

An elastic job scheduler for HPC applications on the cloud

Aditya Bhosale

University of Illinois Urbana-Champaign
Urbana, IL, USA
adityab@illinois.edu

Laxmikant Kale

University of Illinois Urbana-Champaign
Urbana, IL, USA
kale@illinois.edu

Kavitha Chandrasekar

University of Illinois Urbana-Champaign
Urbana, IL, USA
kchndrs2@illinois.edu

Sara Kokkila-Schumacher

IBM Research
Yorktown Heights, NY, USA
saraks@ibm.com

Abstract

The last few years have seen an increase in adoption of the cloud for running HPC applications. The pay-as-you-go cost model of these cloud resources has necessitated the development of specialized programming models and schedulers for HPC jobs for efficient utilization of cloud resources. A key aspect of efficient utilization is the ability to rescale applications on the fly to maximize the utilization of cloud resources. Most commonly used parallel programming models like MPI have traditionally not supported autoscaling either in a cloud environment or on supercomputers. While more recent work has been done to implement this functionality in MPI, it is still nascent and requires additional programmer effort. Charm++ is a parallel programming model that natively supports dynamic rescaling through its migratable objects paradigm. In this paper, we present a Kubernetes operator to run Charm++ applications on a Kubernetes cluster. We then present a priority-based elastic job scheduler that can dynamically rescale jobs based on the state of a Kubernetes cluster to maximize cluster utilization while minimizing response time for high-priority jobs. We show that our elastic scheduler, with the ability to rescale HPC jobs with minimal overhead, demonstrates significant performance improvements over traditional static schedulers.

CCS Concepts

- Computing methodologies → Parallel programming languages;
- Software and its engineering → Cloud computing.

Keywords

HPC, job scheduler, kubernetes, elasticity

ACM Reference Format:

Aditya Bhosale, Kavitha Chandrasekar, Laxmikant Kale, and Sara Kokkila-Schumacher. 2025. An elastic job scheduler for HPC applications on the cloud. In *Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops '25), November 16–21, 2025, St Louis, MO, USA*. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3731599.3767358>



This work is licensed under a Creative Commons Attribution 4.0 International License.
SC Workshops '25, St Louis, MO, USA
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1871-7/2025/11
<https://doi.org/10.1145/3731599.3767358>

1 Introduction

Cloud platforms have traditionally been used for embarrassingly parallel workloads and loosely coupled web services, microservices, etc. However, due to the cost-effective nature of cloud computing, there has been a steady increase in adoption by the HPC community. In recent years, rapid advancements in AI have only accelerated the adoption of cloud for HPC workloads.

The pay-as-you-go cost model of cloud resources has underscored the importance of efficient utilization of these resources. An important aspect for maximizing resource utilization is the ability to rescale applications on the fly. Several popular distributed ML frameworks have evolved to support efficient execution in cloud environments, featuring well-developed modules for elastic training and fault tolerance [24, 25].

HPC faces additional challenges in running effectively on the cloud. First, HPC applications are typically strongly coupled and highly latency-sensitive. They need high-performance interconnects such as InfiniBand^{TM/SM} to show good strong scaling performance, which traditionally have not been available on cloud platforms. Recent developments, such as the Elastic Fabric Adapter (EFA) [28] in AWS and the increased availability of dedicated HPC instances with InfiniBand interconnects [4], have significantly improved network performance.

Second, HPC applications are not inherently fault-tolerant and cannot continue execution if one of the nodes is killed during execution. Thus, changing the resources assigned to a job at runtime requires careful handling by the application. As a result, most HPC frameworks do not support elastic execution.

Popular parallel programming frameworks such as MPI have not traditionally supported elastic execution. MPI applications requiring rescaling have in the past relied on checkpointing and restarting the application with a different number of ranks. This requires massive programming effort and application-specific knowledge to carefully handle checkpointing and restarting while maintaining correctness and balancing the load among ranks. Moreover, checkpointing to disk is an expensive operation, especially in a cloud environment.

MPI introduced support for dynamic rescaling of processes using the MPI_Comm_Spawn functions in MPI 2.x [11]. However, not all MPI implementations have this functionality, and it requires a significant programming effort to support rescaling and efficient load balancing after rescaling. These functions are also primarily meant for increasing the number of ranks in an application and cannot directly handle scaling down the number of ranks. MPI

Sessions [14], introduced more recently in MPI 4.x, provides a new, more efficient framework for dynamic resource management in MPI jobs, which also handles scaling down the number of ranks. However, it does not address the huge programming effort required for implementing rescaling support in the application.

Charm++ [6, 18] is an object-oriented parallel programming framework that natively supports rescaling and dynamic load balancing without additional programming effort [12], making it an ideal choice for elastic execution.

As the use of cloud resources for running HPC applications increases, it becomes critical to have an efficient scheduler framework for managing these HPC jobs. This presents new challenges for job schedulers, such as batch scheduling, elastic scheduling, etc. There have been several projects in recent years aimed at these challenges. Some of these works are discussed in more detail in a later section of this paper. These projects have been able to make significant advances toward achieving HPC-Cloud convergence by developing scalable batch schedulers [3, 5, 22], demonstrating elastic scheduling for distributed ML training jobs [13], and demonstrating elastic execution of MPI jobs on a Kubernetes® cluster [20]. However, the key challenge of developing an automated HPC job scheduler that can dynamically scale jobs up or down based on resource availability and job traffic conditions in a cloud environment remains unaddressed.

