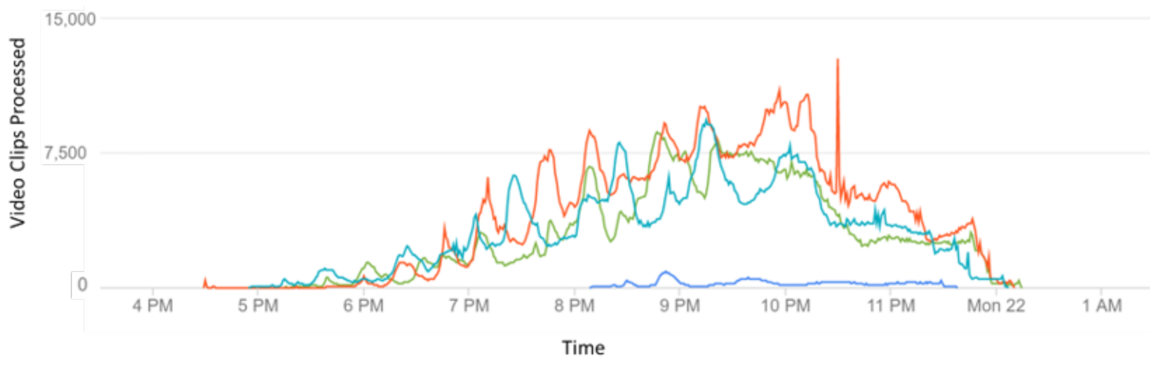


Figure 7.5: A timeline view of the video clips analyzed during the execution of our workflow. Each line represents a different GCP Region, the names are not shown for space reasons.

6,227,593 core hours with a total cost of $55,044.95. This gives us an overall cost per core hour of approximately $0.008 USD which aligns well with the cost estimation for local resources which is typically between $0.02-$0.01 USD per core hour.

7.6 Conclusions

We present a proof of concept that commercial clouds can handle urgent HPC processing of massive data by provisioning, utilizing, and de-provisioning an HPC cluster with more than 1.5M vCPUs in about 8 hours. Our application ran on top of the Google Cloud Platform and demonstrates how urgent HPC can assist with evacuations in the event of an impending disaster. We have discussed how the different features offered by each commercial cloud provider can provide users with a cost-effective on-demand infrastructure for urgent HPC and computing in general.

We utilize the cloud-agnostic Automated Provisioning And Workflow Management tool (PAW) to build the HPC environments utilized to execute our worflow. Utilizing PAW and the underlying CloudyCluster APIs we were able to analyze 2,006,170 hours (211TB) of video with an average cost of $0.008 per vCPU hour. We also discuss the challenges that we ran into when executing this workflow and provided solutions for each of them.

The execution of this massive scale on-demand HPC environment across multiple geographic regions showcases the flexibility and redundancy offered by the commercial cloud along with providing a starting point for others to get started building their own workflows.

Chapter 8

Summary

8.1 Summary of Contributions

In this dissertation we have examined how the architecture of different commercial clouds can affect the performance and viability of different HPC workloads, explored how to better enable commercial cloud access to academic researchers, and pushed the scalability limits of the commercial cloud. In this chapter we provide a summary of the contributions of this dissertation.

8.1.1 HPC Workload Performance Evaluation

First, we evaluate the viability of executing certain types of HPC workloads within the commercial cloud. We execute the benchmarks found in the HPC Challenge benchmark suite on different configurations within the commercial cloud to determine which types of workloads may be best suited for execution on the commercial cloud. The benchmarks executed include HPL, DGEMM, STREAM, PTRANS, RandomAccess, FFT, and Communication bandwidth and latency. Each of these benchmarks stresses a different part of the cloud resources and represents a different class of workload. We find that although the results of the benchmark are not as good as they are on similar local hardware, there are certain classes that still perform at an acceptable level. These benchmarks were those that were CPU intensive and did not require a lot of network communication in order to execute. However, we also find that although the performance may not be as high, the “on-demand” nature of the resources can outweigh the wait time required for traditional on-premise resources.

