Enabling Workload-Driven Elasticity in MPI-based Ensembles

Md Rajib Hossen
The University of Texas at Arlington
Arlington, Texas, USA
mdrajib.hossen@mavs.uta.edu

Vanessa Sochat
Lawrence Livermore National Laboratory
Livermore, CA, USA
sochat1@llnl.gov

Abhik Sarkar
Lawrence Livermore National Laboratory
Livermore, CA, USA
sarkar6@llnl.gov

Mohammad A. Islam
The University of Texas at Arlington
Arlington, Texas, USA
mislam@uta.edu

Daniel J. Milroy
Lawrence Livermore National Laboratory
Livermore, CA, USA
milroy1@llnl.gov

***Abstract*—Interdisciplinary workflows are evolving to accommodate the growing resource diversity and parallelism in modern computing systems. The necessity of integrating various components, including multi-scale simulations and Artificial Intelligence and Machine Learning (AI/ML) with ensemble methods, has made the workflows increasingly complex and challenging to manage using traditional High performance computing (HPC) infrastructure. Cloud computing provides capabilities such as container orchestration, automation, and elasticity to manage the growing heterogeneity, scale, and complexity of HPC systems and workflows. Converged computing, a growing movement that integrates HPC and cloud technologies into a seamless environment, can provide a means to bridge the gap between needs and capabilities. In particular, ensemble-based HPC workflows can benefit from the potential efficiency improvements afforded by these capabilities.**

While MPI-based (Message Passing Interface) workflows have been demonstrated to scale using cloud-native orchestration in Kubernetes, there is little work on understanding the combined impact of autoscaling and elasticity on MPI-based workflows. In this study, we explore the cost and performance of elasticity applied to ensembles of MPI-based HPC simulations using the Flux Operator. We propose a workload-driven autoscaling algorithm that outperforms CPU utilization-based autoscaling for MPI-based ensembles. The efficiency gains afforded by elastic, autoscaled approaches for MPI-based ensembles are described. We demonstrate that the workload-driven algorithm can reduce ensemble completion time by up to 4.7× in comparison with CPU utilization-based autoscaling.

***Index Terms*—MPI-based ensembles, cloud computing, autoscaling, elasticity, converged computing**

I. Introduction

The end of Dennard scaling together with the progressive waning of Moore's Law has caused an explosion of parallelism and a rapid increase of resource heterogeneity and complexity, making it much harder to run and reproduce multidisciplinary scientific workflows [1]. To cope with increased complexity, resource management needs to adapt to schedule diverse workflow components, including multi-scale simulations, in-situ data analysis, and Artificial Intelligence and Machine Learning (AI/ML) techniques [2]–[5], accounting for changing events, resource dynamism, and costs [1]. To add complexity, each component may be mapped to a specific on-node resource (e.g., GPU or dedicated accelerator) or even to a separate cluster with the desired hardware capabilities [6].

The increase in system parallelism has encouraged a transition to ensemble techniques for simulation and uncertainty quantification (UQ) [7]. Ensembles are a core component of high performance computing (HPC) workflows for drug discovery [8], [9], molecular screening [6], cancer research [10], inertial confinement fusion [11], [12], and weather and climate models [13]–[15] among many others. At Lawrence Livermore National Laboratory, one of the largest scientific computing centers in the world, nearly 50% of jobs on large production clusters are ensemble-based [16]. HPC ensemble-based workflows can adapt dynamically based on Machine Learning (ML) models and incorporate in-situ feedback while using Message Passing Interface (MPI) for intra-job communication [10], [17], and be composed of tens [14], [18] to over 100 million jobs [12]. Ensemble-based composite HPC workflows present significant challenges in deployment, portability, reproducibility, and automated management. Running complex scientific workflows on systems of increasing size and heterogeneity with resource dynamism compounds the challenges.

New economic forces are shaping the future of computing, leading to innovations to manage system and workflow complexity. A new fragmentary economic cycle creates specialized computing islands that bring fewer benefits to each other; communities that cannot use innovations financed and produced by market leaders will be left behind [19]. Cloud computing is on pace to overtake all other computing sectors by 2025, attaining nearly $1T in total revenue [20]. As cloud computing becomes a dominant market force, it drives innovation in both hardware [19] and the software needed to manage the rapidly increasing resource diversity, scale, and complexity of current and future systems.

Converged computing, or an environment that provides the performance and efficiency of HPC together with the automation, portability, and reproducibility of the cloud, is a promising emerging area of computing that seeks to address

challenges faced by complex scientific workflows. Kubernetes (K8s) [21], the de-facto standard container orchestration framework, provides native support for automation, reproducibility, and elasticity. K8s' vast (over 90,000) contributor base and widespread adoption in the industry have made it the second largest open-source software project after Linux [22], [23]. The scale of adoption and reliance on K8s makes it an excellent choice for ensuring workflow portability.

From the HPC side of converged computing, Flux, a hierarchical, graph-based resource management and scheduling framework, was created to solve portability, throughput, and scheduling challenges faced by complex scientific workflows running at exascale [16]. The Flux Framework features rich application programming interfaces (APIs) and a modular design that facilitates integration into cloud technologies [16], [24], [25]. Flux and K8s are well-matched technologies that, when integrated, can bring the benefits of converged computing to workloads.

Creating a seamless HPC+K8s converged environment that supports HPC- and cloud-oriented components of workloads requires K8s to adopt characteristics of HPC, and HPC those of the cloud. Large, dynamic MPI ensemble-based complex workloads running in static HPC allocations can experience long startup times, leading to resource under-utilization [17]. The ability to dynamically increase the size of an HPC resource allocation and improve efficiency through autoscaling remains a highly desirable [17] but an unrealized goal. There have been recent advancements in the ability to run MPI-based workloads at scale using the automation of cloud-native orchestration in K8s [26], [27], but unlocking efficiencies of elasticity also requires autoscaling capability in HPC. In this study, we develop and implement a workload-driven autoscaling algorithm that adjusts the number of nodes running an ensemble based on the length of a job queue and the application median runtime. We perform a series of experiments to measure the performance, overheads, and costs of running ensembles of four well-known MPI-based proxy applications under static and autoscaled resources via the Flux Operator running on Elastic Kubernetes Service (EKS) in Amazon Web Services (AWS).

Specifically, we make the following contributions:

- Develop a workload-driven autoscaling algorithm to change the number of nodes running an MPI-based ensemble workload
- Analyze end-to-end runtime and dollar cost of ensembles in three configurations— static allocation, full autoscaling, and workload-driven autoscaling
- Demonstrate that workload-driven autoscaling costs less and reduces ensemble runtime in comparison to fully automatic autoscaling and enables a trade-off option that balances cost and runtime
- Measure and analyze overheads of operations such as cluster creation and resource scale-out operations

II. Background

A. Flux Framework

Emerging scientific workflows present throughput, portability, co-scheduling, and job coordination challenges that cannot be addressed by current HPC resource managers and schedulers [16]. The Flux Framework [28] supports hierarchical resource management, which allows it to instantiate external resource manager instances (or itself) for portability [28] and high throughput [16]. Its modular architecture and APIs facilitate integration with the cloud [25], [26]. Additionally, the Flux Framework's directed graph-based resource model, sophisticated scheduling policies, and resource management algorithms provide flexible representation to enable workloads to run efficiently.

Flux follows a leader-follower architecture. A Flux broker is a distributed message broker that runs in each cluster node. A single dedicated leader is the root of a tree-based overlay network to which other follower brokers connect. The brokers can self-identify their roles via a shared system configuration file. This setup works equivalently on HPC nodes as it does on "nodes" in the cloud, which are typically virtual machine instances. Thus, in this paper, we use the terms "node" and "instance" interchangeably.

B. Kubernetes

There are several compelling reasons behind K8s' widespread adoption compared to other orchestration solutions. Its declarative management approach allows users to define the desired state, and K8s automatically maintains that state and recovers from failures. Although K8s typically runs as an infrastructure as a service in the cloud, it can be deployed on premises. We describe several K8s components relevant to the study in the following sections.

1) Pods

A Pod [29] is a fundamental component of application deployment, grouping one or more containers. These containers share storage and network resources. To ensure isolation between Pods, Kubernetes utilizes Linux namespaces and cgroups.

2) Horizontal Pod Autoscaler

The K8s Horizontal Pod Autoscaler (HPA) [30] is a dynamic control mechanism that adjusts the number of Pod replicas in an application deployment or replica set based on specified metrics or resource utilization. It periodically evaluates resource metrics such as CPU and memory utilization or custom metrics. It then automatically adjusts the number of Pod replicas within a deployment or replica set to maintain a specified resource utilization target.

3) Cluster Autoscaler

The Cluster Autoscaler (CA) [31] is a tool that automatically adjusts the size of a K8s cluster based on resource demands. It increases the cluster size when resource shortages are causing Pod scheduling failures and decreases it when nodes are consistently underutilized, ensuring resource utilization and improving efficiency. Cloud providers provide cluster autoscalers for their respective resources, including AWS.