In this paper, we present an operator for running Charm++ applications on a Kubernetes cluster. We also present a priority-based elastic job scheduling policy that is integrated into the operator, which maximizes cluster utilization while minimizing response times and completion times for high-priority jobs by rescaling running jobs on the fly. We then study the scaling performance of Charm++ applications on a Kubernetes cluster. We also evaluate the effectiveness of our elastic scheduler by comparing it to other static scheduling strategies in varying job traffic conditions.

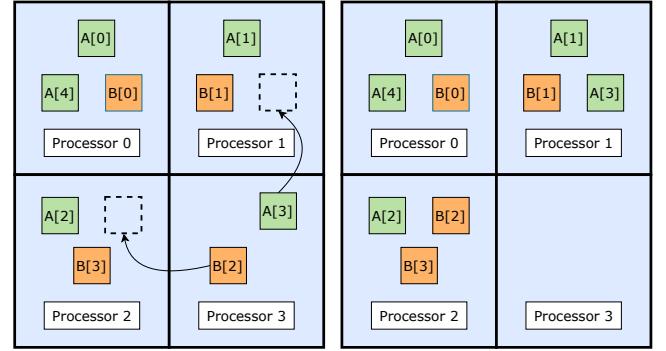
2 Background

2.1 Charm++

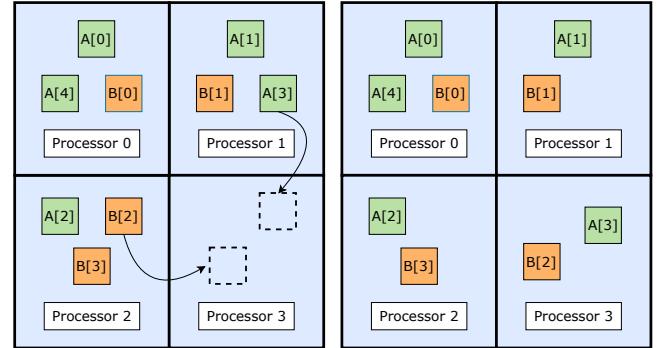
Charm++ [6, 18] is an asynchronous message-driven parallel programming model. Users express computation in terms of objects (*chares*) that communicate via remote method invocations. The mapping of these objects to physical processors is managed by the Charm++ runtime system. Each Processing Element (PE) runs a scheduler and has a message queue. The scheduler picks messages from the message queue and delivers them to the destination object.

When a remote method is called on an object, the destination PE of the object is looked up in a distributed location manager. The arguments to the remote method invocation are then serialized, packed into a message, and sent to the destination PE, where the message is enqueued in the message queue. The scheduler retrieves the message from the queue, deserializes it, and calls the method on the destination object with the deserialized arguments.

These chares are migratable and can be dynamically moved between processors by the runtime system. This allows the runtime system to support dynamic load balancing without additional programming effort by moving chares from overloaded processors to underloaded ones. To enable effective load balancing, Charm++ programs are often overdecomposed, i.e., they have more chares than



(a) When shrinking from 4 to 3 PEs, the runtime moves objects away from processor 3



(b) When expanding from 3 to 4 PEs, the load balancer moves objects from other PEs into the new processor to balance the load

Figure 1: Shrink/Expand in Charm++

processors. Overdecomposition also results in additional performance improvements through computation-communication overlap, making Charm++ applications latency tolerant. This makes Charm++ a good fit for running on cloud platforms with high-latency interconnects [7].

Charm++ is well-suited for dynamic applications with load imbalance and applications with frequent messaging between processes. Charm++ has been adopted in several large-scale applications such as molecular dynamics [15, 26], computational cosmology [16], state space search [19], discrete event simulation [21], and others.

2.2 Shrink/Expand in Charm++

Apart from dynamic load balancing, the concept of migrability discussed in the previous section also enables dynamic rescaling of resources at runtime [12]. To shrink the number of processes in the application, chares can be moved away from certain processors and redistributed among the remaining processors. Similarly, to expand the number of processes, objects from the existing processors can be redistributed to balance the load among the new processors. Figure 1 illustrates the shrink/expand functionality in Charm++. Rescaling is initiated by sending a signal to the Charm++ application from an external program using the Converse Client-Server (CCS) interface [10]. The application then triggers rescaling during the next load-balancing step after receiving the signal.

On receiving a signal to *shrink*, the load balancer in Charm++ disables the assignment of objects to the PEs to be removed. Thus, the load balancer moves objects out of the processes to be killed. To update the runtime data structures in Charm++ after rescaling, the application's state is checkpointed and the application is restarted with the new resources. The checkpointing is performed in Linux® shared memory to avoid the high latency of reading from and writing to disk. Similarly, when a signal to *expand* is received, the application is first restarted with the new processes. A load balancing step is performed after the restart to distribute the load evenly among the new processes.

Charm++ previously supported the rescaling functionality only in the netlrts build, which is a general-purpose, portable communication layer built on standard TCP/UDP sockets. As part of this work, we extended the rescaling functionality to the MPI build of Charm++, which resulted in a significant reduction in rescaling overheads.

