Towards Compute Flexibility for Genome Analysis in the Hybrid Cloud

Takeshi Yoshimura[1,a)] Tatsuhiro Chiba[1,b)]

Abstract: Genome analysis has become an emerging research area in medical and life science. Genome Analysis Toolkit (GATK), an industry-standard genome analysis tool, enables to run and speed up genome analysis in the cloud. Cromwell, a workflow engine for GATK enables to define and reproducible pipelines for complex data processing as files written in WDL, a domain-specific language for defining genome workflows. Their motivation is to efficiently process a huge amount of genome data in the cloud. However, the current design of Cromwell and GATK has less flexibility in terms of choices of the cloud vendors and storage due to vendor lock-in. In this paper, we extend Cromwell to support OpenShift and multiple cloud object storage to avoid vendor lock-in. We also characterize performance overheads and resource underutilization of existing Cromwell. It leads to our design leveraging copy bypassing with container storage interface and cost-efficient pod scheduling with cluster autoscaling. This paper demonstrates early experimental results of copy overheads and resource utilization under managed Red Hat OpenShift 4.6 on IBM Cloud. The experiments show that copy reductions of our backend reduced the elapsed time for an example workflow by 14% and 20% compared to existing backends. Also, cluster autoscaling reduced the cost of a best-practice workflow by 31%.

Keywords: Cloud, Container, Cloud object storage, Genomics, OpenShift

1. Introduction

Genome analysis has become an emerging research area in medical and life science. In particular, understanding human DNA structures is their essential element. DNA structures are represented as a string of initials of four nucleic acids such as `TACTTGATC`. Their variant discovery (insertion, deletion, etc.) helps understand diseases, for example.

From the perspective of computing systems, genome analysis poses challenges in its data scale and required computation power. A single genome file is generated as a 100 GB dataset including errors and redundancy. Typical genomics workflows start from data preprocessing and eventually step into the variant discovery of input datasets.

As well as other data analytics, much effort has been paid to move genome analysis to the cloud [6]. Genome Analysis Toolkit (GATK) [3], an industry-standard genome analysis toolkit, enables genome analysis in the cloud while speeding up their analysis with scatter-gather. Cromwell [5], a workflow engine for GATK enables reproducible pipelines for complex data processing as files written in WDL, a domain-specific language for genome workflows. WDL files consist of multiple tasks of GATK and Linux command lines and are submitted to backend compute infrastructures by Cromwell. Supported backends are cloud job execution engines (e.g., AWS Batch, Google life science, and Tomcat-based deployments for Kubernetes) and on-premise HPC clusters (e.g., LSF [7]). Cloud infrastructures enable users to easily leverage on-demand compute resources and automated cluster management of cloud infrastructures to maximize analysis speed and minimize costs for computing and storage. Cloud object storage (COS) is often used as primary storage for its cheap, unlimited, and high-available capacity.

However, the current design of Cromwell and GATK has less flexibility in terms of choices of the cloud vendors and storage due to vendor lock-in. In particular, cloud backends for Cromwell assume all the input and output resources are at the same cloud as compute nodes. Deployments of compute nodes and storage at the same site improve the performance of workloads, while it prevents us from running them under complex situations. For example, users may need to run a part of genome analysis on an on-premise or non-supported cloud such as IBM Cloud to meet constraints of data location and privacy. In that case, they need to use on-premise backends with HPC schedulers to run GATK on multiple locations. Unfortunately, it is more challenging for users to set up and manage on-premise infrastructures than public clouds.

In this paper, we introduce our OpenShift [12] backend for Cromwell with multiple cloud object storage to avoid vendor lock-in. Our backend enables users to reuse their workflows deployed on OpenShift regardless of underlying clouds and data location. OpenShift is now provided as a common managed enterprise Kubernetes at public clouds but also deployed as a cluster manager for on-premise while offering standardized interfaces for any sites. Cromwell translates WDL files to OpenShift batch jobs and they are deployed as containers in a compute cluster. Users

[1] IBM Research — Tokyo, Hakozaki Chuo-ku, Japan
[a)] tyos@jp.ibm.com
[b)] chiba@jp.ibm.com

can manage and monitor each job as a container with existing tools such as Grafana while reducing cloud costs with cluster autoscaling public OpenShift services offer.

Towards genome analysis in the hybrid cloud, our primary concern is cost and performance since workloads may access remote compute or storage in any sites. In this work, we also characterize problems of current designs of Cromwell and GATK workflows.

First, Cromwell backends have an overhead to copy input and output files between local filesystem (FS) and COS since data can be located at COS but GATK workflows depend on local Linux FSs. COS has different semantics from local FSs to provide its simple and cheap functionality for massive data warehouses with high availability. Therefore, every Cromwell backend needs to copy input data from COS to local FS (localization) and synchronize local outputs to COS (delocalization). Localization is a common performance overhead of GATK workflows and existing backends.

Our backend leverages CSI (cloud storage interface) to solve the copy overheads. CSI enables containers to mount multiple COS as a normal Linux FS via FUSE (filesystem in userspace). Unlike existing backends, legacy workflows do not need additional localizations since they can directly access COS via POSIX-like FS interfaces. Our backend translates COS reads and writes to operations on mounted paths at container FSs so that users still can specify unique file schemas for cloud object storage such as s3:// and gs://. File readers from COS are directly translated into a file read on mounted paths. In contrast, Cromwell splits writing a file to COS into temporal writes to a local file and an indirect copy of it to COS since FSs for COS slowdown (or do not provide) random writes due to the limitation of COS.