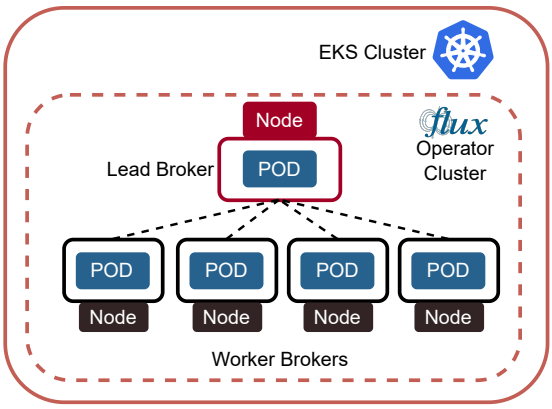


Fig. 1: Example of HPC with cloud computing convergence used in our study. The Flux Operator is deployed in a Kubernetes cluster, which deploys an HPC cluster in K8s. The K8s cluster is managed by, e.g., AWS.

The CA operates by first identifying the autoscaling group designated for scaling instances. It constantly monitors the K8s cluster for pending Pods. When it identifies pending Pods, it assesses whether the autoscaling group's configuration can accommodate them, and if so, requests the cloud provider to increase the number of instances.

C. Convergence of Flux and K8s

A K8s operator is a controller that uses domain-specific knowledge to automate the entire lifecycle of complex applications within the K8s ecosystem. The Flux Operator [32] was developed to deploy an entire Flux cluster on demand. When deployed in Kubernetes, this Custom Resource is called a MiniCluster. Each K8s Pod running a Flux broker is mapped to a node, and the network to connect the nodes is provided by a K8s Headless Service. Pods are mapped one-to-one to K8s nodes to manage hardware resources efficiently. Several features distinguish the Flux Operator from other MPI operators, such as interactive modes, queue monitoring, bursting to external resources, and, most importantly, elasticity (ability to expand/retract resources dynamically). The Flux MiniCluster can scale due to the Scale Subresource [33], which increases the size of the underlying indexed Job that comprises the cluster nodes. When new follower brokers appear (running pods) and register with the lead broker, the Flux workload manager registers them as up nodes from a down state. This ability to grow in size is essential for both autoscaling strategies described in this work. Due to its unique design and utilization of K8s resources efficiently, the Flux Operator consistently outperforms the MPI operator [34] by completing the execution of an application at least 5% faster [32]. Figure 1 shows our adaptation of the converged computing utilizing the Flux Operator and K8s.

D. Public Cloud

Public cloud providers, such as AWS, Google Cloud Platform (GCP) [35], Microsoft Azure [36], and IBM Cloud [37] offer scalable and elastic infrastructure as a service (IaaS), enabling access to compute resources, storage, and services on-demand. The scale and elasticity of the cloud model poses a challenge for the connectivity needs for HPC applications.

Unlike HPC, cloud clusters are often geographically or spatially distributed. We run our experiments on AWS, which uses the Elastic Fabric Adapter (EFA) [38]–[40]. EFA provides lower latency communications for HPC applications running across instances. EFA uses a specialized operating system bypass mechanism to enable HPC applications to scale on AWS [41]. To improve performance, AWS suggests that EFA be used with a placement group. Placement groups [42] ensure low-latency communication by launching instances within the same subnet, binding them to the same Availability Zone [43], reducing data transfer time [44].

E. MPI-based Ensembles

Ensembles are workloads composed of similar runs of an application, differing by initial conditions or parameters and often based on MPI. Ensembles can number tens [14], [18] to hundreds of millions [12] of elements or members. Ensembles can have a fixed number of members or be dynamic [6], [10], [17], depending on the input or parameter space to explore. They are frequently part of a larger complex workflow, which may have AI/ML components or data analysis stages that create new ensemble members. Ensemble-based workflows running near exascale can suffer under-utilization due to startup delays caused by dependence on input generation [10], [17]. Ensembles, AI/ML, and data analysis components of emerging workflows can benefit from improving utilization via cloud techniques like autoscaling.

III. Methodology

Enabling elasticity in MPI-based ensemble workflows poses unique challenges. For example, cloud autoscaling solutions are designed for microservices with variable resource utilization and dynamic traffic. However, autoscaling solutions designed for microservices are impractical for HPC applications, which often exhibit 100% CPU utilization. Transitioning HPC to the cloud requires understanding various cloud features such as instance acquisition time or the time to pull large container images. Additionally, setting up an HPC cluster in the cloud is complex, necessitating automation. In the next section we address these challenges and discuss our contributions to the autoscaling of ensemble workloads and the performance analysis on running HPC in the cloud.

A. Autoscaling

We studied the benefits of elasticity on MPI-based ensemble workloads using two approaches. First, we examined the default autoscaling behavior of K8s with the Flux Operator and MPI-based ensemble workloads. Then, we developed a workload-driven autoscaling approach where the workload manager can launch instances based on parameters such as the job queue size, time to get instances from the cloud, and the median job completion time. We compared these mechanisms' time and dollar costs with the baseline approaches of no autoscaling.

Figure 2 shows the overall system flow diagram of the two autoscaling approaches. A K8s cluster with a fixed number

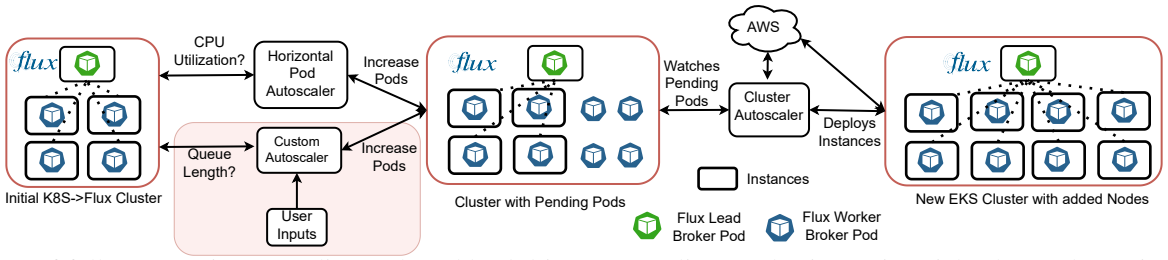


Fig. 2: Flow of fully automatic autoscaling and workload-driven autoscaling mechanism. Given jobs that each require 4 nodes, a fully automatic strategy (top) scales based on CPU utilization, often resulting in extra pods that cannot be utilized. A workload-driven mechanism (bottom and marked area) adds nodes that fit job requirements exactly.

nodes is deployed. Then the Flux Operator deploys the application container to a pod on each node, adding Flux to create a MiniCluster. The Flux MiniCluster is responsible for maintaining the Flux Operator Pods cluster's size and deploying one Pod per physical instance to maintain the one-to-one mapping. Then, the MPI-based ensemble application is launched. Each ensemble task (also referred to as a job) is submitted to the Flux *lead* broker. The jobs are scheduled by the Flux scheduler based on resource availability. If resources are unavailable, jobs have to wait in the queue.

1) Fully Automatic

This setup is fully automatic because no intervention or external input is required for the autoscaling decision. Fully automatic autoscaling involves the integration of two crucial components within the K8s orchestration system: the Horizontal Pod Autoscaler (HPA) [30] and the Cluster Autoscaler (CA) [31].

HPA tracks the CPU utilization of all Pods managed by the Flux Operator and increases the number of Pods to maintain a CPU utilization target. As Flux Operator Pods occupy the entire resources of one node, the newly deployed Pods stay pending due to the unavailability of the instances [32].

The CA considers the requirements of the pending Pods and finds matching instances from AWS that can host these Pods. If it finds matching instances satisfying user-provided constraints, it launches the instances from AWS by communicating via the AWS cloud API.

Once these new nodes are successfully integrated into the K8s cluster and become visible to the K8s control plane, the pending Pods are scheduled onto these nodes. Finally, the Flux follower brokers are initiated on the newly added nodes, allowing Flux to recognize and seamlessly incorporate these nodes into its cluster. The Flux lead broker can then launch pending jobs from its queue.

2) Workload Driven

For our workload-driven mechanism, we devised an algorithm that determines required resources rather than relying solely on automated response to CPU utilization. Figure 2 shows the steps involved in the full and workload-driven autoscaling setup. In the case of workload-driven autoscaling, instead of relying on HPA, we utilize custom metrics and algorithm 1.

Algorithm 1 runs one loop when a job completes (Line 1). The algorithm examines the Flux resource manager queue depth and considers the median duration of job execution in the ensemble and the time taken to provision AWS instances as input. Then, the algorithm calculates whether it is appropriate to acquire a new node and, if so, how many nodes should be requested. The algorithm also takes into account the jobs that will finish before acquiring new instances and adjusts pending jobs accordingly (Line 9). Note that the algorithm calculates the total new nodes according to job requirements and instance acquisition time. A new node is added if the time it takes to provision the node is less than the time needed to wait for an existing node to become available. Finally, the algorithm also considers a maximum budget and returns the lower of the calculated and maximum allowed node counts.