2.3 Kubernetes

Kubernetes is an open-source software for cluster management and deployment of containerized applications. While initially designed for microservices and web applications, there has been a growing interest in using Kubernetes for HPC applications. Containerized execution on Kubernetes provides several advantages for HPC applications, such as reproducibility, consistent environments across different platforms, and ease of distribution. Kubernetes pod scheduling can be used for controlling CPU, memory, and GPU resources allocated to an HPC job. Other features, such as node affinity and anti-affinity, can be used for controlling pod placement to optimize communication performance.

While the default kube-scheduler doesn't support batch scheduling, more recent work has resulted in a rich ecosystem of efficient plugin schedulers developed specifically for HPC workloads [1, 5, 22, 23].

The operator framework in Kubernetes can be used to extend the native Kubernetes functionality by implementing domain-specific application management features. A Kubernetes operator consists of 2 main components.

- Custom Resource Definition (CRD) - which defines custom resource types for a specific application.
- Controller - A control loop that manages the custom resources and takes actions to maintain a desired state

Kubeflow's® MPI Operator [2] uses this operator pattern to manage MPI applications on Kubernetes. The MPI Operator launches a launcher pod and several worker replicas based on the MPI Job specification YAML file. Each worker replica can run several MPI ranks. Henceforth in this paper, job is used to indicate either an MPI Job or a Charm++ Job, and the use of replicas refers to the number of worker replicas in an MPI/Charm++ Job. The controller creates a hostfile with the domain names for worker pods used by MPI to connect to those workers. The worker pods run an SSH® server and the launcher pod runs the `mpirun` command.

3 Implementation

3.1 Charm++ operator

To run Charm++ applications on Kubernetes clusters, we extended the MPI operator. We added functionality to rescale jobs when the deployment YAML file is modified. We use the default kube-scheduler for pod placement. We added pod affinity to the operator to ensure locality-aware placement of pods in the cluster. Similar to the hostfile, the controller creates a nodelist file that Charm++ uses to connect to the worker replicas.

We used the non-SMP build of Charm++ for our implementation, ie. each process is a single PE. We use a single process per worker replica to allow a more fine-grained control over elasticity.

Since Charm++ uses Linux shared memory for checkpointing, we do not need a persistent volume mount in our jobs. However, a common default shared memory size for each pod is 64MB. We used an `emptyDir` memory-backed volume mounted to `/dev/shm` to work around this restriction.

To shrink a running job, we follow these steps,

- Send shrink signal to Charm++ application
- After the Charm++ application returns an acknowledgment for the shrink operation, remove extra pods from the job

Similarly, to expand a job,

- Add new pods to a job
- Update the nodelist file to include new pods
- Send expand signal to the Charm++ application

3.2 Elastic scheduling

Previous work on a proof-of-concept adaptive scheduler for elastic Charm++ jobs on an HPC system has been shown to have significant performance improvements over scheduling rigid jobs for a small number of jobs [12]. This scheduler, however, did not consider user-defined priorities for jobs and used a First Come First Serve policy for job allocation, which can potentially result in under-utilization of the cluster if a job with large resource requirements blocks smaller jobs that were submitted later from filling the gaps in the cluster. Our approach incorporates the following improvements beyond the previous work [12]: (a) considering user-defined priorities for making scheduling decisions, (b) out-of-order allocations if they improve cluster utilization, and (c) a Kubernetes operator that can handle a much larger number of jobs.

Several scheduling policies can utilize the rescaling functionality of Charm++ to maximize the cluster performance using different performance metrics. In this paper, we present and evaluate one such policy, which attempts to maximize cluster utilization while minimizing wait times for high-priority jobs. At the end of this section, we will discuss other factors that we haven't considered in our implementation, which may be of interest when defining an elastic scheduling policy.

3.2.1 Scheduling policy. We modified the MPI operator CRD to include `minReplicas` and `maxReplicas` fields for the workers specification. The scheduler can launch and rescale a job to any number of PEs between its minimum and maximum replicas configuration. We also added a `priority` field to the job specification. Two jobs with the same user-defined priority are prioritized based on the job submission time. The job that was submitted earlier has a higher

```

def newJob(job):
    replicas = min(freeSlots - 1, job.maxReplicas)
    if replicas >= job.minReplicas:
        createOrExpandJob(job, replicas)
    else:
        numToFree = job.minReplicas - freeSlots + 1
        index = len(runningJobs) - 1
        while numToFree > 0 and index > 0:
            # runningJobs is a list of running jobs sorted
            # in decreasing order of priority
            j = runningJobs[index--]
            if currentTime() - j.lastAction <
                rescaleGap:
                continue
            if j.priority > job.priority:
                break
            if j.replicas > j.minReplicas:
                newReplicas = max(j.minReplicas,
                                   j.replicas - numToFree)
                numToFree -= (j.replicas -
                               newReplicas)

        if numToFree > 0:
            enqueueJob(job)
        return

    minToFree = job.minReplicas - freeSlots + 1
    maxToFree = job.maxReplicas - freeSlots + 1
    index = len(runningJobs) - 1
    while maxToFree > 0 and index > 0:
        j = runningJobs[index--]
        if currentTime() - j.lastAction <
            rescaleGap:
            continue
        if j.priority > job.priority:
            break
        if j.replicas > j.minReplicas:
            newReplicas = max(j.minReplicas,
                               j.replicas - maxToFree)
            oldReplicas = j.replicas
            if (shrinkJob(j, newReplicas)):
                numFreed = (oldReplicas -
                            newReplicas)
                minToFree -= numFreed
                maxToFree -= numFreed

    if minToFree > 0:
        enqueueJob(job)
    return