8.1.2 Dynamic Resource and Workflow Management

Secondly in this dissertation, we find that although there are a number of tools that exist to either provision commercial cloud resources or handle the execution of user workflows, there are very few tools that exist that can handle both tasks. This can be a major limitation to researcher adoption as they have to learn an additional tool to manage the resources required for their workflow and integrate it with their existing workflow manually. To help address this issue, we developed the Automated Provisioning And Workflow Management Tool (PAW) for parallel scientific workflows.

PAW provides a modular base that can be extended to multiple commercial clouds and other types of resource providers in the future. The initial implementation of PAW is built utilizing AWS and CloudyCluster to help allow researchers with minimal commercial cloud knowledge to take advantage of some of the new technologies and scale the cloud can offer. Through PAW, researchers can execute custom defined parallel scientific workloads within AWS just as they would on traditional HPC clusters by utilizing a user-defined PAW workflow. A PAW created environment also allows researchers to have exclusive access to the created cloud based HPC environment which eliminates contention for resources.

We showcase the scalability of PAW by executing a custom user-defined PAW workflow to execute a large scale topic modeling experiment. This experiment studied how the variations in different variables affected the topic model output. Utilizing the CloudyCluster and AWS version of PAW we executed a workflow that utilized 28,000 vCPUs over 5,000 instances in only 25 minutes.

8.1.3 Considerations When Executing At Scale In The Commercial Cloud

The previous contributions of this dissertation show that executing on the commercial cloud is possible and provide a tool to help automate the deployment of cloud resources. Our next contributions for this dissertation is to push the limitations of scalability within the commercial cloud. In this work we identify and resolve seven different limitations to massive scaling on the commercial cloud. These limitations include: Shared Filesystem Scaling, NAT Limitations, Dynamic Pricing Effects On Spot Prices, Heterogeneous Instance Types With Spot, Scheduler Scalability, User Limits, and API Limits. Several of these limitations are common to execution on all commercial cloud providers including shared filesytem and network limitations, launching heterogeneous instance types to control costs, HPC scheduler scalability, user limits, and API limits.

We also provide a detailed cost analysis for executing at a large scale on AWS and how costs can be lower for smaller workloads by utilizing the same technologies that we utilize at scale. We find that the cost of executing a massive scale workload on AWS is comparable to the cost of execution on local resources.

We utilize PAW to execute a topic modeling CPU intensive workload on 1,119,196 vCPUs simultaneously with minimal user input. The average cost per vCPU-hour for the execution was US \$0.0172 which is comparable to the typical target cost of US \$0.02 per core-hour for local HPC resources. By utilizing this massive HPC environment within the commercial cloud we are able to execute just under a half a million jobs in just two hours. If we had attempted to execute this many jobs on a shared local resource, it would have taken days or even weeks. The execution of this massive CPU intensive HPC application in the commercial clodu is a promising demonstration of how the commercial cloud can be utilized for the execution of scientific applications by researchers.

8.1.4 Commercial Cloud Infrastructure Within Transportation Cyber Physical Systems

After focusing on CPU intensive workloads in our previous work, we shifted to looking into data intensive environments and workloads. We investigate how the commercial cloud and the paradigms that it has introduced can be utilized within data intensive Transportation Cyber Physical Systems (TCPS). We investigate and dissect each layer of the traditional TCPS architecture and the arguments for and against moving each layer to the commercial cloud. We also showcase and present cloud native solutions for each layer and how they can be integrated into the layers to increase performance within the three major commercial clouds.

Our analysis shows that although it is possible to migrate all of the layers within TCPS to the commercial cloud, this may not be feasible due to the real-time constraints or data volume requirements for the different layers. We find that out of all the layers with TCPS, the batch layer is the most promising layer to migrate to the commercial cloud. Within the batch layer, there are no real-time constraints and it is designed for non-interactive jobs. However, due to the typical co-location of the streaming and batch layers if the streaming layer cannot be migrated to cloud it may create duplicate infrastructure and may not be the ideal solution.