Workload-driven autoscaling creates the number of Pods based on these calculated values returned by the algorithm. Recognizing the existence of pending Pods, the cluster autoscaler responds by provisioning new EC2 instances, aligning with the demands signaled by the pending Pods. Once the new instances are added to the cluster, the pending Pods are scheduled, and the Flux follower brokers in those Pods start executing and connecting to the lead broker. Once the

Algorithm 1 Workload-Driven Autoscaling

Input: Median job runtime X, job resource requirement R, instance acquisition time I, maximum number of instances N_{max}

Output: Scaling Operation, num. nodes N at job completion event t (UPSCALE/DOWNSCALE/NOACTION, N_t)

1: **for** each job completion event t **do**
2: Get no. jobs in the queue J_t, current no. nodes N_t
3: No. nodes required for queued jobs $R_t = J_t \times R$
4: **if** $R_t < N_t$ **then**
5: DOWNSCALE by $N_t - R_t$
6: **else if** $R_t == N_t$ **then**
7: NOACTION, N_t
8: **else**
9: Consider jobs that will finish before I
10: Update num. nodes required $R'_t = R_t - \lfloor \frac{I \times N_t}{X \times R} \rfloor$
11: **if** $R'_t > N_t$ **then**
12: UPSCALE by $N_t + \min(R'_t, N_{max} - N_t)$
13: **end if**
14: **end if**
15: **end for**

scheduler detects newly available resources satisfying job requirements, the resource manager starts queued jobs on the resources.

3) Study of Performance Overhead

Along with developing an autoscaling strategy for ensemble workloads, we also investigated the unique issues that can arise when running HPC applications in a converged computing setting. For example, with elasticity, when we request instances from the cloud there is a waiting time to provision the instances. Moreover, we also need to understand how container pulling impacts application launch time and incurs additional costs. Moreover, the container images built for HPC applications can be larger than traditional cloud service-oriented applications. Requesting hundreds of them from a central registry at the same time may create a bottleneck.

To automate the deployment of cloud infrastructure and measure these times, we used a tool called "kubescaler" that uses the Kubernetes API and automates the deployment of clusters, controls resource elasticity, and records timings of various events that are important when running MPI-based ensemble in the cloud [45]. The tool automates the deployment of an EKS cluster to aid in performing the scalability study of EKS.

B. Selection of MPI-based ensemble

For our evaluation, we used four benchmarks from the CORAL2 [46] suite: LAMMPS [47], AMG [48], [49], Kripke [50], and Laghos [51], [52]. CORAL-2 benchmarks have been designed specifically for performance analysis of upcoming supercomputers. LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator) [47] is a classical molecular dynamics (MD) code that models a large set of particles. The application can be run in serial mode, in distributed parallel mode with MPI, in multithreaded mode with OpenMP, and in hybrid MPI and OpenMP mode. AMG [48], [49] is a parallel multigrid solver that solves linear systems on unstructured grids. Kripke [50] is a scalable 3D deterministic particle transport code designed to study the impact of data layout, programming paradigms, and architectures on implementation and performance. Laghos (LAGrangian High-Order Solver) [51], [52] is a mini-application designed to solve the time-dependent Euler equations for compressible gas dynamics.

We created an ensemble workload by running 20 identical jobs of each application. These jobs are identical in their resource requests and in the initial conditions or parameter space they explore. We also ran a larger ensemble with 100 jobs and an ensemble with varying application problem sizes to study the effectiveness of workload-driven autoscaling on ensembles with a greater number of jobs and variable job runtimes, respectively.

C. Determining Utilization Threshold

To determine the appropriate target CPU utilization for HPA in a fully automatic setup, we conducted a simulation considering that HPC applications typically maintain a consistent CPU utilization close to 100%. Figure 3 shows the corresponding new instance counts for desired utilization thresholds. By varying the target CPU utilization, it was observed that starting above 50% was necessary to prevent unbounded behavior. When the desired metric value was set below 50%, the HPA continuously doubled instance counts due to the rapid job launches on newly added nodes, resulting in a repetitive cycle. Conversely, setting the metric to 100% prevented HPA from adding instances. As a balanced approach, a target metric value of 80% was chosen for the experiments. Moreover, our experiments show that the target metric value is also independent of application end-to-end completion time.

TABLE I: Node and Cluster Setup

Module	Attributes
Kubernetes Version	V1.27
eksctl version	base - v0.168 with customization for ARM
EFA	enabled
Placement Group	enabled
Instance CPU	64 ARM cores
Threads Per Core	1
Instance Memory	128 GB
Instance Network	200 Gigabit
Initial Cluster Size	8 nodes
Total Ranks	512
Ranks in each node	64

IV. Experimental Setup

A. Autoscaling Study

This section describes the experimental design for testing a novel autoscaling strategy in the context of deployment overhead such as time to completion and cost. For each of a suite of MPI applications, we create a cluster that is either static (no autoscaling) or dynamic (autoscaling). For each cluster we submit a set of jobs and allow the jobs to run given static resources or with an autoscaling strategy. We compare our workload-driven autoscaling approach against CPU utilization-based autoscaling and static cluster sizes.

1) Cluster Creation

Both static and dynamic EKS clusters are created using the `eksctl` [53] command line tool. We measured the cluster creation times for all autoscaling study experiments. Once the cluster and associated resources are available, we set up the Flux Operator using the kubectl command line tool. This deploys a "MiniCluster," an entire HPC cluster (Flux Framework) in K8s with lead and follower brokers connecting pods [32]. The job submission of each experiment is performed from the lead broker pod. For a static cluster the size (8, 16, 32, or 64 nodes) is constant. A dynamic or autoscaling cluster starts with a preliminary size of 8 nodes that is then scaled up or down depending on application needs.

2) Ensemble Job Design

An ensemble is composed of a group of jobs, where each job occupies a specific number of nodes. For all of our experiments, we chose a job size of eight nodes as a baseline configuration. For these experiments, we used `hpc7g.16xlarge` nodes with 64 CPU cores. This choice was driven by cost and temporal feasibility – obtaining hundreds of nodes in the

Fig. 3: HPA calculates new instances based on desired utilization value and current utilization

cloud would have been time-consuming and costly. Thus, this choice is practical, and strikes a balance between representing an HPC environment and managing costs effectively. For the dynamic, autoscaling clusters, we chose to match the initial cluster size to the job size to ensure greater resource utilization at the onset of creation of a cluster. The workload-driven and fully automatic setups add more instances to the eight nodes. Additional details of the node and cluster setup are available in Table I.

3) Application Setup

Applications, autoscaling strategies, iterations, problem sizes, and parameters are detailed in Table II. Problem sizes were chosen explicitly to achieve a desired runtime of 2-3 minutes for each job, and exact median runtimes were recorded for each application to inform the algorithm. For AMG, the largest possible problem size that would run on the `hpc7g.16xlarge` instance without running out of memory resulted in a runtime comparable to LAMMPS. Parameters were chosen for Laghos that ensured a longer runtime to introduce runtime heterogeneity in the experiments.

For this set of experiments, each application was run at least 20 times. A time span of a few minutes is short enough to run substantial repetitions to measure variability, but not too long to add cost without additional benefit to our analysis. The workload-driven autoscaling algorithm (Algorithm 1) considers the median job runtime and the time it takes to acquire new instances from the cloud, and longer median runtimes do not impact the algorithm behavior. Further, longer runtimes are more susceptible to cloud maintenance and failures. Each job is launched with 64 MPI ranks for each of 8 nodes, resulting in 512 ranks in total.

To test the applicability of the workload-driven algorithm to larger ensembles and varying ensemble member runtimes we ran AMG experiments with an increased number of iterations and varying problem sizes. These experiments are "Larger Ensemble" and "Runtime Variability," respectively (see Table II). For the varying problem sizes, we generated parameters by sampling from a normal distribution, imposing an upper bound on the distribution to ensure successful job execution given instance memory constraints. As an example, we set the mean to 150 for a particular job parameter with a standard deviation of 5 and then constrained the values to fall within the range of 140 to 160 by setting the lower and upper bounds. This design is important to the study because ensemble jobs can exhibit variability in problem sizes or parameters, leading to varying runtimes.

4) Pods and Container Timings

The time to pull an application container from a registry until the time it can run an application is an important consideration when assessing experimental costs. Further, in the context of autoscaling when containers must wait for a node allocation to be scheduled, pending time is important to assess. We used the Kubernetes API to record pod events (creation, scheduling, and ready) during all repetitions of the automatic and workload-driven experiments. With the timing event data we can determine the time it takes for a Pod to be scheduled, then be pulled from the registry and to enable it to run. Once all these steps are completed, the new Flux broker Pods join the lead broker of the MiniCluster.