Second, GATK workflows tend to underutilize cluster resources due to a scatter-gather execution model. Scatter jobs are highly parallelized to consume data. In contrast, gather jobs tend to run as a single job to combine outputs of scatter jobs. Thus, it is difficult for users to determine the best cluster size for each workflow.

To solve resource underutilization, our backend enables users to specify job scheduling policies that are suitable to cluster autoscaling in OpenShift. Cluster autoscaling assumes to accumulate as many pods as possible to the minimum number of nodes, but the default scheduling policy of Kubernetes is the opposite; they try to balance resource usages among a cluster. As a result, our backend enables each workflow to modify the pod scheduler to increase the chances to remove nodes. Also, it allows advanced scheduling constraints such as node selector and taints tolerations.

This paper demonstrates early experimental results of copy overheads and resource utilization under managed Red Hat OpenShift on IBM Cloud. The experiments show that copy reductions of our backend reduced the elapsed time for an example workflow by 14% and 20% compared to existing backends. Also, cluster autoscaling with our modified job scheduling reduced the cost of a best-practice workflow by 31%. The used best-practice workflow assumed input files at Google Storage but we successfully run it on IBM Cloud with IBM COS as our output destination. The experimental result indicates that ClusterAutoscaler is essential to improve cost-efficiency, but we still underutilize compute clusters.

2. Cromwell and GATK

GATK offers a Java-based command for genome analysis in Linux. It supports various sub-commands for genome analysis such as variant discovery with target and reference human genome. Users also can implement complex genome pipelines in Linux combining these sub-commands and Linux commands such as `python`.

In this section, we pick up an example workflow from Genomics on AWS [1]. The workflow consists of GATK subcommands `HaplotypeCaller` and `MergeGVCF`. They generate `GVCF` output files from input files for interval (how to partition processing), `BAM` (analyzed human genome sequence), and `Reference` (reference genome sequences to be compared with). The example workflow defines scatter jobs with `HaplotypeCaller` and a gather job with `MergeGVCF`.

In this paper, we do not deeply explain the details of each subcommand internal and data format. However, we pick up and discuss key characteristics of GATK and Cromwell from the perspective of their model of exectuion, storage, and engineering. We then discuss their problems when we create a new backend for Cromwell.

2.1 Scatter-gather executions

GATK supports the "scatter-gather" based pipeline execution. Users can launch multiple GATK subcommands with different interval files. Then, each scatter job utilizes interval files to determine consumed region of input data and generates scattered outputs. Finally, users can start combine all the scattered outputs with a gather job. Spark often shows better scalability than scatter-gather, but many workflows still utilize scatter-gather commands.

Scatter-gather execution in a static computing cluster underutilizes computing resources because scatter requires many nodes but gather uses only one. AWS Batch and Google life sciences provide dynamic cluster autoscaling for on-demand resource usages.

A solution to resource underutilization for OpenShift is its cluster autoscaling add-on. However, we still need to carefully build cluster environments since OpenShift tries to deploy many system pods on cluster nodes. Also, we need to ensure storage consistency to tolerate dynamic node join and release to cluster storage.

2.2 Gap between COS and POSIX

BAM and Reference files can be from 10 to 100 GB. Thus, storage capacity and cost are common challenges for GATK workflows. Many example genome inputs are stored on COS such as Google storage and AWS S3.

Existing GATK workflows heavily depend on GATK and Linux command lines running on a local POSIX FS. COS purposely provides simplified APIs to improve costs and availability compared to local FS. For example, COS does not support random and even append writes to each file on a bucket. Random file

reads are supported, but they still have huge latency.

Consequently, users need to manually copy files between COS and local FSs unless they use an external workflow engine, i.e., Cromwell. A key role of Cromwell backends is to fill the gap between COS and local files. They automatically copy input/output files before/after executing a GATK workflow between local FS and COS.

Unfortunately, file copies between COS and local FSs cause a performance bottleneck at storage for jobs including GATK scatter-gather and normal Linux command lines. The AWS Batch backend requires file copies before and after `HaplotypeCaller` and `MergeGVCF` in the AWS example workflow. They read huge input files and write intermediate files, which the next job may read again. Also, scatter-gather execution easily causes duplicated reads within a node. For example, many scatter tasks (`HaplotypeCaller` in the example) assigned at a node read the same BAM file from COS, but they are not cached or deduplicated.

Kubernetes (and OpenShift) provides rich functionality to customize cluster storage as a persistent volume with container storage interface (CSI). An S3 plugin for CSI enables us to directly mount and access a COS bucket as a local FS. However, it still does not fill the gap between COS and local FSs. Genome tasks cannot randomly write or read with optimized performance on a local FS for a COS bucket.

2.3 Legacy code with WDL

GATK and Cromwell enable analysis reproduction with WDL files as a key part of scientific activities. WDL files contain a set of GATK/Linux command lines to implement genome analysis. These binaries are packaged as Docker container images. Users can easily replay the same analysis on the same/different datasets even if the analysis requires complex pipelines. WDL files are distributed as best practice workflows on public code repositories.

Figure 1 and 2 show an example WDL file and input JSON file. WDL files contain various built-in functions to easily define complex pipelines. For example, read_lines() enable to extract inputs and pass the file content to other built-in functions or command lines. ‘scatter‘ clauses are used to define scatter tasks. Each task can consume outputs of former jobs. Cromwell calculates task execution flows and automatically requests task execution to backends with input and output files.

So, the backend components need to translate given WDL files to be suitable for backend infrastructures while preserving WDL portability. To do so, all the backends must be able to start Docker images with specified resource usages. Also, they need to translate input and output files according to the configuration such as COS as output destinations before/after file copies described in Section 2.2. For our case, we need to translate WDL files to be a container job with valid mount paths for inputs and outputs. Note that files on COS can be accessed not only by genome tasks but also Cromwell itself for built-in file functions such as read_lines() and size().