We also find that the Infrastructure as Code (IaC) paradigm has the potential to dramatically change how TCPS infrastructure is maintained and managed. By utilizing the IaC paradigm, TCPS infrastructure changes can be tracked within version control which makes documenting, maintaining, and changing the TCPS infrastructure simpler and more efficient. Utilizing IaC allows for TCPS systems to become more agile and be more adaptive to the ever changing conditions that they are constantly facing. As the IaC paradigm is still in it's infancy, the power and scale of the IaC tools will only get better with time. The use of the IaC paradigm should be utilized when migrating any portion of TCPS infrastructure to ensure the reliability and flexibility of the system.

8.1.5 Utilizing Commercial Clouds For Urgent HPC

Finally, in this dissertation we explore the scalability of data intensive workloads and how commercial clouds can be utilized for urgent HPC. Our work provides a proof-of-concept workflow that showcases how commercial clouds can handle urgent HPC tasks that require the processing of a massive amount of data. We accomplish this through the provisioning, utilization, and de-provisioning of an HPC cluster with more than 1.5M vCPUs in about 8 hours. For this experiment we utilized the Google Cloud Platform and PAW for the execution of the experiment. The experiment showcased how urgent HPC can assist with evacuations in the event of an impending disaster and how the different features offered by each commercial cloud provider can provide researchers with a cost-effective on-demand infrastructure for their urgent HPC computing needs.

The workload studied was a data intensive workload that processed traffic video data to detect incidents and other traffic patterns from a collection of pre-recorded video data. Utilizing our massive scale environment, we were able to analyze 2,006,170 hours ($\tilde{2}$11TB) of video data with an average cost of US \$0.008 per vCPU hour. In addition to the seven previous limitations to scaling that we have discussed in this dissertation, we identified and solved two more limitations that were GCP specific: rapid provisioning of instances and API limitations per GCP project.

We also showcased how the commercial cloud allows researchers to execute their workflows across multiple geographic regions. This shows the flexibility and redundancy that the commercial cloud offers researchers. This is extremely useful for urgent HPC workloads as there is some type of impending disaster that could wipe out the computing power in the immediate area. By being able to spread the workload out across multiple geographic regions, the likelihood of total loss/failure decreases.

8.1.6 Future Work

The ever evolving nature of the commercial cloud lends itself well to the extension and continuation of this work. The re-execution and evaluation of the HPCC benchmark suite or similar types of workloads would make for a good comparison with how far the commercial cloud has come since the beginning of this work. As the services and resources change so do the characteristics of the clouds and their capabilities. It would be an interesting comparison study to see how far the cloud has come and to predict where it could be heading in the future.

Another natural extension of this work would be to natively integrate Microsoft Azure into PAW and deploy and evaluate a massive scale environment. This would allow for a more in-depth comparison between the three major commercial cloud providers and may uncover some additional limitations to scaling.

Another possible extension of this work would be to migrate the different layers of the TCPS infrastructure to the commercial cloud. This would allow for an evaluation study to determine the performance of the system and what different bottlenecks are encountered. Through the use of the IaC paradigm, this could provide researchers with a reference architecture and deployment model for further study.

Bibliography

[1] Top 500. Top500 supercomputer sites — june 2019, Jun 2019. Available at: `https://www.top500.org/lists/2019/06/`.

[2] A. Aboudina, I. Kamel, M. Elshenawy, H. Abdelgawad, and B. Abdulhai. Harnessing the power of hpc in simulation and optimization of large transportation networks: Spatio-temporal traffic management in the greater toronto area. *IEEE Intelligent Transportation Systems Magazine*, 10(1):95–106, Spring 2018.

[3] Enis Afgan, Dannon Baker, Nate Coraor, Hiroki Goto, Ian M Paul, Kateryna D Makova, Anton Nekrutenko, and James Taylor. Harnessing cloud computing with galaxy cloud. *Nature biotechnology*, 29(11):972, 2011.

[4] A. W. Apon, P. D. Wolinski, and G. M. Amerson. Sensitivity of cluster file system access to i/o server selection. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'02)*, pages 183–183, May 2002.

[5] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The NAS parallel benchmarks. *The International Journal of Supercomputing Applications*, 5(3):63–73, 1991.

[6] Stephen Bailey. QDO. Available at: `http://www.nersc.gov/users/data-analytics/workflow-tools/other-workflow-tools/qdo/`.