5) Autoscaling Protocol

For each experiment described in Table II, the cluster is created and the Flux Operator deployed (IV-A1), and the set of jobs (IV-A2) are submitted to run on the Flux MiniCluster. For a static cluster, the jobs are allowed to run to completion with no additional intervention. When autoscaling with the CPU utilization-based (fully automatic) strategy, the cluster autoscaler adds nodes in response to pending Pods based on CPU-utilization. The workload-driven autoscaler uses a custom Flux metrics API [54] to change the number of nodes based on Algorithm 1. Within the fully automatic setup, we opt for a target CPU utilization of 80% averaged across all pods (Section III-C). We run five repetitions for each of the static and fully automatic experiments, and three repetitions for each workload-driven experiment. Each experiment progresses until all ensemble jobs are completed, and then the cluster is deleted.

To measure the impact on cost of downscaling with the workload-driven algorithm we ran two ensembles of 20 AMG jobs with and without downsizing. Workload-driven autoscaling with downsizing enabled allows reduction of the number of nodes when those nodes are no longer running jobs. Without downscaling, the nodes persist and continue to incur costs until the cluster is terminated when all jobs are complete.

B. Scaling Study

We conducted a scaling study to assess the time and cost overhead of cluster creation and deletion.

1) Cluster Creation

The kubescaler tool [45] provides a means to create and scale Kubernetes clusters on AWS (EKS) and Google Cloud. It provides a Python SDK that leverages the AWS boto3 API for automating the process [55], which is similar to `eksctl` in that it also uses AWS CloudFormation, an infrastructure-as-code service, to model, provision, and manage AWS and third-party resources.

Kubescaler initially creates a CloudFormation stack to establish a Virtual Private Cluster (VPC), which provides a logically isolated virtual network for launching resources [56].

TABLE II: Autoscaling Experiments *S (static), F (fully automatic), W (workload-driven)*

Experiment Name	Application	Autoscaling-Strategy	Ensemble Size	Parameters
Autoscaling Study	LAMMPS	S8 S16 S64 F W	20	(problem size) `64x16x16`
	AMG	S8 S16 S64 F W	20	(problem size) `160x145x70`
	Kripke	S8 S16 S64 F W	20	`--groups 500 --zones 64,64,64 --procs 16,8,4`
	Laghos	S8 S16 S64 F W	20	(mesh) `-m cube_211_hex.mesh` (solver and mesh refinements) `--ode-solver 7 -rs 4 -rp 1` (steps) `--max-steps 160`
Larger Ensemble	AMG	S8 S16 S64 F W	100	(problem size) `160x145x70`
Runtime Variability	AMG	S8 S16 S64 F W	20	(problem size) `varying (Section IV-A3)`

Subsequently, another CloudFormation stack is created to deploy a K8s cluster. Finally, a third CloudFormation stack (a worker stack) is created to set up a managed nodegroup responsible for managing EC2 instances for K8s. When creating the worker stack, it generates an AWS Auto Scaling Group (ASG) in the background and maintains the desired number of instances. To scale the instances of the K8s cluster, the user can adjust the desired size of the CloudFormation stack. The CloudFormation stack then updates the size of the underlying AWS ASG, which in turn updates the EC2 instances. As the managed nodegroup is a part of the EKS cluster, the nodes are launched, and the K8s controller can locate them. During this process, Kubescaler keeps timings in seconds for each operation. These timings include the creation of instances in AWS during CloudFormation stack updates, the time required for these instances to become ready to accept Pod requests from Kubernetes, and the completion of CloudFormation stack updates.

2) Scaling Operations

In conducting the scale-out operations, we used the `hpc6a.48xlarge` instance type with 96 cores and 384GB memory to match our autoscaling experiments. After cluster deployment (Section IV-B1) we systematically increased the number of instances in increments of 4 nodes until a maximize size of 64 nodes, measuring the time of each incremental scaleout. We repeated this procedure with increments of 8 and 16 nodes (each up to 64 nodes), providing us with cluster creation, deletion, and scaling times for all increments. Understanding how long these processes take is an important factor when designing an autoscaling experiment.

V. Results

In this section we present results of our performance studies of running MPI-based ensembles in the cloud. We assessed the temporal and monetary cost of autoscaling ensembles (Section V-A), along with cluster creation and deletion times (Section V-B). You can find all of our data and scripts to reproduce the experiments here: https://doi.org/10.5281/zenodo.13247408 [57].

A. Autoscaling Study

1) Comparison of Autoscaling Strategies

Figure 4(a) shows the end-to-end runtime of all ensemble members for each of the five strategies. Median runtimes based on application configurations and resources were determined to be 116, 110, 128, and 179 seconds each for each of LAMMPS, AMG, Kripke, and Laghos, respectively. The workload-driven autoscaling ensemble completion time is 4× less than the fully automatic time for AMG, and 4.7× less for Laghos. The fully automatic configuration adds node counts that are incompatible with the job requirements. For example, adding seven nodes when eight are needed causes underutilization because seven nodes cannot be used for jobs. The workload-driven strategy adds an integer multiple of the per-job node count requirement, ensuring that enqueued jobs will be started on the resources. Adjusting the algorithm to account for downscaling produces similar end-to-end median runtimes (382.37 seconds) as without downscaling (383.97 seconds).

TABLE III: EKS cluster creation times (seconds) by size

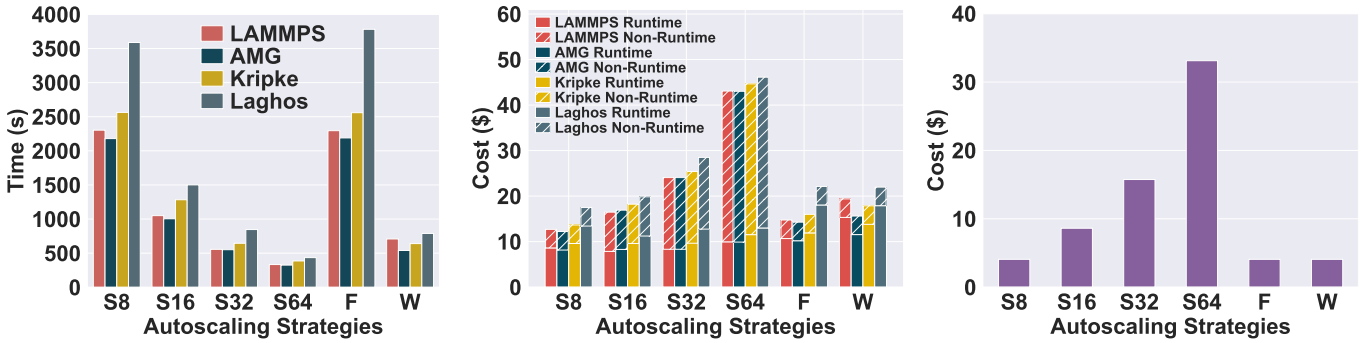
Cluster Size	Median	Min	Max	Repetitions
8	1087.57	957.08	1230.02	14
16	1151.66	1009.05	1261.20	5
32	1052.12	1033.091	1180.450	4
64	1106.82	971.56	1212.55	4
128	1050.46	1010.97	1119.64	3
256	1280.22	1257.55	1300.012	3

Timings in seconds to create clusters with initial instance sizes of 8-256. Timings of size 8-64 were measured as part of the autoscaling study. Size 128 and 256 tests were run to measure larger cluster creation time and did not include autoscaling or application runs.

Table III shows the median, minimum, and maximum times required to create EKS clusters ranging from sizes 8 to 256 nodes. The median time required to launch a cluster does not depend on the number of instances up to 128 instances, and increases by 22% from 128 to 256 instances. Figure 8(a) illustrates the time required for creating an EKS cluster and its associated resources, while Figure 8(b) shows the timings for deletion and associated resources.

2) Cost

Figure 4(b) illustrates the median costs of total end-to-end runtime and non-runtime for various autoscaling strategy. For LAMMPS, the median runtime cost of the static cluster with 8 nodes was $8.62; the workload-driven strategy was $15.32, and the fully automatic scaling $10.68. The static cluster with 64 nodes produces the lowest total ensemble completion time but incurs the highest cost. For our experiments, the largest contributor to the costs are non-runtime costs, as shown in Figure 4(c). The larger the initial cluster size, the greater non-runtime cost incurred. The workload-driven approach incurred only a 1.43× more cost for a 3.24× lower total completion time than the full autoscaling strategy in LAMMPS. With downsizing enabled, we can save as much as 20% cost over

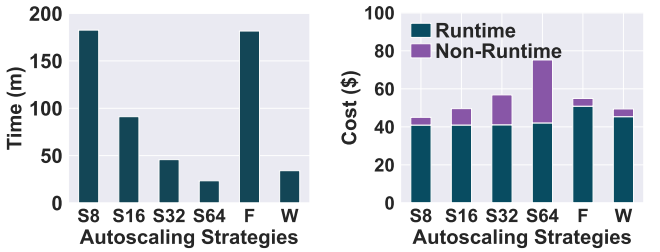


(a) Ensemble end-to-end runtime.