```

Figure 2: Pseudocode for priority-based scheduling algorithm when a new job is submitted

priority. We do not modify the memory limits of the pods when rescaling a job. The memory limit in the worker specification is assumed to be the limit when running on the minimum replicas configuration. To control the frequency of rescaling events, the scheduler maintains a configurable $T_{rescale_gap}$ minimum gap between any two scheduling events (creation, shrink, expand) [12].

Figure 2 shows the pseudocode for the elastic scheduling policy when a new job is submitted. The scheduler scales down jobs with a lower priority as long as they meet the $T_{rescale_gap}$ criteria until enough CPUs are freed to run the new job at its maximum replicas configuration. If the CPUs that would be freed by scaling down existing jobs are not enough to start the new job at its minimum replicas configuration, the new job is enqueued in an internal priority queue.

Figure 3 shows the pseudocode for the elastic scheduling policy when a job finishes execution. The freed CPUs are reassigned to

```

def completeJob(job):
    numWorkers = freeWorkers(job)
    # freeWorkers deletes the pods associated with the job
    # and returns the number of slots that were freed
    index = 0
    while numWorkers > 0 and index < len(allJobs):
        # allJobs is a list of all running and queued jobs
        # sorted in decreasing order of priority
        j = allJobs[index++]
        if currentTime() - j.lastAction <
            rescaleGap:
            continue
        if j.replicas < j.maxReplicas:
            addReplicas = min(numWorkers,
                              j.maxReplicas - j.replicas)
            if j.replicas + addReplicas >=
                j.minReplicas:
                if (createOrExpandJob(j,
                                      j.replicas + addReplicas)):
                    numWorkers -= addReplicas
            freeSlots += numWorkers

```

Figure 3: Pseudocode for priority-based scheduling algorithm when a job completes

either existing running jobs or used to start new jobs from the internal job queue based on the priorities of these jobs.

Note that this algorithm tries to maximize cluster utilization by scaling jobs while using a minimal number of rescaling calls to avoid the rescaling overheads. Thus, it may not always ensure that the highest priority jobs are running with their maximum replicas. For instance, a lower priority job may be running with maximum replicas on the cluster when a higher priority job arrives. If the free slots in the cluster are not enough to run the higher priority job at its maximum replicas configuration, but are enough to run with its minimum replicas configuration, our scheduling algorithm will run the higher priority job at its minimum replicas configuration to avoid a shrink call to the lower priority job. However, if enough slots are not available to start the higher priority job even at its minimum replicas configuration, the lower priority job will be scaled down as long as the $T_{rescale_gap}$ criteria is met to run the higher priority job.

3.2.2 Discussion. The following are some factors that we do not consider in the scheduling policy presented in this section, but that could be of interest for an elastic scheduler.

Fault tolerance. Node failures are not an uncommon occurrence in cloud environments. In the scheduling policy we presented, we do not consider fault tolerance, as our goal was to demonstrate rescaling without requiring a shared filesystem. However, Charm++ natively supports fault tolerance by enabling checkpointing of charé data to disk every few iterations, and restarting from a checkpoint by adding an extra command-line parameter to the application launch command. The operator can be modified to launch with the extra restart parameter when a job restarts after a failure, which would start the application from the checkpoint if checkpoint data is found.

Job preemption. The scheduling policy we presented does not preempt lower-priority jobs to run higher-priority jobs. This decision was again made to avoid the requirement of a shared filesystem. However, in the presence of a shared filesystem, lower-priority jobs could be sent a signal to checkpoint to disk and then be preempted

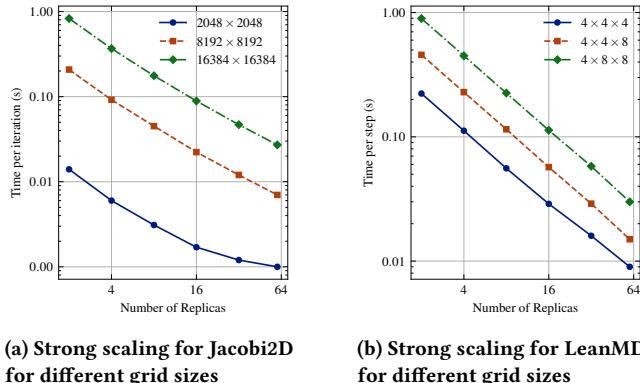


Figure 4: Scaling performance of Charm++ on Kubernetes

to make room for higher-priority jobs. The lower-priority job can then be restarted from its checkpoint at a later time by simply using the restart command-line parameter.

Aging priorities. A potential issue with the scheduling policy we presented is the starvation of low-priority jobs. A dynamic priority system could be implemented to gradually increase the priority of waiting jobs, ensuring that low-priority jobs get resources during times of high traffic.