[7] B. Balis. Increasing scientific workflow programming productivity with hyperflow. In *2014 9th Workshop on Workflows in Support of Large-Scale Science*, pages 59–69, Nov 2014.

[8] Bartosz Balis, Marek Kasztelnik, Maciej Malawski, Piotr Nowakowski, Bartosz Wilk, Maciej Pawlik, and Marian Bubak. Execution management and efficient resource provisioning for flood decision support. *Procedia Computer Science*, 51:2377–2386, 2015.

[9] Kyle Banker. *MongoDB in action.* Manning Publications Co., 2011.

[10] Alex Barrett and Michael Basilyan. 220,000 cores and counting: MIT math professor breaks record for largest ever compute engine job, Apr 2017. Available at: `https://cloudplatform.googleblog.com/2017/04/220000-cores-and-counting-MIT-math-professor-breaks-record-for-largest-ever-Compute-Engine-job.html`.

[11] Pete Beckman, Suman Nadella, Nick Trebon, and Ivan Beschastnikh. Spruce: A system for supporting urgent high-performance computing. In Patrick W. Gaffney and James C. T. Pool, editors, *Grid-Based Problem Solving Environments*, pages 295–311, Boston, MA, 2007. Springer US.

[12] H.J.C. Berendsen, D. van der Spoel, and R. van Drunen. Gromacs: A message-passing parallel molecular dynamics implementation. *Computer Physics Communications*, 91(1):43 – 56, 1995.

[13] David M. Blei. Probabilistic topic models. *Commun. ACM*, 55(4):77–84, April 2012.

[14] Jim Bottum, Dustin Atkins, Alan Blatecky, Rick Mcmullen, Todd Tannenbaum, Jan Cheetham, Jim Wilgenbusch, Karan Bhatia, Erik Deumens, James von Oehsen, Marcin Ziolkowski, Asbed Bedrossian, Dan Fay, and Geoffrey Fox. The future of cloud for academic research computing, 04 2017.

[15] Alexander V Boukhanovsky and Sergey V Ivanov. Urgent computing for operational storm surge forecasting in saint-petersburg. *Procedia Computer Science*, 9:1704–1712, 2012.

[16] Andrew J Collins, Peter Foytik, Erika Frydenlund, R Michael Robinson, and Craig A Jordan. Generic incident model for investigating traffic incident impacts on evacuation times in large-scale emergencies. *Transportation Research Record*, 2459(1):11–17, 2014.

[17] Louis Columbus. Forrester's 10 cloud computing predictions for 2018, Nov 2017. Available at: `http://www.forbes.com/sites/louiscolumbus/2017/11/07/forresters-10-cloud-computing-predictions-for-2018/#2719d1694ae1`.

[18] Edward Curry. Message-oriented middleware. *Middleware for communications*, pages 1–28, 2004.

[19] Christian Davatz, Christian Inzinger, Joel Scheuner, and Philipp Leitner. An approach and case study of cloud instance type selection for multi-tier web applications. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGrid '17, pages 534–543, Piscataway, NJ, USA, 2017. IEEE Press.

[20] Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip J Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira da Silva, Miron Livny, and Kent Wenger. Pegasus: a workflow management system for science automation. *Future Generation Computer Systems*, 46:17–35, 2015. Funding Acknowledgements: NSF ACI SDCI 0722019, NSF ACI SI2-SSI 1148515 and NSF OCI-1053575.

[21] Yuri Demchenko, Paola Grosso, Cees De Laat, and Peter Membrey. Addressing big data issues in scientific data infrastructure. In *2013 International Conference on Collaboration Technologies and Systems (CTS)*, pages 48–55. IEEE, 2013.

[22] Jack Dongarra, Iain Duff, Jeremy Du Croz, and Sven Hammarling. LAPACK: dgemm. Available at: `http://www.netlib.org/lapack/explore-html/d1/d54/group__double__blas__level3_gaeda3cbd99c8fb834a60a6412878226e1.html`.

[23] ElasticHPC. ElasticHPC. Available at: `http://www.elastichpc.org/`.