(b) Total ensemble costs including non-runtime and runtime

(c) Non-runtime costs

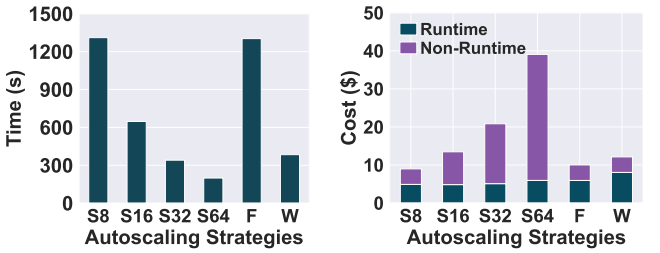
Fig. 4: Comparison of MPI ensembles end-to-end runtime and costs in various autoscaling strategies for each of a specific size of static (***S***), fully automatic (***F***), and workload-driven (***W***) setups with 3 repetitions. End-to-end runtime is fastest across applications for the static size 64 and workload-driven setups (a), however costs are highest for the same static size 64 setup (b), primarily resulting from the non-runtime costs for the setup (c).



(a) End to end runtime

(b) Total costs with runtime and non-runtime

Fig. 5: Total cost and end-to-end runtime of 100-member ensemble of AMG jobs with fixed resource requirements. The majority of cost across strategies is incurred during setup, and the temporal pattern (a) is consistent with smaller ensemble runs in Figure 4(a).



(a) End to end runtime

(b) Total costs with runtime and non-runtime

Fig. 6: End-to-end runtime (a) and total cost (b) of 20 runs of AMG with variable job sizes and fixed resource requirements. Parameters are generated from a normal distribution.

workload-driven autoscaling without downsizing.

3) Larger Ensemble

We assessed the performance of our workload-driven autoscaling algorithm for larger MPI ensembles. Figure 5(a) shows the end-to-end runtime of ensembles with 100 members, whereas figure 5(b) shows the corresponding cost of running the larger ensemble and non-runtime cost of the cluster, respectively. Figure 5(a) demonstrates that workload-driven autoscaling outperforms static (except for static-64) and fully automatic setup in terms of runtime and incurs only a small percentage more cost than the static-8 setup.

4) Runtime Variability

Figure 6(a) illustrates the overall costs of running an ensemble of jobs with greater runtime variability, which is controlled by varying the problem size (Section IV-A2). While there is no strategy that is both fastest and lowest cost, workload-driven autoscaling offers a balance between end-to-end completion time and cost, a trade-off that we will discuss in Section VI.

5) Pods and Container Timings

We observed no significant differences in median registry pull times between workload-driven autoscaling (50.1 seconds) and fully automatic strategies (50.02 seconds). However, the median time for pending Pods was lower for workload-driven autoscaling (155.11 seconds) compared to fully automatic (158.27 seconds), with greater variability (standard deviation - 11.85 and 3.28 seconds, respectively). The workload-driven setup results in greater variability because the autoscaler system requests a larger number of instances simultaneously, leading to longer wait times for some instances to become available.

B. Scaling Study

1) Cluster Operations

The operations in this experiment start with creating the VPC and configuring subnets, followed by cluster creation, and conclude with worker stack creation. Figure 8(a) depicts the timing for creating a VPC and worker stack or managed node group. The worker stack creation time is measured to be constant. As seen in Figure 8(a), the median time for cluster creation is approximately 550 seconds.

We also measure the timings of deletion of the same components upon EKS cluster termination. Figure 8(b) displays the

timing for deleting the VPC stack, worker stack, and overall cluster. Similar to resource deployment, resource deletion occurs sequentially. Cluster deletion typically takes less time than creation; however, there are cases when cluster deletion may require more time.

2) Scaling Out Operations

Figure 7 illustrates the timings of scale-out operations under different instance increment sizes. Scaling out a cluster can be done in large increments less frequently or in small increments more frequently. Since each scaling out operation accrues additional time, selecting a scale out increment that matches the job requirements and minimizes the number of scale out operations is suggested.

VI. Discussion

Ensembles composed of MPI-based applications are increasingly common in HPC [6]–[15] and form a large percentage of HPC workloads at large centers like Lawrence Livermore National Laboratory [16]. Ensembles running on static resource allocations can suffer from underutilization due to staged workloads and long startup times. The cloud provides native technologies such as autoscaling and resource dynamism to dynamically manage and provision resources, which can be used to improve the utilization and efficiency of MPI-based ensemble workloads. Autoscaling in the cloud has traditionally used simple metrics like CPU utilization that work well for scaling stateless services. However, it is not clear if CPU utilization accurately accounts for the resource needs of ensembles.

When the needs of a workload, including the number of ensembles, ensemble members, and application parameters are known before runtime, deploying a fixed number of resources that match the exact requirements is sufficient. However, a different strategy is needed for dynamic ensembles, where the amount of work depends on simulation values, statistical properties of output data, or the workload is steered by AI/ML. We describe our key findings in the following subsections.

A. Novel workload-driven autoscaling algorithm

We demonstrate that our workload-driven algorithm outperforms both traditional CPU-based autoscaling strategies, along with different choices of static cluster sizes with four HPC proxy applications. Our results showed that the workload-driven autoscaling outperforms both static and fully automatic autoscaling setups by up to 4.7×, while incurring only 1.43× higher costs compared to the static setup. We reproduced this result with a larger ensemble size and with varying problem sizes, showing the extensibility of the algorithm. While our study focused on running MPI-based ensembles, this general pattern can be extended to any kind of application that requires a group of nodes to perform coordinated work concurrently, which also includes AI/ML and cloud-native workloads.

B. Analysis of end-to-end runtime and costs

Another purpose of our study was to understand the overhead of necessary operations for running MPI-based ensemble applications in the cloud with elasticity. We measured the time required to launch an EKS cluster with varying numbers of instances *and found that the creation time does not depend on the requested number up to 128 instances, and increases by 22% from 128 to 256 nodes*. Additionally, we observed that there is a waiting period to obtain instances from AWS, and to pull and execute containers. Consequently, it is essential to design a scale-out policy strategically to maximize utilization and minimize instance costs. Due to the unknown contributions to non-runtime costs, it is essential for the community to measure, analyze, and understand the component costs of deploying workloads on cloud infrastructure.

C. Determining the impact of scale-out strategies

In a workload-driven autoscaling setup, we have the option of selecting scale-out strategies, such as scaling out with fewer instances more frequently or scaling out with more instances less frequently. We observe that scaling out by 16 instances proves to be more effective because the less frequent scaling incurs lower waiting times and thus translates to lower cost.

However, scaleup may result in under-utilization of allocation when some of the additional instances are not needed. To address this, we introduced downscaling (scaling in) to our workload-driven autoscaling algorithm. This addition saved us more than 20% in costs compared to the prior version. Based on our findings, we conclude that requesting more instances less frequently is more cost-effective.

VII. Related Work

A. Autoscaling HPC Applications

Dynamic autoscaling of high-performance applications is a complex task that involves challenges on multiple fronts. Typically, HPC applications are designed to use a fixed number of resources that can not utilize elasticity provided by the cloud [58], [59]. Elasticity enabling API extensions with MPI V2.0 are proposed and applied to an application with a prototype solution in [59]. Other approaches involve checkpoint-restart mechanisms that lead to application termination and redeployment, which require data redistribution and impact performance [58]. Reinit [60] and Reinit++ [61] support global-restart recovery and ULFM [62] provides MPI fault-tolerance, but these techniques do not support full elasticity.

Substantial effort has been put into autoscaling workflows and dynamic resource management of workflow stages. Liu et al. [63] propose an autoscaling framework for ML workflows. Their work demonstrates that application performance can be gained for ML workflows through auto-tuning the horizontal Pod autoscaler threshold by monitoring and utilizing application run-time metrics like fluctuations in CPU utilization [63]. However, as we demonstrated, such CPU utilization-based auto-tuning mechanisms are ineffective in the HPC domain since these applications are designed to maintain CPU utilization close to 100%. The paper [64] presents a novel autoscaling strategy designed for scientific workflows in cloud computing environments. The strategy leverages spot instances for cost efficiency and employs a heuristic scheduling method to optimize makespan while mitigating the impact of out-of-bid failures.