4 Evaluation

For evaluation, we ran experiments on a 4-node Amazon EKS^{TM/SM} cluster with c6g.4xlarge instance type having 16 vCPUs each with a total of 64 vCPUs. We created all nodes in a single subnet in a cluster placement group for better networking performance. We used the non-SMP `mpi-linux-arm8` build of Charm++ with shrink/expansion enabled, built with OpenMPI v3.1.

4.1 Charm++ performance on Kubernetes

First, we study the scaling performance of Charm++ applications on our EKS cluster using 2 applications.

- Jacobi2D - This application solves the steady-state heat equation on a 2D grid using Jacobi iteration. This is a communication-intensive application.
- LeanMD - This is a molecular dynamics application that simulates atoms considering only the Lennard-Jones potential. This is a computationally intensive application.

Strong scaling results for Jacobi2D are seen in figure 4a. The different problem sizes are the dimensions of the 2D grid. We observe that for larger datasets, Jacobi2D scales well with an increasing number of replicas.

Figure 4b shows LeanMD's strong scaling performance. The problem sizes here are the number of cells in the application. Each cell has a fixed initial number of atoms. The simulation computes forces between atoms in the cells iteratively. Given that it's a compute-intensive application, we observe that LeanMD scales well with increasing replicas as expected.

4.2 Rescaling overhead

The rescaling overhead for Charm++ applications can be split into 4 parts [12].

- Checkpoint - Time taken to checkpoint state to Linux shared memory
- Restart - Time taken to restart the application
- Restore - Time taken to restore the checkpoint data from shared memory
- Load balance - Time taken to load balance. In the case of shrink, before the checkpoint/restart, and in the case of expand, after.

In this experiment, we use the 2D Jacobi solver from the previous section to study the contribution of each factor to the total rescaling overhead.

To study the effect of the number of replicas on the overhead, we use a constant grid size of $8k \times 8k$. We conduct 2 experiments. First, we shrink or scale down a job to half the number of replicas with a different number of original replicas and measure the time taken in each stage of rescaling. Second, we expand or scale up a job to double the number of replicas and take similar runtime measurements.

Figure 5a and 5b show the overhead of shrink and expand, respectively. We see that the application restart time increases with an increase in the number of replicas since the MPI startup time increases with the number of ranks. The checkpoint and restore time decreases with an increase in replicas because the size of the checkpoint data per replica reduces as we increase the number of replicas, since the total problem size is constant. The load balancing time stays flat with an increase in the number of replicas.

To study the effect of problem size on the overhead, we shrink jobs of different sizes from 32 to 16 replicas and measure the overhead of each stage. Figure 5c shows the result of this experiment. We see that for small problem sizes, the overhead is dominated by the restart time. As the problem size increases, the load balancing, checkpoint, and restore time scale up while the restart time remains flat. We see that the overhead of in-memory checkpointing and restoring remains significantly low even for a problem with data size of 4GB.

Figure 6a shows the time taken per 10 iterations for a $16k \times 16k$ grid. When the application is scaled down from 32 replicas to 16 replicas, we see that the time for each iteration increases. Similarly, when the application is scaled back up to 32 replicas, the time per iteration drops back down to its original value.

Figure 6b shows the timeline plot for the time at which each iteration finishes execution. The slope of the line indicates the speed at which iterations are executed. We see that the slope of the line increases after scaling down, indicating a slower speed of executing iterations, and decreases after scaling up. The gaps in the timeline plot during shrink and expand show the overhead of rescaling.

4.3 Elastic scheduling performance

We compare our elastic scheduling policy with 3 other scheduling strategies. All these strategies use the same priority logic.

- Rigid with `min_replicas` - All jobs are launched with `min_replicas` number of replicas without rescaling

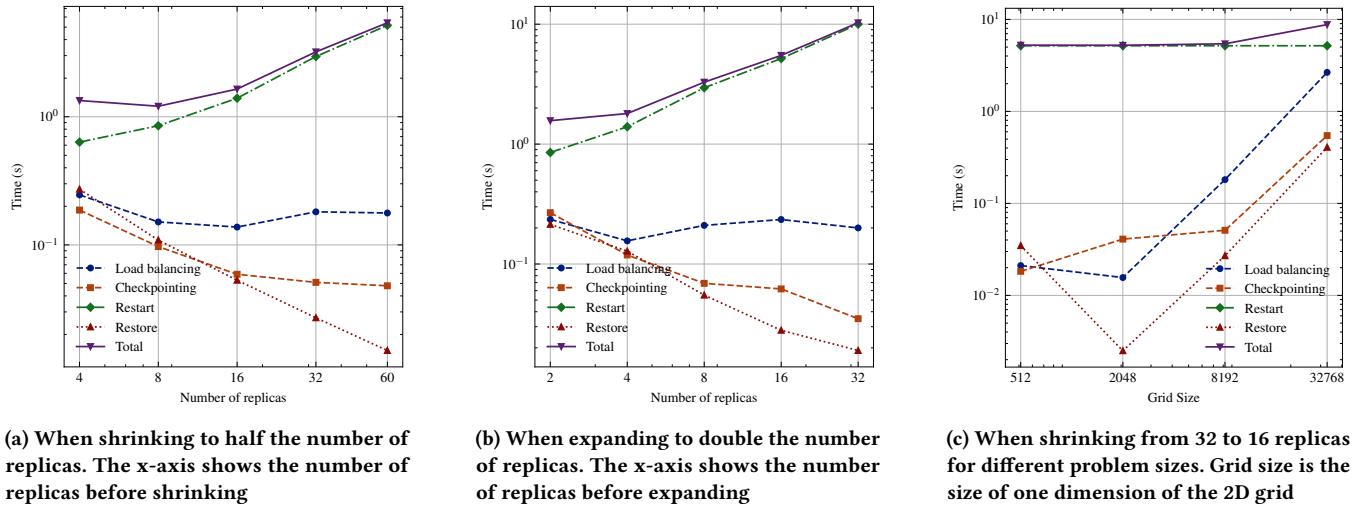
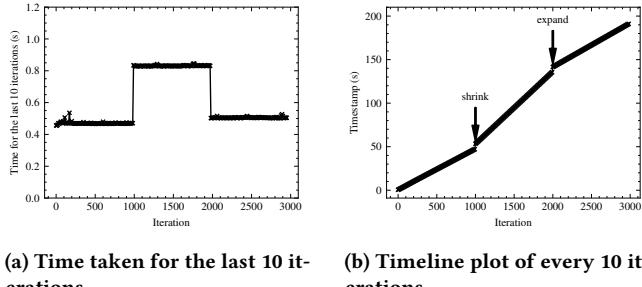