A problem of current Cromwell is that existing backends are not flexible enough to support multiple COS. In particular, they

```
workflow Test {
  File bam
  File reference
  File intervals
  Array[File] invs = read_lines(intervals)
  String gvcf = basename(bam, ".bam")+".g.vcf.gz"
  scatter (interval in invs) {
    call HCTask {
      input:
        bam = bam,
        reference = ref_fasta,
        interval = interval,
        gvcf = gvcf
  }}
  call MergeTask {
    input:
      input_vcfs = HCTask.output_gvcf,
      gvcf = gvcf
  }
  output { File hcgvcf = MergeTask.output_vcf }
}
task HCTask {
  File bam
  File reference
  String interval
  String gvcf
  command {
    /gatk/gatk HaplotypeCaller \
    -R ${reference} -I ${bam} \
    -O ${gvcf} -L ${interval}
  }
  runtime {
    docker: "broadinstitute/gatk:4.0.0.0"
    memory: "10 GB"
    cpu: 1
  }
  output { File output_gvcf = "${gvcf}" }
}
task MergeTask {
  Array [File] input_vcfs
  String gvcf
  command {
    /gatk/gatk MergeVcfs \
    --INPUT=${sep=' --INPUT=' input_vcfs}
    --OUTPUT=${gvcf}
  }
  runtime {
    docker: "broadinstitute/gatk:4.0.0.0"
    memory: "30 GB"
    cpu: 1
  }
  output { File output_vcf = ${gvcf}" }
}
```

Fig. 1 Example WDL.
The figure shows simplied job definitoin with scatter-gather. WDL enables users to specify docker images and resource usages. Each task can execute any command lines. WDL also has built-in functions such as read_lines, which extracts file contents.

```
{
  "Test.bam":"s3://bucket/input.bam",
  "Test.reference": "s3://bucket/ref.fasta",
  "Test.intervals": "s3://bucket/intervals.txt"
}
```

Fig. 2 Example input JSON file.
Users can pass input variables to WDL files in JSON format. This increases the reusability of WDL files. However, built-in functions such as read_lines make WDL files dependent on input sources.

do not optimize heterogeneity such as using AWS S3 from an on-premise HPC cluster within the same workflow. Also, Cromwell backends do not support changing API endpoints for COS to enable S3-compatible storage like IBM Cloud and local object storage such as MinIO and Ceph [13]. To obtain optimized performance, users need to manually migrate data to local storage or a single COS and modify all the inputs to point to the storage location. This is not an easy task for Cromwell users since they also

Infrastructure	Copy opt.	Autoscaling	Hybrid cloud
AWS, Google	No	Yes	No
TESK (Kubernetes)	No	No	No
LSF	Yes	Yes	No
OpenShift (ours)	Yes	Yes	Yes

Table 1 Comparison with existing infrastructures.

Cromwell supports various cloud infrastructures or software backends for genome workflows. "Copy opt." represents automation and optimization of huge file copies between COS and local FS. The "Autoscaling" column shows the Cromwell backend and its backend infrastructure supports cluster autoscaling. Users can deploy any sites with any COS with OpenShift ("Hybrid cloud"). Cromwell supports more infrastructures such as other HPC clusters, Azure Batch, and Alibaba Cloud **?**, but all of them do not satisify the above properties.

need to check and update interval files (e.g., Figure 1). Thus, we aim to ensure the portability of JSON input files as well as WDL files.

2.4 Summary and Comparison with existing approaches

In summary, Cromwell causes a bottleneck at storage for many legacy GATK workflows. A drastic approach is to rebuild a new workflow engine to improve the storage model like Glow [11]. However, we rather retrofit existing backends to preserve rich amounts of legacy genome analysis codes. Figure 5 lists the summary of comparison of Cromwell backends. Our OpenShift backend can resolve three requirements discussed in this section, but others cannot.

Our backend design is strongly inspired by existing public cloud backends for AWS Batch and Google life science. They expose APIs to start containers with commands and specified resource usages on top of their virtualized infrastructures. They automatically install Docker environments on the allocated nodes and dynamically create/delete nodes according to a load of workflows. Unfortunately, they do not support multiple inputs and outputs for S3-compatible COS (e.g., IBM COS) and depend on simple file copies between COS and local FS.

Our backend simplifies job deployments compared to the existing Kubernetes backend, `TESK`. `TESK` additionally creates an API service container, in which users can submit tasks via standardized APIs for genome tasks. Cromwell is deployed to the same cluster as the API container and it communicates with each other. Each genome task is deployed as a container, but its storage model is different from ours. It creates a container to share input, intermediate, and output files among task containers. Currently, they assume using cluster-local storage to store all the data, but the model is not suitable for external storage like COS. Also, they cannot eliminate the bottleneck at storage due to huge file copies. Their architecture is also difficult to apply cluster autoscaling because of the difficulty of restarting or migrating their API and storage services. We utilize CSI and carefully manage file I/O to avoid the bottleneck at storage.