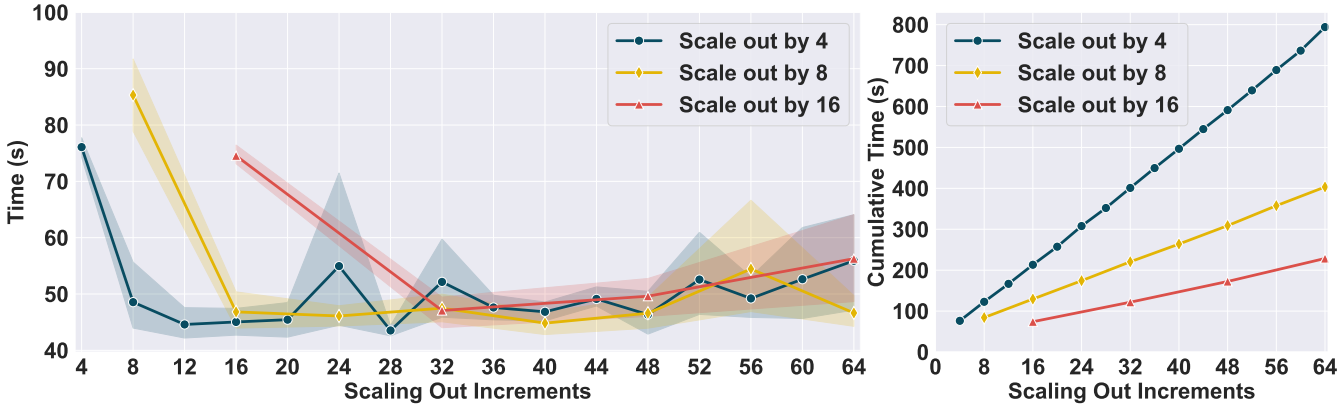
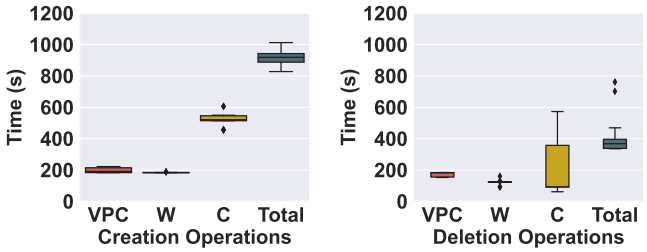


Fig. 7: Time in seconds to increase the cluster to total nodes (x-axis) by increments of 4 (blue), 8 (yellow), and 16 (red).



(a) Cluster creation Timings (b) Cluster deletion Timings

Fig. 8: Timings of creating (a) and deleting (b) EKS clusters and supporting stacks. "**W**"-worker stack, "**C**"-cluster creation

Finally, the KubeFlow project [65] provides the MPI Operator [34] and other necessary tools to facilitate MPI-based applications. The MPI Operator bootstraps MPI but does not provide queuing, resource management, or scheduling and struggles to scale [27].

B. Autoscaling in the Cloud

There is much work on addressing resource management and autoscaling of cloud applications. Some of the work focuses on improving the horizontal autoscaling of web applications [66]–[70], and some of it focuses on resource allocation and quality of services [71]–[81]. K8s provides a horizontal autoscaling mechanism based on the utilization threshold of various metrics such as CPU or Memory [30]. The authors of [67] also propose a rule-based autoscaling mechanism based on CPU and memory utilization. The author of SHOWAR [68] uses the variance in historical usage for vertical scaling and a proportional-integral-derivative (PID) controller for horizontal scaling. Nonetheless, SHOWAR still requires extensive tracing from the CPU scheduler for its scaling decision. However, threshold-based approaches work best if there is a variable amount of resource utilization of the applications and some relationship between utilization and performance. HPC applications do not show such behaviors.

These works [72], [73], [75], [82] proposed various techniques and frameworks to find the efficient resource allocation and quality of services for cloud applications. They used machine learning and iterative techniques to identify the root cause of performance bottlenecks and find the optimum resources for iterative applications. The above approaches are explicitly designed for microservices and can not be readily applicable to MPI-based ensemble workflows.

Recently, AWS developed a cluster autoscaler module called Karpenter [83] to assist autoscaling nodes for the EKS cluster. Karpenter provides several advantages over the cluster autoscaler, such as supporting a full range of instance types across availability zones and providing instances directly from AWS without a managed node group. While Karpenter could have been used to perform autoscaling, the ability to extend our work to other cloud providers was important. We choose not to use Karpenter because it did not have support for additional providers at the onset of this work.

To the best of our knowledge, we are the first to enable autoscaling for MPI-based ensemble workflows.

VIII. Conclusion & Future Work

The work performed in this study provides valuable insights for running MPI ensembles in the cloud. We started with a series of comprehensive tests to measure the costs of deploying, managing, and running ensembles of MPI-based applications within AWS EKS clusters. To improve upon traditional approaches toward CPU utilization-based autoscaling, we developed and implemented a workload-driven scaling algorithm that reduced end-to-end runtime by 3-5×. To inform the sizing strategy for scaling out a cluster, we studied the costs and timings of resource scale-out based on different node increments. While cloud infrastructures and cost models change rapidly, our workload-driven scaling algorithm, test methodology, and data presented here provide valuable insights to inform future improvements in cloud workload deployment. Since cloud technologies and their cost models evolve rapidly, it is essential to understand the interaction between HPC workload deployments and cloud infrastructure costs.

Our findings offer guidance to engineers, developers, and scientists running HPC workloads in Kubernetes environments, but also emphasize the need for resource and allocation elasticity in traditional HPC to reap the same benefits. Efficient resource provisioning and workload-driven resource scaling are essential for improving performance and cost-effectiveness.

In future work, we will focus on enhancing the workload-driven autoscaling algorithm by introducing machine learning techniques to adapt and update these algorithm parameters dynamically. We will improve upon the deployment and autoscaling strategies performed here by testing advanced metrics such as inter-job dependency representations, dynamic job lengths, variable resource requirements, and variable instance acquisition patterns by AI/ML. Advanced analysis techniques can expose further efficiencies and increase the performance of complex, dynamic workflows composed of traditional HPC applications as well as cloud services and integrated AI/ML.

ACKNOWLEDGMENTS

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-864304) and was supported by the LLNL-LDRD Program under Project No. 22-ERD-041 and US National Science Foundation under grants ECCS-2152357 and CCF-2324915. We wish to thank Evan Bollig and Heidi Poxon of AWS for their help deploying MPI workloads on EKS with EFA. We also wish to thank AWS for providing credits that enabled our experimental work. We thank Jae-Seung Yeom for providing constructive feedback to improve the manuscript.

REFERENCES

[1] J. S. Vetter, R. Brightwell, M. Gokhale, P. McCormick, R. Ross, J. Shalf, K. Antypas, D. Donofrio, T. Humble, C. Schuman, B. Van Essen, S. Yoo, A. Aiken, D. Bernholdt, S. Byna, K. Cameron, F. Cappello, B. Chapman, A. Chien, M. Hall, R. Hartman-Baker, Z. Lan, M. Lang, J. Leidel, S. Li, R. Lucas, J. Mellor-Crummey, P. Peltz Jr., T. Peterka, M. Strout, and J. Wilke, "Extreme Heterogeneity 2018 - Productive Computational Science in the Era of Extreme Heterogeneity: Report for DOE ASCR Workshop on Extreme Heterogeneity," 12 2018.

[2] S. H. Langer, B. Spears, J. L. Peterson, J. E. Field, R. Nora, and S. Brandon, "A HYDRA UQ Workflow for NIF Ignition Experiments," in *2016 Second Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV)*, pp. 1–6, 2016.

[3] D. Wang, X. Luo, F. Yuan, and N. Podhorszki, "A Data Analysis Framework for earth system simulation within an In-Situ Infrastructure," *Journal of Computer and Communications*, vol. 05, no. 14, p. 76–85, 2017.

[4] U. U. Turuncoglu, B. Önol, and M. Ilicak, "A New Approach for in Situ Analysis in Fully Coupled Earth System Models," in *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, ISAV '19, (New York, NY, USA), p. 6–11, Association for Computing Machinery, 2019.

[5] M. Dorier, J. M. Wozniak, and R. Ross, "Supporting Task-Level Fault-Tolerance in HPC Workflows by Launching MPI Jobs inside MPI Jobs," in *Proceedings of the 12th Workshop on Workflows in Support of Large-Scale Science*, WORKS '17, (New York, NY, USA), Association for Computing Machinery, 2017.

[6] D. H. Ahn, X. Zhang, J. Mast, S. Herbein, F. Di Natale, D. Kirshner, S. A. Jacobs, I. Karlin, D. J. Milroy, B. De Supinski, B. Van Essen, J. Allen, and F. C. Lightstone, "Scalable Composition and Analysis Techniques for Massive Scientific Workflows," in *2022 IEEE 18th International Conference on e-Science (e-Science)*, pp. 32–43, 2022.

[7] D. M. Higdon, M. C. Anderson, S. Habib, R. Klein, M. Berliner, C. Covey, O. Ghattas, C. Graziani, M. Seager, J. Sefcik, *et al.*, "Uncertainty quantification and error analysis," tech. rep., Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2010.

[8] L. Casalino, A. Dommer, Z. Gaieb, E. P. Barros, T. Sztain, S.-H. Ahn, A. Trifan, A. Brace, A. Bogetti, H. Ma, H. Lee, M. Turilli, S. Khalid, L. Chong, C. Simmerling, D. J. Hardy, J. D. C. Maia, J. C. Phillips, T. Kurth, A. Stern, L. Huang, J. McCalpin, M. Tatineni, T. Gibbs, J. E. Stone, S. Jha, A. Ramanathan, and R. E. Amaro, "AI-Driven Multiscale Simulations Illuminate Mechanisms of SARS-CoV-2 Spike Dynamics," *bioRxiv*, 2020.