[24] Philip Ewels, Felix Krueger, Max Käller, and Simon Andrews. Cluster flow: A user-friendly bioinformatics workflow tool. *F1000Research*, 5, 2016.

[25] Alces Flight. Flight appliance documentation. Available at: `http://docs.alces-flight.com/`.

[26] Larry Franks, David Giroux, Jason Howell, Matthew Sebolt, and Hassan Rasheed. An introduction to apache kafka on hdinsight - azure, Jun 2019. Available at: `https://docs.microsoft.com/en-us/azure/hdinsight/kafka/apache-kafka-introduction`.

[27] Gartner. Gartner says worldwide IaaS public cloud services market grew 29.5 percent in 2017, Aug 2018. Available at: `https://www.gartner.com/newsroom/id/3884500`.

[28] Amir Globerson, Gal Chechik, Fernando Pereira, and Naftali Tishby. Euclidean embedding of co-occurrence data. *Journal of Machine Learning Research*, 8(Oct):2265–2295, 2007.

[29] P. Gonzlez, X. C. Pardo, D. R. Penas, D. Teijeiro, J. R. Banga, and R. Doallo. Using the cloud for parameter estimation problems: Comparing spark vs MPI with a case-study. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 797–806, May 2017.

[30] Google. Google cloud platform. Available at: `https://cloud.google.com`.

[31] A. Gupta, P. Faraboschi, F. Gioachin, L. V. Kale, R. Kaufmann, B. Lee, V. March, D. Milojicic, and C. H. Suen. Evaluating and improving the performance and scheduling of HPC applications in cloud. *IEEE Transactions on Cloud Computing*, 4(3):307–321, July 2016.

[32] A. Gupta, P. Faraboschi, F. Gioachin, L. V. Kale, R. Kaufmann, B. S. Lee, V. March, D. Milojicic, and C. H. Suen. Evaluating and improving the performance and scheduling of HPC applications in cloud. *IEEE Transactions on Cloud Computing*, 4(3):307–321, July 2016.

[33] A. Gupta and D. Milojicic. Evaluation of HPC applications on cloud. In *2011 Sixth Open Cirrus Summit*, pages 22–26, Oct 2011.

[34] A. Gupta, O. Sarood, L. V. Kale, and D. Milojicic. Improving HPC application performance in cloud through dynamic load balancing. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 402–409, May 2013.

[35] V. Hendrix, J. Fox, D. Ghoshal, and L. Ramakrishnan. Tigres workflow library: Supporting scientific pipelines on HPC systems. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 146–155, May 2016.

[36] A. Iosup, S. Ostermann, M. N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema. Performance analysis of cloud computing services for many-tasks scientific computing. *IEEE Transactions on Parallel and Distributed Systems*, 22(6):931–945, June 2011.

[37] Anubhav Jain, Shyue Ping Ong, Wei Chen, Bharat Medasani, Xiaohui Qu, Michael Kocher, Miriam Brafman, Guido Petretto, Gian-Marco Rignanese, Geoffroy Hautier, Daniel Gunter, and Kristin A. Persson. Fireworks: a dynamic workflow system designed for high-throughput applications. *Concurrency and Computation: Practice and Experience*, 27(17):5037–5059, 2015. CPE-14-0307.R2.

[38] Project Jupyter. About us. Available at: `http://jupyter.org/about`.

[39] Project Jupyter. Jupyterhub. Available at: `https://jupyterhub.readthedocs.io/en/latest/`.

[40] Project Jupyter. Jupyterlab overview. Available at: `https://jupyterlab.readthedocs.io/en/latest/getting_started/overview.html`.

[41] N. K. Kanhere and S. T. Birchfield. A taxonomy and analysis of camera calibration methods for traffic monitoring applications. *IEEE Transactions on Intelligent Transportation Systems*, 11(2):441–452, June 2010.

[42] Neeraj K. Kanhere and S. T. Birchfield. Real-time incremental segmentation and tracking of vehicles at low camera angles using stable features. *IEEE Transactions on Intelligent Transportation Systems*, 9(1):148–160, March 2008.

[43] Neeraj K. Kanhere, Stanley T. Birchfield, and Wayne A. Sarasua. Automatic camera calibration using pattern detection for vision-based speed sensing. *Transportation Research Record*, 2086(1):30–39, 2008.