Users can use HPC schedulers to build flexible compute environments to meet our requirements. For example, LSF [7] supports cluster autoscaling at various public clouds and copy optimization with hard link and cached inputs. However, it does not support S3-like COS such as IBM COS and the capability of hard links are limited to specific devices. To obtain the optimized

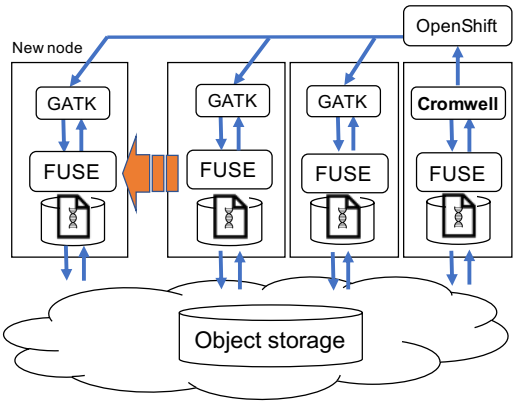

Fig. 3 Architecture overview.

Cromwell runs as a deployment pod to communicate with clients and OpenShift master. Each genome task starts as a container job with COS mounted as a container local FS. Cromwell configurations and COS credentials are stored as config maps or secrets in OpenShift. OpenShift automatically starts or shutdowns nodes according to cluster resource usages with cluster autoscaling add-on.

performance, they need to manually mount a POSIX FS shared among a cluster and specify them as input and output paths. However, the cluster setups require advanced knowledge to create consistent distributed storage even under cluster autoscaling. Also, it remains challenging for users to carefully migrate data to avoid file copies. Our backends simplify user's additional efforts to optimize storage and manage clusters leveraging the features of OpenShift.

3. OpenShift backend for Cromwell

In this work, we develop a new Cromwell backend to deploy genome workflows on OpenShift. OpenShift abstracts backend cloud infrastructure and provides advanced features including our requirements (i.e., cluster autoscaling and CSI) to efficiently run applications on a cloud cluster. Users can simplify cluster management regardless of underlying cloud infrastructure including public and private ones.

Figure 3 overviews our backend architecture including COS. We deploy Cromwell as a deployment pod in an OpenShift cluster with special permissions assigned to manage jobs. COS is mounted as a normal Linux FS in each container as persistent volume for S3-like storage using CSI (deployed as daemonset). When OpenShift cluster autoscaling decides to increase/decrease nodes, the CSI daemonset is automatically deployed/destroyed from/to the target nodes. Note that Cromwell already provides Swagger Web UI so that clients can deploy their jobs to their clusters. We need to expose the port for the Swagger Web UI in OpenShift settings.

Problems described in Section 2 cannot be solved without carefully updating multiple software components. Cluster autoscaling requires appropriate configurations of OpenShift as well as stateless CSI daemonset. Storage bottleneck is solved by utilizing node-wide shared mounts and Cromwell's strategic file copies. We enable hybrid cloud usages by enabling reading Kubernetes secrets in Cromwell and Cromwell's change to translate paths for multiple COS. We describe individual solutions per logical soft-

```
k8s {
  auths = [{
    name = "k8sauth",
    scheme = "incluster"
  }]
}
aws {
  application-name = "cromwell"
  region = "ap-northeast-1"
  auths = [{
    name = "s3_auth"
    scheme = "k8s_secrets"
    keys = [{
      name = "s3-secret",
      namespace = "cromwell"
    }]
  }]
}
google {
  application-name = "cromwell",
  auths = [{
    name = "no_auth",
    scheme = "no_auth"
  }]
}
engine { filesystems {
    s3 { auth = "s3_auth" },
    gcs { auth = "no_auth" },
}}
backend {
  default = "k8s"
  providers { k8s {
    actor-factory = "..."
    config {
      auth = "k8sauth"
      filesystems {
        s3 { auth = "s3_auth" },
        gcs {auth = "no_auth" }
      }
      namespace = "cromwell"
      k8sServiceAccountName = "cromwell-sa"
      pullImageSecrets = ["regcred"]
      s3PvcName = "cos-pvc"
      root = "/cromwell_root/cos-bucket/cromwell"
      schedulerName = "my-scheduler"
      tolerations = "app=cromwell:NoSchedule"
      nodeSelector = "nodeType:cromwell"
    }
  }}
}
```

Fig. 4 Example configuration

ware component in the following subsections.

3.1 Backend configuration

Cromwell users must provide a configuration to specify credentials for cloud infrastructures and how to utilize them. Specifically, our backend requires to pass credentials for OpenShift (or Kubernetes) as compute and COS as storage. We also enable configurations to specify a namespace, scheduling constraints, and other essential pieces to deploy a job to OpenShift as a container.

Figure 4 shows an example configuration for our backend. The file is passed as an input argument for Cromwell. It has five top blocks, “k8s”, “aws”, “google”, “engine”, “backend”. The first three contain credential information to use OpenShift and COS. The latter two specify Cromwell behavior such as which directory is used as outputs.

The “k8s” block allows associating in-cluster credentials for Kubernetes/OpenShift to Cromwell. In the case where `scheme="incluster"` is specified as Figure 4, Cromwell can directly communicate with the Kubernetes/OpenShift master to deploy jobs as containers in the same cluster where Cromwell is deployed. Note that users need to assign an appropriate role to Cromwell pod so that Cromwell can deploy and look up jobs, namespaces, etc. We also provide `scheme="default"` credentials to pass current default credential that `kubectl` uses to Cromwell. It can deploy jobs to a remote cluster as well as `kubectl` users often use for their cluster management.

The “aws” and “google” blocks are used to pass credentials for S3-like COS and Google storage, respectively. Figure 4 shows a case where a user passes a Kubernetes secret as credentials for S3-like COS (IBM Cloud in this case) and sets anonymous ones for Google. By specifying these two block names in the “engine” block, Cromwell can utilize these two credentials when it directly reads input files with S3/Google APIs for built-in functions such as `read_lines()`. Anonymous credentials are useful for many example best-practice workflows distributed by Broad Institute, which use example inputs on publicly available buckets. Users can reuse Kubernetes secrets passed to CSI as described in Section 3.3.