[9] S. A. Jacobs, T. Moon, K. McLoughlin, D. Jones, D. Hysom, D. H. Ahn, J. Gyllenhaal, P. Watson, F. C. Lightstone, J. E. Allen, I. Karlin, and B. V. Essen, "Enabling rapid COVID-19 small molecule drug design through scalable deep learning of generative models," *The International Journal of High Performance Computing Applications*, vol. 35, no. 5, pp. 469–482, 2021.

[10] F. Di Natale, H. Bhatia, T. S. Carpenter, C. Neale, S. Kokkila-Schumacher, T. Oppelstrup, L. Stanton, X. Zhang, S. Sundram, T. R. W. Scogland, G. Dharuman, M. P. Surh, Y. Yang, C. Misale, L. Schneidenbach, C. Costa, C. Kim, B. D'Amora, S. Gnanakaran, D. V. Nissley, F. Streitz, F. C. Lightstone, P.-T. Bremer, J. N. Glosli, and H. I. Ingólfsson, "A Massively Parallel Infrastructure for Adaptive Multiscale Simulations: Modeling RAS Initiation Pathway for Cancer," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, (New York, NY, USA), Association for Computing Machinery, 2019.

[11] J. L. Peterson, K. D. Humbird, J. E. Field, S. T. Brandon, S. H. Langer, R. C. Nora, B. K. Spears, and P. T. Springer, "Zonal flow generation in inertial confinement fusion implosions," *Physics of Plasmas*, vol. 24, p. 032702, 03 2017.

[12] J. L. Peterson, B. Bay, J. Koning, P. Robinson, J. Semler, J. White, R. Anirudh, K. Athey, P.-T. Bremer, F. Di Natale, D. Fox, J. A. Gaffney, S. A. Jacobs, B. Kailkhura, B. Kustowski, S. Langer, B. Spears, J. Thiagarajan, B. Van Essen, and J.-S. Yeom, "Enabling machine learning-ready HPC ensembles with Merlin," *Future Generation Computer Systems*, vol. 131, pp. 255–268, 2022.

[13] D. H. Ahn, A. H. Baker, M. Bentley, I. Briggs, G. Gopalakrishnan, D. M. Hammerling, I. Laguna, G. L. Lee, D. J. Milroy, and M. Vertenstein, "Keeping Science on Keel When Software Moves," *Commun. ACM*, vol. 64, p. 66–74, jan 2021.

[14] J. E. Kay, C. Deser, A. Phillips, A. Mai, C. Hannay, G. Strand, J. M. Arblaster, S. C. Bates, G. Danabasoglu, J. Edwards, M. Holland, P. Kushner, J.-F. Lamarque, D. Lawrence, K. Lindsay, A. Middleton, E. Munoz, R. Neale, K. Oleson, L. Polvani, and M. Vertenstein, "The Community Earth System Model (CESM) Large Ensemble Project: A Community Resource for Studying Climate Change in the Presence of Internal Climate Variability," *Bulletin of the American Meteorological Society*, vol. 96, no. 8, pp. 1333 – 1349, 2015.

[15] C. Tebaldi, K. Dorheim, M. Wehner, and R. Leung, "Extreme metrics from large ensembles: investigating the effects of ensemble size on their estimates," *Earth System Dynamics*, vol. 12, no. 4, pp. 1427–1501, 2021.

[16] D. H. Ahn, N. Bass, A. Chu, J. Garlick, M. Grondona, S. Herbein, H. I. Ingólfsson, J. Koning, T. Patki, T. R. Scogland, B. Springmeyer, and M. Taufer, "Flux: Overcoming scheduling challenges for exascale workflows," *Future Generation Computer Systems*, vol. 110, pp. 202–213, 2020.

[17] H. Bhatia, F. Di Natale, J. Y. Moon, X. Zhang, J. R. Chavez, F. Aydin, C. Stanley, T. Oppelstrup, C. Neale, S. K. Schumacher, D. H. Ahn, S. Herbein, T. S. Carpenter, S. Gnanakaran, P.-T. Bremer, J. N. Glosli, F. C. Lightstone, and H. I. Ingolfsson, "Generalizable Coordination of Large Multiscale Workflows: Challenges and Learnings at Scale," in *SC21: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–16, 2021.

[18] R. C. J. Wills, Y. Dong, C. Proistosecu, K. C. Armour, and D. S. Battisti, "Systematic Climate Model Biases in the Large-Scale Patterns of Recent Sea-Surface Temperature and Sea-Level Pressure Change," *Geophysical Research Letters*, vol. 49, no. 17, p. e2022GL100011, 2022. e2022GL100011 2022GL100011.

[19] N. C. Thompson and S. Spanuth, "The Decline of Computers as a General Purpose Technology," *Commun. ACM*, vol. 64, p. 64–72, Feb 2021.

[20] Gartner, Inc., "Gartner Says More Than Half of Enterprise IT Spending in Key Market Segments Will Shift to the Cloud by 2025." https://www.gartner.com/en/newsroom/press-releases/2022-02-09-gartner-says-more-than-half-of-enterprise-it-spending, 2022. Accessed: Sept 22, 2023.

[21] The Kubernetes Authors, "Production-Grade Container Orchestration." https://kubernetes.io. Accessed: September 26, 2023.

[22] T. K. Authors, "Kubernetes project dashboard." https://k8s.devstats.cncf.io/d/24/overall-project-statistics?orgId=1. Accessed: 2024-8-5.