[44] A. Katsifodimos and S. Schelter. Apache flink: Stream analytics at scale. In *2016 IEEE International Conference on Cloud Engineering Workshop (IC2EW)*, pages 193–193, April 2016.

[45] David Koester and Bob Lucas. Randomaccess benchmark. Available at: `https://icl.utk.edu/projectsfiles/hpcc/RandomAccess/`.

[46] James Lee and Brent Ware. *Open Source Web Development with LAMP: Using Linux, Apache, MySQL, Perl, and PHP.* Addison-Wesley Professional, 2003.

[47] Siew Hoon Leong, Anton Frank, and Dieter Kranzlmller. Leveraging e-infrastructures for urgent computing. *Procedia Computer Science*, 18:2177 – 2186, 2013. 2013 International Conference on Computational Science.

[48] Siew Hoon Leong and Dieter Kranzlmüller. Towards a general definition of urgent computing. *Procedia Computer Science*, 51:2337–2346, 2015.

[49] Piotr Luszczek and Jack Dongarra. HPC challenge: design, history, and implementation highlights. In *Contemporary High Performance Computing*, pages 13–30. Chapman and Hall/CRC, 2013.

[50] Piotr Luszczek, Jack J Dongarra, David Koester, Rolf Rabenseifner, Bob Lucas, Jeremy Kepner, John McCalpin, David Bailey, and Daisuke Takahashi. Introduction to the HPC challenge benchmark suite. *Technical Report*, 2005.

[51] Annie Ma-Weaver and Aaron Blasius. Hpc made easy: Announcing new features for slurm on gcp — google cloud blog, Mar 2019. Available at: `https://cloud.google.com/blog/products/compute/hpc-made-easy-announcing-new-features-for-slurm-on-gcp`.

[52] G. Mariani, A. Anghel, R. Jongerius, and G. Dittmann. Predicting cloud performance for HPC applications: A user-oriented approach. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 524–533, May 2017.

[53] Giovanni Mariani, Andreea Anghel, Rik Jongerius, and Gero Dittmann. Classification of thread profiles for scaling application behavior. *Parallel Computing*, 66:1 – 21, 2017.

[54] Giovanni Mariani, Andreea Anghel, Rik Jongerius, and Gero Dittmann. Predicting cloud performance for HPC applications before deployment. *Future Generation Computer Systems*, 87:618 – 628, 2018.

[55] Nathan Marz and James Warren. *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2015.

[56] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[57] Piyush Mehrotra, Jahed Djomehri, Steve Heistand, Robert Hood, Haoqiang Jin, Arthur Lazanoff, Subhash Saini, and Rupak Biswas. Performance evaluation of amazon EC2 for NASA HPC applications. In *Proceedings of the 3rd Workshop on Scientific Cloud Computing*, ScienceCloud '12, pages 41–50, New York, NY, USA, 2012. ACM.

[58] Microsoft. Microsoft azure. Available at: `https://azure.microsoft.com`.

[59] Microsoft. Power BI: Interactive data visualization bi tools. Available at: `https://powerbi.microsoft.com/en-us/`.

[60] Michael Milligan. Interactive HPC gateways with jupyter and jupyterhub. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*, PEARC17, pages 63:1–63:4, New York, NY, USA, 2017. ACM.

[61] MIT. What is starcluster? Available at: `http://star.mit.edu/cluster/docs/latest/overview.html`.

[62] Kief Morris. *Infrastructure as Code: Managing Servers in the Cloud*. O'Reilly Media, Inc., 2016.

[63] Akihiro Musa, Osamu Watanabe, Hiroshi Matsuoka, Hiroaki Hokari, Takuya Inoue, Yoichi Murashima, Yusaku Ohta, Ryota Hino, Shunichi Koshimura, and Hiroaki Kobayashi. Real-time tsunami inundation forecast system for tsunami disaster prevention and mitigation. *The Journal of Supercomputing*, 74(7):3093–3113, 2018.

[64] M. Naghshnejad and M. Singhal. Adaptive online runtime prediction to improve HPC applications latency in cloud. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 762–769, July 2018.