The “backend” block specifies essential configurations for deployed containers. Cromwell uses k8s credentials with “k8sauth” (from “auth”) to deploy containers to specified namespace (“namespace”) with pull credential (“pullImageSecrets”). If Cromwell needs to directly read files at built-in functions, it looks up credential names in other blocks from attribute “filesystem”. Many existing images for genome analysis assume root privileges, and so, users need to set up service accounts for them and pass it as “k8sServiceAccountName”, since OpenShift allows limited permissions by default. Each container mounts a persistent volume claim “s3PvcName” points onto the local path at “root”. As described in Section 3.3, our CSI extracts mount points for multiple COS under the path at “root” and containers can easily access them via the path. Also, users can pass scheduler costraints as `schedulerName`, `tolerations`, `nodeSelector` to enable cluster autoscaling as described in Section 3.4.

3.2 Task manager for OpenShift

Cromwell submits each task in WDL files (e.g., Figure 1) as a container in OpenShift. As described in Section 3.1, users can configure essential credentials and task management of Cromwell. Then, Cromwell monitors the status of each deployed job and terminates them if they are finished after collecting debug information such as terminated container status on an output directory. Users can also monitor each job status via Kubernetes general command like `kubectl` or `oc` for OpenShift.

By default, Cromwell generates a bash script to run the command in WDL files (e.g., Figure 1) and collect standard outputs and errors for it. We reuse the generated script to deploy a task as a container. However, we need to modify input and output files if users pass files with unique schemas (e.g., `s3://` and `gs://`).

Cromwell traverses input WDL files and can determine which files are used as inputs and outputs. The configuration for “aws” and “gs” blocks enable us to know which schemas and buckets can be used in WDL files. Our backend maps COS files with `schema://bucket/path` to `/cromwell_root/bucket/path` if the configuration `root=/cromwell_root/` in the “backend” block. We hook the command generation to replace the string for inputs and outputs to be appropriate local paths. Details for how

to mount COS as local paths are described in Section 3.3.

Compared to backends for AWS Batch and Google, our backends support multiple schemes and buckets and set up endpoints. If Cromwell needs to directly access a bucket for built-in functions, we set environmental variables for credentials at every access to the bucket. The library for S3 integrated to Cromwell looks up and uses the particular keys for environmental variables to access buckets. Also, we added anonymous credentials for many best-practice workflows.

WDL files also enable users to specify resource usages for each task. We reuse the Kubernetes resource claim to specify the resource usages. As far as our observation, AWS Batch and Google life sciences utilize this information only for instance selection. Thus, their resource usage includes ones of all the system services on the instance. In contrast, our backend allocates the specified resource usage only for a genome task.

A side-effect of this slightly different behavior is that users need to be aware of the resource usages of system services for OpenShift and file cache for the genome task. System services consume a small part of every node memory (less than 1 GB), and so, tasks cannot utilize less memory than users expect from what the instance catalog shows. Also, OpenShift and Kubernetes count memory usages for inactive file caches in a container as their memory usages. So, when users try to naively copy huge amounts of files within a container, it can wake up out-of-memory killers to stop the container. This is another motivation to utilize CSI to indirectly access COS via a different container and try to avoid huge copies as described in Section 3.3. We believe this strict model for resource allocation reduces unpredictable performance degradation.

3.3 CSI

CSI modules enable users to mount custom FSs on containers. As shown in Figure 4, the name of persistent volume claim ("s3PvcName") is mounted at a path of "root" in each task container. Each CSI module is deployed as FUSE to every node as a Kubernetes pod (i.e., daemonset). They are instantiated at the associated node startup and terminated at the node shutdown. FUSE instances enable tasks to access files on COS as local FS in Linux.

Our CSI module interleaves each file read/write from/to multiple COS. We reuse Goofys [8], an S3-like storage client for the core logic of our CSI module. Goofys optimizes write performance by utilizing multi-part uploads of COS and large prefetch reads while allowing random reads. However, it does not support random writes. It also increases too much memory footprints for containers if we mount multiple COS as different FUSE instances. We solve these challenges by modifying Cromwell's backend and Goofys itself.

We modify file access operations depending on file operation, i.e., read or write. As shown in Section 3.2, our Cromwell backend translates each COS file path to a corresponding local mount path. Its path translation can also capture the operation type of each file access. We directly specify a local path if the file access is a read since Goofys supports random reads and most of the reads from genome workflows are sequential. In contrast, file writes are separated into temporal file writes to local FS and copying it to COS. As reported in optimization work for overlay FSs [15], writing container FSs may increase overheads, but we take this simple approach because most file writes were small and the writing throughput of genome analysis was bound to CPU.

The issue of utilizing too much memory footprint is resolved by modifying Goofys to support multiple COS at a single FUSE instance. Goofys is already well-structured to support multiple COS, and thus, we added a new pseudo-COS client, which changes target COS and calls the original backend for it according to file paths. We currently put bucket names as the second top directory of a path where Goofys is mounted. Our Cromwell backend also assumes this directory structure to access each COS.

Our CSI module instantiates a single FUSE instance with a single mount point for multiple COS and invokes Linux "bind-mount" to virtually share the mount point among each task container. By doing so, we can de-duplicate file accesses from multiple task containers in a node with the Linux page cache. This situation is typical when users try to run scatter-gather jobs like Figure 1 in a few huge instances. Scatter-gather jobs often read the same input file with different intervals. In this case, existing approaches like AWS and Google cannot utilize node-level page cache although file accesses are duplicated.