[23] "Project journey report." https://www.cncf.io/reports/

kubernetes-project-journey-report/, June 2023. Accessed: 2023-9-26.
[24] D. H. Ahn, J. Garlick, M. Grondona, D. Lipari, B. Springmeyer, and M. Schulz, "Flux: A Next-Generation Resource Management Framework for Large HPC Centers," in *2014 43rd International Conference on Parallel Processing Workshops*, pp. 9–17, 2014.
[25] C. Misale, D. J. Milroy, C. E. A. Gutierrez, M. Drocco, S. Herbein, D. H. Ahn, Z. Kaiser, and Y. Park, "Towards Standard Kubernetes Scheduling Interfaces for Converged Computing," in *Driving Scientific and Engineering Discoveries Through the Integration of Experiment, Big Data, and Modeling and Simulation* (J. Nichols, A. B. Maccabe, J. Nutaro, S. Pophale, P. Devineni, T. Ahearn, and B. Verastegui, eds.), (Cham), pp. 310–326, Springer International Publishing, 2022.
[26] C. Misale, M. Drocco, D. J. Milroy, C. E. A. Gutierrez, S. Herbein, D. H. Ahn, and Y. Park, "It's a Scheduling Affair: GROMACS in the Cloud with the KubeFlux Scheduler," in *2021 3rd International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*, pp. 10–16, 2021.
[27] D. J. Milroy, C. Misale, G. Georgakoudis, T. Elengikal, A. Sarkar, M. Drocco, T. Patki, J.-S. Yeom, C. E. A. Gutierrez, D. H. Ahn, and Y. Park, "One Step Closer to Converged Computing: Achieving Scalability with Cloud-Native HPC," in *2022 IEEE/ACM 4th International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*, pp. 57–70, 2022.
[28] D. H. Ahn, J. Garlick, M. Grondona, D. Lipari, B. Springmeyer, and M. Schulz, "Flux: A Next-Generation Resource Management Framework for Large HPC Centers," in *2014 43rd International Conference on Parallel Processing Workshops*, pp. 9–17, 2014.
[29] The Kubernetes Authors, "Pods in kubernetes." https://kubernetes.io/docs/concepts/workloads/pods/. Accessed: April 22, 2024.
[30] "Kubernetes: Horizontal pod autoscaling." https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, 2023. Online; Accessed: 08/29/2023.
[31] "Kubernetes: Cluster Autoscaler." https://github.com/kubernetes/autoscaler, 2023. Online; accessed 29-August-2023.
[32] V. Sochat, A. Culquicondor, A. Ojea, and D. Milroy, "The Flux Operator [version 1; peer review: 2 approved] ," *F1000Research*, vol. 13, no. 203, 2024.
[33] https://kubernetes.io/docs/tasks/extend-kubernetes/custom-resources/custom-resource-definitions/#scale-subresource. Accessed: 05/09/2024.
[34] "Kube Flow-MPI Operator." https://github.com/kubeflow/mpi-operator. Online: accessed 19-September-2023.
[35] "Google Cloud-The new way to cloud starts here." https://cloud.google.com/. Accessed: 07/25/2023.
[36] "Microsoft Azure-Endless possibilities—from the edge to the cloud ." https://azure.microsoft.com/en-us. Accessed: 07/25/2023.
[37] "IBM Cloud. Hybrid. Open. Resilient.." https://www.ibm.com/cloud. Accessed: 07/25/2023.
[38] None, "Lower the Time-to-Results for Tightly Coupled HPC Applications on the AWS Cloud with the Elastic Fabric Adapter," tech. rep., AWS, Intel, 2020.
[39] "Elastic Fabric Adapter in AWS." https://aws.amazon.com/hpc/efa/. Last Accessed: 09/17/2023.
[40] L. Shalev, H. Ayoub, N. Bshara, and E. Sabbag, "A Cloud-Optimized Transport Protocol for Elastic and Scalable HPC," *IEEE Micro*, vol. 40, no. 6, pp. 67–73, 2020.
[41] "Scale HPC Workloads with Elastic Fabric Adapter and AWS ParallelCluster." https://aws.amazon.com/blogs/opensource/scale-hpc-workloads-elastic-fabric-adapter-and-aws-parallelcluster/. Accessed: 05/07/2024.
[42] "Placement Groups in AWS." https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/placement-groups.html. Last Accessed: 09/17/2023.
[43] "AWS Global Infrastructure and Availability Zones." https://aws.amazon.com/about-aws/global-infrastructure/. Accessed: 10/03/2023.
[44] D. Perez, L.-H. Hung, S. Xu, K. Y. Yeung, and W. Lloyd, "An Investigation on Public Cloud Performance Variation for an RNA Sequencing Workflow," in *Proceedings of the 11th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*, BCB '20, (New York, NY, USA), Association for Computing Machinery, 2020.
[45] Sochat and Hossen, "converged-computing/kubescaler: v0.0.15.1," Aug. 2023.
[46] "CORAL-2 Benchmarks, Advanced Simulation and Computing." https://asc.llnl.gov/coral-2-benchmarks. Last Accessed: 09/18/2023.
[47] A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. in 't Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen, R. Shan, M. J. Stevens, J. Tranchida, C. Trott, and S. J. Plimpton, "LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales," *Comp. Phys. Comm.*, vol. 271, p. 108171, 2022.
[48] V. E. Henson and U. M. Yang, "BoomerAMG: A parallel algebraic multigrid solver and preconditioner," *Applied Numerical Mathematics*, vol. 41, no. 1, pp. 155–177, 2002. Developments and Trends in Iterative Methods for Large Systems of Equations - in memorium Rudiger Weiss.
[49] "Amg - algebraic multigrid solver codebase." https://github.com/LLNL/AMG2023. Accesses: 05/02/2024.
[50] A. J. Kunen, T. S. Bailey, and P. N. Brown, "KRIPKE - A MASSIVELY PARALLEL TRANSPORT MINI-APP," *OSTI*, 6 2015.
[51] V. A. Dobrev, T. V. Kolev, and R. N. Rieben, "High-Order Curvilinear Finite Element Methods for Lagrangian Hydrodynamics," *SIAM Journal on Scientific Computing*, vol. 34, no. 5, pp. B606–B641, 2012.
[52] "Laghos." https://github.com/CEED/Laghos. Accessed: 07/17/2024.
[53] "eksctl-The official CLI for Amazon EKS." https://eksctl.io/. Accessed: 10/02/2023.
[54] V. Sochat, "converged-computing/flux-metrics-api: Flux Metrics API v0.0.11," July 2024.
[55] "Boto3: AWS SDK for Python." https://boto3.amazonaws.com/v1/documentation/api/latest/index.html. Last Accessed: 9/18/2023.
[56] "AWS CloudFormation." https://aws.amazon.com/cloudformation/. Last Accessed: 07/30/2024.
[57] M. R. Hossen, V. Sochat, A. Sarkar, M. Islam, and D. Milroy, "Repository for workload-driven autoscaling Cluster 2024," Aug. 2024.
[58] A. Raveendran, T. Bicer, and G. Agrawal, "A Framework for Elastic Execution of Existing MPI Programs," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pp. 940–947, 2011.
[59] I. Comprés, A. Mo-Hellenbrand, M. Gerndt, and H.-J. Bungartz, "Infrastructure and API Extensions for Elastic Execution of MPI Applications," in *Proceedings of the 23rd European MPI Users' Group Meeting*, EuroMPI 2016, (New York, NY, USA), p. 82–97, Association for Computing Machinery, 2016.
[60] I. Laguna, T. Gamblin, K. Mohror, M. Schulz, H. Pritchard, and N. Davis, "A Global Exception Fault Tolerance Model for MPI," 9 2014.
[61] G. Georgakoudis, L. Guo, and I. Laguna, "Reinit++: Evaluating the performance of global-restart recovery methods for mpi fault tolerance," in *High Performance Computing* (P. Sadayappan, B. L. Chamberlain, G. Juckeland, and H. Ltaief, eds.), (Cham), pp. 536–554, Springer International Publishing, 2020.
[62] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra, "Post-failure recovery of MPI communication capability: Design and rationale," *The International Journal of High Performance Computing Applications*, vol. 27, no. 3, pp. 244–254, 2013.
[63] P. Liu, G. Bravo-Rocca, J. Guitart, A. Dholakia, D. Ellison, and M. Hodak, "Scanflow-K8s: Agent-based Framework for Autonomic Management and Supervision of ML Workflows in Kubernetes Clusters," in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pp. 376–385, 2022.
[64] D. A. Monge and C. García Garino, "Adaptive Spot-Instances Aware Autoscaling for Scientific Workflows on the Cloud," in *High Performance Computing* (G. Hernández, C. J. Barrios Hernández, G. Díaz, C. García Garino, S. Nesmachnow, T. Pérez-Acle, M. Storti, and M. Vázquez, eds.), (Berlin, Heidelberg), pp. 13–27, Springer Berlin Heidelberg, 2014.
[65] "Kubeflow: The Machine Learning Toolkit for Kubernetes." https://www.kubeflow.org/. Accessed: 10/02/2023.
[66] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-Scaling Web Applications in Clouds: A Taxonomy and Survey," in *ACM Comput. Surv.*, vol. 51, (New York, NY, USA), Association for Computing Machinery, July 2018.
[67] A. Kwan, J. Wong, H.-A. Jacobsen, and V. Muthusamy, "Hyscale: Hybrid and network scaling of dockerized microservices in cloud data centres," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pp. 80–90, 2019.
[68] A. F. Baarzi and G. Kesidis, "Showar: Right-sizing and efficient scheduling of microservices," in *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, (New York, NY, USA), p. 427–441, Association for Computing Machinery, 2021.

[69] F. Rossi, M. Nardelli, and V. Cardellini, "Horizontal and Vertical Scaling of Container-Based Applications Using Reinforcement Learning," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pp. 329–338, 2019.

[70] M. Yan, X. Liang, Z. Lu, J. Wu, and W. Zhang, "Hansel: Adaptive horizontal scaling of microservices using bi-lstm," *Applied Soft Computing*, vol. 105, p. 107216, 2021.

[71] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "FIRM: An Intelligent Fine-Grained Resource Management Framework for SLO-Oriented Microservices," in *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, (USA), USENIX Association, 2020.

[72] Y. Zhang, W. Hua, Z. Zhou, G. E. Suh, and C. Delimitrou, "Sinan: ML-Based and QoS-Aware Resource Management for Cloud Microservices," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, (New York, NY, USA), p. 167–181, Association for Computing Machinery, 2021.

[73] X. Hou, C. Li, J. Liu, L. Zhang, S. Ren, J. Leng, Q. Chen, and M. Guo, "AlphaR: Learning-Powered Resource Management for Irregular, Dynamic Microservice Graph," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 797–806, 2021.

[74] M. R. Hossen, K. Ahmed, and M. A. Islam, "Market mechanism-based user-in-the-loop scalable power oversubscription for hpc systems," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 485–498, 2023.

[75] M. R. Hossen, M. A. Islam, and K. Ahmed, "Practical Efficient Microservice Autoscaling with QoS Assurance," in *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '22, (New York, NY, USA), p. 240–252, Association for Computing Machinery, 2022.

[76] M. R. Hossen, "Pema+: A comprehensive resource manager for microservices," *SIGMETRICS Perform. Eval. Rev.*, vol. 51, p. 10–12, jan 2024.

[77] M. Wajahat, A. Karve, A. Kochut, and A. Gandhi, "MLscale: A machine learning based application-agnostic autoscaler," *Sustainable Computing: Informatics and Systems*, vol. 22, pp. 287–299, 2019.

[78] Z. Yang, P. Nguyen, H. Jin, and K. Nahrstedt, "Miras: Model-based reinforcement learning for microservice resource allocation over scientific workflows," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pp. 122–132, 2019.

[79] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Cost-effective scaling for microservice applications in the cloud with an online learning approach," *IEEE Transactions on Cloud Computing*, vol. 10, no. 2, pp. 1100–1116, 2022.

[80] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch, "Autoscale: Dynamic, robust capacity management for multi-tier data centers," *ACM Transactions on Computer Systems*, vol. 30, 2012.

[81] Q. Li, B. Li, P. Mercati, R. Illikkal, C. Tai, M. Kishinevsky, and C. Kozyrakis, "Rambo: Resource allocation for microservices using bayesian optimization," *IEEE Computer Architecture Letters*, vol. 20, no. 1, pp. 46–49, 2021.

[82] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou, "Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, (New York, NY, USA), p. 19–33, Association for Computing Machinery, 2019.

[83] AWS, Open Source Community, "Karpenter: Just-in-time Nodes for Any Kubernetes Cluster." https://karpenter.sh/, 2023. Version-v0.29.2, Accessed-07/25/2023.