[65] Henry Neeman. Private Communication, 2017.

[66] Jeremy W. Nicklas, Doug Johnson, Shameema Oottikkal, Eric Franz, Brian McMichael, Alan Chalker, and David E. Hudak. Supporting distributed, interactive jupyter and rstudio in a scheduled hpc environment with spark using open ondemand. In *Proceedings of the Practice and Experience on Advanced Research Computing*, PEARC '18, pages 26:1–26:8, New York, NY, USA, 2018. ACM.

[67] Omnibond. CCQ and HPC jobs. Available at: `http://docs.cloudycluster.com/ccq-and-hpc-jobs/`.

[68] Omnibond. Cloudycluster: Self service cloud HPC. Available at: `http://www.cloudycluster.com/`.

[69] Parkbench. PARKBENCH matrix kernel benchmarks. Available at: `http://www.netlib.org/parkbench/html/matrix-kernels.html`.

[70] A Petitet, R C Whaley, J Dongarra, and A Cleary. Hpl - a portable implementation of the high-performance linpack benchmark for distributed-memory computers. Available at: `http://www.netlib.org/benchmark/hpl/`.

[71] Steve Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of computational physics*, 117(1):1–19, 1995.

[72] B. Posey, C. Gropp, B. Wilson, B. McGeachie, S. Padhi, A. Herzog, and A. Apon. Addressing the challenges of executing a massive computational cluster in the cloud. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 253–262, May 2018.

[73] Brandon Posey. Dynamic HPC clusters within amazon web services (AWS). *Masters Thesis*, 2016. Available at: `https://tigerprints.clemson.edu/all_theses/2392/`.

[74] Brandon Posey, Adam Deer, Vanessa July, Neeraj Kanhere, Dan Speck, Boyd Wilson, and Amy Apon. On-demand urgent high performance computingutilizing the google cloud platform. In *2019 Urgent HPC*. IEEE TCHPC, 2019.

[75] Brandon Posey, Christopher Gropp, Alexander Herzog, and Amy Apon. Automated cluster provisioning and workflow. management for parallel scientific applications in the cloud. In *10th Workshop on Many-Task Computing on Grids and Supercomputer*

(MTAGS17) held in conjunction with the International Conferencefor High Performance Computing, Networking, Storage, and Analysis (SC17)., Denver, CO, Nov 2017.

[76] Brandon Posey, Linh Bao Ngo, Mashrur Chowdhury, and Amy Apon. Infrastructure for transportation cyber-physical systems. In *Transportation Cyber-Physical Systems*, pages 153–171. Elsevier, 2018.

[77] L. Ramakrishnan, S. Poon, V. Hendrix, D. Gunter, G. Z. Pastorello, and D. Agarwal. Experiences with user-centered design for the tigres workflow API. In *2014 IEEE 10th International Conference on e-Science*, volume 1, pages 290–297, Oct 2014.

[78] Hassan Rasheed and David Giroux. What are the apache hadoop and apache spark technology stack? - azure hdinsight, Jun 2019. Available at: `https://docs.microsoft.com/en-us/azure/hdinsight/hdinsight-overview`.

[79] A. Rayamajhi, M. Rahman, M. Kaur, J. Liu, M. Chowdhury, H. Hu, J. McClendon, K. Wang, A. Gosain, and J. Martin. Things in a fog: System illustration with connected vehicles. In *2017 IEEE 85th Vehicular Technology Conference (VTC Spring)*, pages 1–6, June 2017.

[80] Robert Ricci, Eric Eide, and CloudLab Team. Introducing cloudlab: Scientific infrastructure for advancing cloud architectures and applications. *; login:: the magazine of USENIX & SAGE*, 39(6):36–38, 2014.

[81] William Rohm, Gary Ericson, Alma Jenks, Tamra Myers, and Matt Winkler. Provision a linux centos data science virtual machine on azure, Mar 2018. Available at: `https://docs.microsoft.com/en-us/azure/machine-learning/data-science-virtual-machine/linux-dsvm-intro`.