Figure 5: Contribution of different stages of rescaling to the total rescaling overhead

Figure 6: Timeline for Jacobi2D with a *shrink* from 32 replicas to 16 replicas and an *expand* from 16 replicas to 32 replicas

- Rigid with `max_replicas` - All jobs are launched with `max_replicas` number of replicas without rescaling
- Moldable - The scheduler assigns the number of replicas based on the cluster state to maximize utilization, but doesn't rescale any job during execution [9]

We use the following metrics for evaluating the performance of these scheduling policies,

- Total time - The end-to-end runtime from the start of the first job to the end of the last job
- Cluster utilization - Average percentage utilization of the cluster over the duration of the experiment
- Weighted mean response time - Mean response time weighted by job priority. We define response time as the time between a job submission and start
- Weighted mean completion time - Mean completion time weighted by job priority. Completion time is defined as time between a job submission and completion

The performance of the 4 schedulers depends on several factors, such as the size of the jobs, the order in which jobs are submitted,

the rate of job submission, the choice of $T_{rescale_gap}$, etc. An experimental study to measure the effect of each of these factors on the scheduler performance using actual runs would be infeasible. In this section we present the results from a simulator we wrote for modeling the scheduler performance for the Jacobi 2D example. We also present the results from an experimental run on a Kubernetes cluster.

4.3.1 Simulation results. We use the Jacobi2D solver example from the previous section for our simulations. We use 4 different problem sizes - small, medium, large, and `xlarge` based on their grid sizes and number of timesteps.

- Small - 512×512 grid. 40,000 timesteps. `min_replicas` = 2, `max_replicas` = 8
- Medium - 2048×2048 grid. 40,000 timesteps. `min_replicas` = 4, `max_replicas` = 16
- Large - 8192×8192 grid. 40,000 timesteps. `min_replicas` = 8, `max_replicas` = 32
- XLarge - $16,384 \times 16,384$ grid. 10,000 timesteps. `min_replicas` = 16, `max_replicas` = 64

We pick 16 jobs randomly out of these 4 sizes with random priorities between 1 and 5. We repeat this experiment 100 times and report the average metrics across all runs. We do not consider the overhead added by the operator or by Kubernetes to start up the pods. We use strong scaling performance measurements for the 4 problem sizes to model the runtime of a job for a given number of replicas using a piecewise linear function. We also use the rescaling overhead measurements to model the overhead for rescaling using a piecewise linear function.

Effect of Job Submission Rate. We study the effect of job submission rate on the performance metrics defined earlier by varying the gap between two consecutive submissions from 0 to 300s. We used $T_{rescale_gap} = 180$ s for these simulations. Figure 7 shows the results of the simulations.

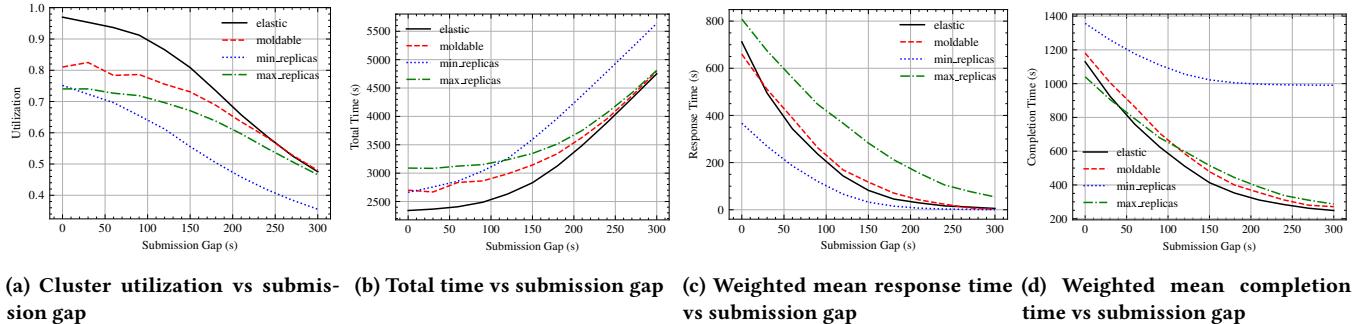


Figure 7: Results for scheduling performance simulation for different job submission rates. The submission gap is the time between two consecutive job submissions