An important design decision here is that Goofys is a *stateless* client for COS with *write-through* cache. Goofys leverages page cache, but it also writes through each write to COS and keeps the container stateless. Stateful clients (e.g., Agni [9]) may drastically optimize file accesses with cluster-wide buffering and caching, but they also suffer from the complexity to handle cluster autoscaling described in Section 3.4.

3.4 Cluster autoscaling

We utilize an add-on for cluster autoscaling in OpenShift. The add-on monitors resource usages of tasks on a worker pool, and creates new instances if a new container cannot run under available resource slots in the pool. Then, workers are terminated if they are underutilized more than configured periods. Unfortunately, we still need additional configurations of clusters to improve resource usages in a cluster. As shown in Figure 4, users need to assign custom scheduling policies.

The default Kubernetes scheduler tries to balance the resource usage among cluster nodes. However, cluster autoscaling requires packing more containers into fewer nodes before deleting unused nodes. So, users should set a custom scheduler `ClusterAutoscalerProvider` to each task. By doing so, the scheduler tries to assign tasks to the most frequently used nodes.

Also, cluster autoscaling cannot delete nodes if there are any system services on them. OpenShift offers various services to users and they can install any services to a cluster to satisfy their requirements. However, such services can be randomly assigned to a worker node that cluster autoscaling manages. To avoid this, users need to add taints (e.g., `app=cromwell:NoSchedule`) to the worker pool that cluster autoscaling manages. The taints prevent any pods without tolerations like Figure 4 from being scheduled to the tainted nodes. The toleration attributes enable Cromwell to automatically assign the tolerations to each task container. In contrast, users may need to avoid scheduling task con-

tainers on particular worker pools. An attribute "nodeSelector" can enable to bind task containers to specific groups of nodes in a cluster.

Users also need to enable task preemptions when the daemonset of our CSI module is scheduled after task containers on a node. Without task preemptions, we sometimes observed scheduling deadlocks.

4. Experiments

In this section, we examine the following characteristics of our OpenShift backend for Cromwell.

- File copy reductions with direct access to COS
- Portability of workflows using different COS
- Cost efficiency of cluster autoscaling

To this end, we first conduct a performance comparison with different backends to understand the benefits of direct access to COS. Next, we present another experiment of cluster autoscaling at IBM Cloud with a best-practice workflow for Google Cloud. Our demonstration runs on Red Hat OpenShift 4.6 on IBM Cloud Container Platform.

4.1 Backend performance

Our first experiment runs a sample scatter-gather workflow described in Section 2 on LSF, TESK, and our OpenShift backends. The workflow uses input files in IBM COS at `jp-tok`. It first reads an interval file at COS and concurrently starts 50 scatter tasks. The scatter tasks generate intermediate files, and then, the workflow eventually starts a single gather task to combine the intermediate files into a compression file.

Our LSF, TESK, and OpenShift clusters consist of ten compute nodes of bx2-8x32 in `jp-tok` at IBM Cloud (i.e, 8 virtual CPUs and 32 GB RAM, 16 Gbps network, and 100-GB, 3000-IOPS block storage). In this experiment, we do not enable cluster autoscaling to focus on the performance difference of backends' storage usages. As discussed in Section 2.4, three clusters differently behave around input and intermediate files.

The LSF cluster has an additional storage node for a shared network filesystem (NFS) to store all the intermediate and output files. The NFS is built with a bx2-8x32 node and 100-GB, 3000-IOPS block storage. The LSF cluster is configured to copy and cache input data to the NFS before tasks for the workflow starts.

Our TESK cluster runs an API server and executes tasks on a managed OpenShift 4.6 cluster with ten nodes of bx2-8x32. TESK creates a persistent volume claim (PVC) with 10 GB, 100-IOPS block storage for each task to store cached input and intermediate files. The API server launches a job pod and it then starts ones for copying inputs from COS to a PVC, executing a task, copying outputs from the PVC to COS.

Our OpenShift cluster uses the same environment as our TESK cluster. However, it does not create additional storage. It reuses the container default storage to store intermediate files and each container eventually copies to COS.

To break down performance results, we divide a job execution into file copies and task executions. For the LSF backend, we estimate the time of file copies by subtracting the starting time for the first task and a workflow submission. For the TESK backend, we retrieve the pod creation time for the very first task execution and estimate file copies as well as the LSF backend. Our OpenShift backend does not copy files, and thus, the time for file copies is estimated as zero although files are transferred on-demand during each task execution.

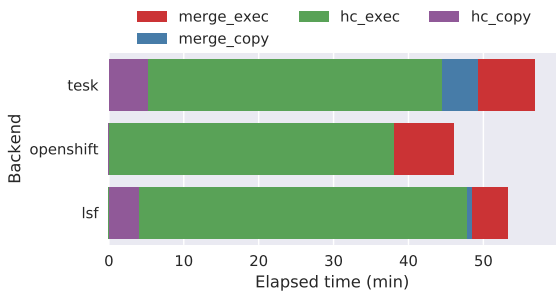

Fig. 5 Performance comparison and breakdown.
Figure shows estimated times for file copies (_copy) and execution (_exec) of two tasks, `HaplotypeCaller` (hc) and `MergeGVCF` (merge). Each estimated time is stacked and shows the total elapsed time for the workflow. The results are the average of five runs.

Figure 5 shows our experimental results and breakdowns of the workflow under each backend. Our OpenShift backend reduced the total time for the workflow by 14% (7.2 minutes) and 19% (10.8 minutes) compared to LSF and TESK, respectively. A major reason for the performance improvement is reduced file copies. Surprisingly, our backend also reduced the execution time for `HaplotypeCaller`.