[82] Reza Rokni and James Malone. Google cloud platform for data scientists: using jupyter notebooks with apache spark on google cloud, Feb 2017. Available at: `https://cloud.google.com/blog/products/gcp/google-cloud-platform-for-data-scientists-using-jupyter-notebooks-with-apache-spark-on-google-cloud`.

[83] E. Roloff, M. Diener, A. Carissimi, and P. O. A. Navaux. High performance computing in the cloud: Deployment, performance and cost efficiency. In *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*, pages 371–378, Dec 2012.

[84] I. Sadooghi, S. Palur, A. Anthony, I. Kapur, K. Belagodu, P. Purandare, K. Ramamurty, K. Wang, and I. Raicu. Achieving efficient distributed scheduling with message queues in the cloud for many-task computing and high-performance computing. In *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 404–413, May 2014.

[85] SchedMD. Large cluster administration guide, Jun 2018. Available at: `https://slurm.schedmd.com/big_sys.html`.

[86] Amazon Web Services. Amazon Web Services (AWS). Available at: `https://docs.aws.amazon.com`.

[87] Amazon Web Services. Cfncluster. Available at: `https://cfncluster.readthedocs.io/en/latest/`.

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

[89] Robert Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.

[90] Tableau. Tableau products. Available at: `https://www.tableau.com/products`.

[91] Ronald C Taylor. An overview of the hadoop/mapreduce/hbase framework and its current applications in bioinformatics. In *BMC bioinformatics*, volume 11, page S1. BioMed Central, 2010.

[92] Kir Titievsky. Google cloud platform and confluent partner to deliver a managed apache kafka service, May 2018. Available at: `https://cloud.google.com/blog/products/gcp/google-cloud-platform-and-confluent-partner-to-deliver-a-managed-apache-kafka-service`.

[93] D. Tomi, Z. Car, and D. Ogrizovi. Running HPC applications on many million cores cloud. In *2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 209–214, May 2017.

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

[95] TrafficVision. Trafficvision. Available at: `http://www.trafficvision.com/`.

[96] UW-Madison. What is HTCondor? Available at: `https://research.cs.wisc.edu/htcondor/description.html`.

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

[98] Mehul Nalin Vora. Hadoop-HBase for large-scale data. In *Proceedings of 2011 International Conference on Computer Science and Network Technology*, volume 1, pages 601–605. IEEE, 2011.

[99] Kimli W and Adrian Johnson. Azure cyclecloud overview. Available at: `https://docs.microsoft.com/en-us/azure/cyclecloud/overview`.

[100] Guoxi Wang and Jianfeng Tang. The NoSQL principles and basic application of cassandra model. In *2012 International Conference on Computer Science and Service System*, pages 1332–1335. IEEE, 2012.

[101] Guozhang Wang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, Mammad Zadeh, Neha Narkhede, Jun Rao, Jay Kreps, and Joe Stein. Building a replicated logging system with apache kafka. *Proc. VLDB Endow.*, 8(12):1654–1655, August 2015.

[102] Yi Wang, Hongjie Bai, Matt Stanton, Wen-Yen Chen, and Edward Y Chang. Plda: Parallel latent dirichlet allocation for large-scale applications. In *International Conference on Algorithmic Applications in Management*, pages 301–314. Springer, 2009.

[103] Henry Weller, Chris Greenshields, and Cristel de Rouvary. The openfoam foundation, Sep 2018. Available at: `http://www.openfoam.org/`.

[104] Michael Wilde, Mihael Hategan, Justin M. Wozniak, Ben Clifford, Daniel S. Katz, and Ian Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9):633 – 652, 2011. Emerging Programming Paradigms for Large-Scale Scientific Computing.

[105] A. Wong, D. Rexachs, and E. Luque. Parallel application signature for performance analysis and prediction. *IEEE Transactions on Parallel and Distributed Systems*, 26(7):2009–2019, July 2015.

[106] Dandong Yin, Yan Liu, Anand Padmanabhan, Jeff Terstriep, Johnathan Rush, and Shaowen Wang. A cybergis-jupyter framework for geospatial analytics at scale. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*, PEARC17, pages 18:1–18:8, New York, NY, USA, 2017. ACM.

[107] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, October 2016.