The cluster utilization is highest for the elastic scheduler and lowest for the `min_replicas` scheduler, as expected. The total time is lowest for the elastic scheduler. The `min_replicas` scheduler has a lower total time than the `max_replicas` scheduler when the submission gap is small because the jobs can run with a higher parallel efficiency when using a smaller number of replicas, and a small gap in job submissions results in a higher cluster utilization. As the submission gap is increased, however, the cluster utilization using the `min_replicas` scheduler drops, and the higher parallel efficiency can no longer offset the lower utilization, which results in a larger total time. The total time for the other 3 schedulers converges as the submission gap increases, since with a large enough gap between jobs, each job can run to completion with the maximum number of replicas.

The weighted mean response time is lowest for the `min_replicas` scheduler because the low cluster utilization leaves resources available for any higher priority jobs submitted later. The elastic scheduler has a better response time than the moldable and `max_replicas` schedulers because of its ability to rescale low-priority jobs to run higher-priority jobs.

The weighted mean completion time is the highest for the `min_replicas` scheduler, even though the response time is low, because it runs jobs with a lesser degree of parallelism, thus increasing their runtime. The `max_replicas` scheduler has the lowest weighted mean completion time for very small submission gaps because it can run the jobs in the correct priority order using the maximum degree of parallelism. The elastic and moldable schedulers have a slightly worse completion time for small submission gaps because they will run some jobs with a smaller number of replicas to fill the gaps in the cluster and maximize cluster utilization. As a result, the elastic and moldable schedulers have a higher cluster utilization and a lower total time and mean response time, but the increased runtime of these jobs increases the completion time. As the submission gap increases, the `max_replicas` and moldable schedulers can saturate the cluster with lower priority jobs before a higher priority job is submitted, resulting in worse completion times. The elastic scheduler has a lower mean completion time since it can rescale these running lower-priority jobs.

Our simulations show that the elastic scheduler can significantly reduce the end-to-end time of jobs and maximize cluster utilization

while at the same time reducing the response time and completion time of jobs.

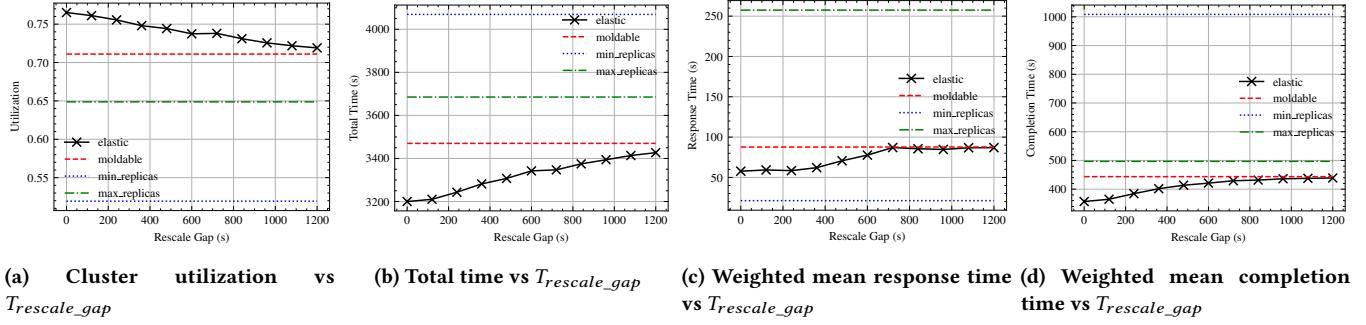
Effect of $T_{rescale_gap}$. We repeat the previous experiment here with a fixed submission gap of 180s and vary $T_{rescale_gap}$ from 0 to 1200s. Figure 8 shows the results of the simulations.

The cluster utilization is highest with a small $T_{rescale_gap}$ because the scheduler can rescale running jobs more frequently to maximize utilization. More frequent rescaling operations add a rescaling overhead to the total time. However, figure 8b shows that the total time increases monotonically as the rescale gap is increased. This is because the rescaling overhead is small enough that the effect of improved cluster utilization outweighs the penalty of a higher rescaling overhead.

The weighted mean response time and weighted completion time behave similarly to the total time. The mean response time for the `min_replicas` scheduler is the lowest because of low cluster utilization. The low cluster utilization ensures that higher-priority jobs start early, but they take longer to complete. This is reflected in the higher mean completion time for the `min_replicas` scheduler. All the metrics for the elastic scheduler approach the moldable scheduler as $T_{rescale_gap}$ is increased, since the moldable scheduler is essentially the elastic scheduler that never rescales any job.

These simulations show that the elastic scheduler performs well for a wide range of choices of $T_{rescale_gap}$. They also show that the overhead of rescaling is low enough that a very small $T_{rescale_gap}$ affects the performance of the elastic scheduler minimally.