TESK added a 10-minutes overhead to copy files for two tasks and 1.2-minutes for `HaplotypeCaller` execution. The major TESK overhead was derived from allocating a PVC for each task as well as file copies between COS and the PVC. Also, on the TESK cluster, many `HaplotypeCaller` execution pods waited for completions of other pods because of CPU/memory slot shortages in the cluster. TESK created four pods for each task (controller, input copier, task execution, and output copier) and it consumed more resource slots than our OpenShift backend.

LSF showed the best performance on `MergeGVCF` tasks since it stores intermediate data on a cluster-local NFS, which have lower latency than COS. However, `HaplotypeCaller` tasks spent more time than our OpenShift backend. Our OpenShift backend enables each task to read files on node-local FS, while LSF tasks access a remote NFS. Another possibility is that the NFS became a bottleneck due to concurrent access. So, we are planning to test other high-performance FSs such as IBM Spectrum Scale (GPFS) and CephFS.

4.2 Cluster autoscaling

As a demonstration for cluster autoscaling and multi-COS supports, we run a best-practice workflow [4] distributed by Broad Institute. The workflow runs data preprocessing and variant discovery with parallelized HaplotypeCaller on public datasets in Google Storage, while we configured outputs are written into IBM COS at `jp-tok`.

We set up from 1 to 24 worker nodes of bx2.32x128 (32 vcores of Cascadelake and 128GB RAM) of OpenShift 4.6 in a Tokyo availability zone with ClusterAutoscaler. we used a very aggres-

metrics	autoscaling	static
runtime hours	3.4	2.8
node hours	26.9	66.4
billing node hours	50.0	72.0
estimated cost	$25	$36

Fig. 6 Cost and performance.
We run the same workflow with autoscaling and static cluster. "Node hours" are the total hours of node runtime during the experimental period. "Billing node hours" represent an estimated cost based on the hourly billing model.

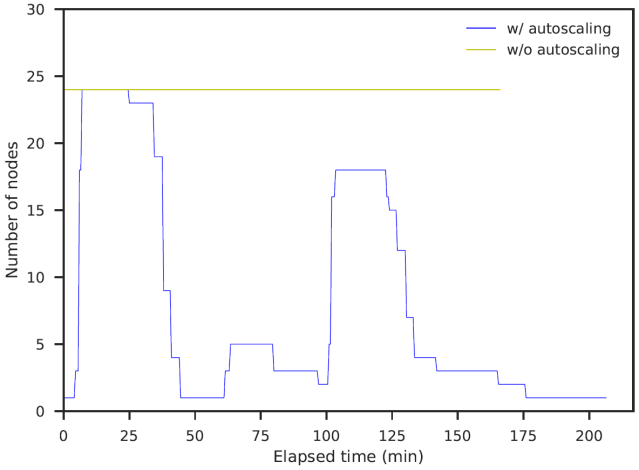

Fig. 7 Number of nodes

sive deletion policy: deleting nodes with 3 minutes of unused time. For comparison, we also conduct the same experiment but with a static cluster running on 24 worker nodes.

Figure 6 shows the estimated cost and performance of the best-practice workflow [4] with or without autoscaling. Autoscaling increased the runtime hours of the workflow by 21% (3.4 hours vs. 2.8 hours) but reduced the total cost by 44% (25*vs*36). Many of the nodes under autoscaling stop within an hour, and so, the node hours of the autoscaling cluster were much lower than the static. The results of billing node hours show that charges even one second from node start to termination as one-hour node usage. The autoscaling-enabled cluster should need 0.5/*hour*x50.0 =25, while the static should cost 0.5/*hour*x72 =36.

Figure 7 show the time-series data of the number of nodes. The best-practice workflow has two big spikes and one small spike in terms of resource consumption. These spikes were derived from scatter parallelization. After these scatters, the workflow uses much fewer resources to gather phase. These workload characteristics highly motivate using ClusterAutoscaler of OpenShift.

Figure 8 and 9 show the ratio of actual resource usage from the total reserved one in the cluster. Autoscaling increased memory utilization especially around the end of this workflow by reducing the number of nodes. However, CPU utilization was still up to around 35%. This number was lower than we expected. A major reason is that the required resources specified in WDL files are overestimated. Precise resource estimation is practically challenging, but we believe other features of OpenShift like Horizontal Pod Autoscaler solve this issue.

5. Related work

Our prior work [14] gives a performance analysis and optimization of GATK sub-commands using Spark and COS. In particular, HDFS showed performance advantages over COS because its implementation does not utilize multi-part uploads of COS. Our CSI module leverages the multi-part uploads to efficiently write files onto COS. Our Cromwell backend can also utilize Spark sub-commands, but HDFS is not suitable for cluster environments like OpenShift and Kubernetes. Utilizing COS with the insights of our prior work is still important for porting workflows using Spark with our backend.

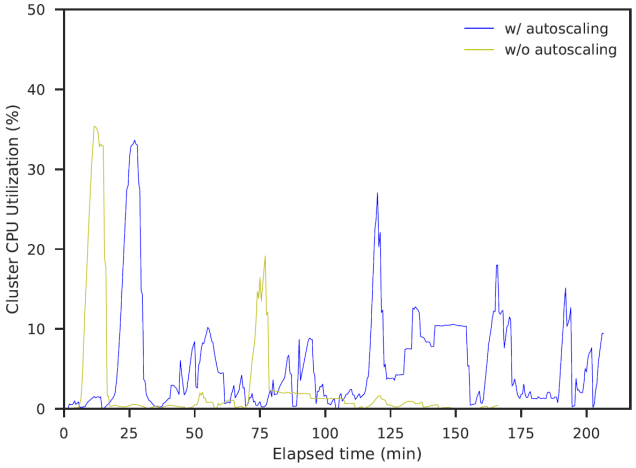

Fig. 8 Cluster CPU utilization

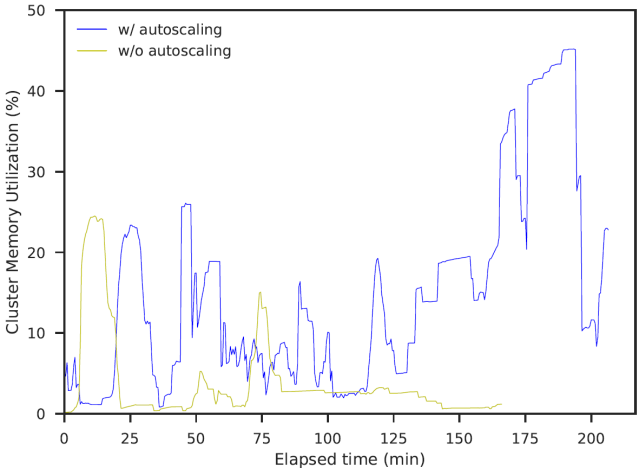

Fig. 9 Cluster memory utilization

Our key observation of this work is that the dominant performance factor of genome analysis is storage. Our architecture utilizes stateless, write-through cache for COS based on Goofys [8]. In contrast, Agni [9] offers a stateful, write-back cache for COS, which shows higher performance than Goofys. However, these write-back cache mechanisms do not support the elasticity of clusters since their consistency models assume a static cluster. Snowflake [10] solves the elasticity of data warehouse by utilizing COS as disaggregated storage. In particular, lazy consistent hashing assumes its stateless storage service (i.e., write-through cache) to easily maintain the data consistency regardless of dynamic node joins and releases. We also take their approach to avoid complex consistency issues under cluster autoscaling. Pocket [2] also offers optimized cache for COS with NVMe while supporting the cluster elasticity. Unfortunately, they assume users

access data via their unique client interfaces, not Linux FS ones, which legacy workflows heavily depend on.

6. Conclusion

In this paper, we present our experiences of developing a new OpenShift backend for Cromwell. Our initial motivation behind it is to reuse legacy codes in the hybrid cloud environments, but current Cromwell is locked in specific cloud vendors or software stacks. In addition to the initial problem, our observation of GATK workflows showed that their characteristics posed challenges of resource underutilization, a storage bottleneck, and a lack of multi-COS usages in a compute cluster. We resolved them by leveraging the rich customizability of OpenShift such as add-on for cluster autoscaling and CSI module as well as our careful design of a new backend. As a demonstration, we reused an existing legacy workflow for Google Cloud on a managed OpenShift at IBM Cloud. The experimental results showed that cluster autoscaling improves resource utilizations although we still do not fully utilize cluster resources.

References

[1] Amazon Web Services: Examples — Genomics Workflows on AWS, (online), available from ⟨https://docs.opendata.aws/genomics-workflows/orchestration/cromwell/cromwell-examples.html⟩ (accessed 2021-06-24).
[2] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle and Christos Kozyrakis: Pocket: Elastic Ephemeral Storage for Serverless Analytics, *In Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)* (2018).
[3] Broad Institute: GATK, (online), available from ⟨https://gatk.broadinstitute.org/hc/en-us⟩ (accessed 2021-06-01).
[4] Broad Institute: gatk-workflows/gatk4-genome-processing-pipeline, (online), available from ⟨https://github.com/gatk-workflows/gatk4-genome-processing-pipeline⟩ (accessed 2021-06-15).
[5] Broad Institute: Home - Cromwell, (online), available from ⟨https://cromwell.readthedocs.io/en/stable⟩ (accessed 2021-06-01).
[6] Geraldine A. Van der Auwera and Brian D. O'Connor: *Genomics in the Cloud*, O'Reilly Media, Inc. (2020).
[7] IBM: IBM Spectrum LSF Suites — IBM, (online), available from ⟨https://www.ibm.com/products/hpc-workload-management⟩ (accessed 2021-06-01).
[8] Ka-Hing Cheung: Goofys, (online), available from ⟨https://github.com/kahing/goofys⟩ (accessed 2021-06-01).
[9] Kunal Lillaney, Vasily Tarasov, David Pease and Randal Burns: Agni: An Efficient Dual-Access File System over Object Storage, *In Proceedings of the ACM Symposium on Cloud Computing (SoCC '19)* (2019).
[10] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala and Thierry Cruanes: Building An Elastic Query Engine on Disaggregated Storage, *In Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)* (2020).
[11] Project Glow: Glow, (online), available from ⟨https://projectglow.io⟩ (accessed 2021-06-01).
[12] Red Hat: Red Hat OpenShift, the open hybrid cloud platform built on Kubernetes, (online), available from ⟨https://www.openshift.com⟩ (accessed 2021-06-01).
[13] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long and Carlos Maltzahn: Ceph: A Scalable, High-Performance Distributed File System, *In Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)* (2006).
[14] Tatsuhiro Chiba and Takeshi Yoshimura: Investigating Genome Analysis Pipeline Performance on GATK with Cloud Object Storage, *In Proceedings of the 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '20)* (2020).
[15] Yu Sun, Jiaxin Lei, Seunghee Shin and Hui Lu: Baoverlay: A Block-Accessible Overlay File System for Fast and Efficient Container Storage, *In Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC '20)* (2020).