4.3.2 Experimental performance. To experimentally measure and compare the performance of the schedulers on a Kubernetes cluster, we pick a configuration out of the randomly generated jobs by the simulator. We use $T_{rescale_gap} = 180$ s and a job submission gap of 90s. Since there is no load imbalance in this example, we only load balance when a job has to be rescaled. The rigid job schedulers are emulated by setting the same value for `min_replicas` and `max_replicas` for all jobs. The moldable scheduler is emulated by setting a large $T_{rescale_gap}$ value to prevent the jobs from rescaling after they are launched [12]. Table 1 shows the comparison of performance metrics obtained from the simulation shown in the previous section, along with the actual performance metrics obtained from the experimental run.

Figure 8: Results for scheduling performance simulation for different $T_{rescale_gap}$

Scheduler	Total time (s)		Cluster utilization		Weighted mean response time (s)		Weighted mean completion time (s)	
	Actual	Simulation	Actual	Simulation	Actual	Simulation	Actual	Simulation
min_replicas	2511	2402	58.12%	60.88%	162.87	207.21	888.59	915.08
max_replicas	2149	1914	81.32%	85.86%	147.51	195.79	275.36	326.68
Moldable	2111	2078	71.54%	78.39%	92.93	122.40	273.89	326.15
Elastic	1796	1813	87.80%	92.26%	69.93	32.96	265.76	241.29

Table 1: Actual and simulation performance results for the 4 scheduling policies

The `min_replicas` scheduler has the lowest cluster utilization since all jobs run with the minimum number of replicas specified by the user. When running with `max_replicas`, the utilization is higher, but because of the rigidity of jobs, some gaps in utilization are not filled. The total time for the `max_replicas` scheduler is lower than the total time for the `min_replicas` scheduler because of the low cluster utilization of the latter. Both the `min_replicas` and `max_replicas` schedulers have a high response time since higher-priority jobs that were submitted after the cluster was fully utilized had to wait for the lower-priority jobs to finish executing.

In the case of moldable scheduling, the cluster utilization is high because the scheduler always picks the number of replicas to maximize cluster utilization. However, because the jobs cannot be rescaled after they are launched, a job that was submitted during high traffic can continue to run at a lower configuration even after the cluster has free slots, as can be seen in our experiment in figure 9a. The response time for the moldable scheduler is lower than the `max_replicas` scheduler even when they have similar utilization because the moldable scheduler is able to start higher-priority jobs at a lower number of replicas configuration earlier, unlike the `max_replicas` scheduler that has to queue them until the required slots are free.

The elastic scheduler is able to maximize the cluster utilization as well as minimize the mean response time and the mean completion time. The increase in cluster utilization compared to other scheduling strategies can be seen in figure 9a. When a higher-priority job is submitted to the cluster, unlike the moldable scheduler, the elastic scheduler can rescale the running lower-priority jobs to

make room for the higher-priority job. Figure 9b shows an `xlarge` job that rescales multiple times with the elastic scheduler.

5 Related work

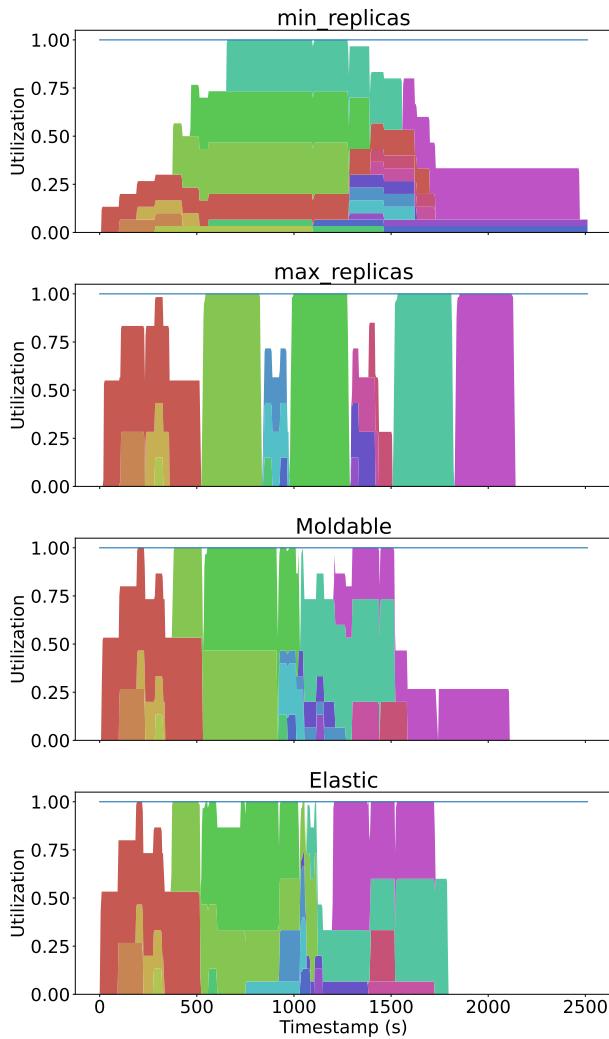
Over the past few years, there have been several projects to enable efficient execution of HPC workloads on the cloud. Previous work on running HPC applications on Kubernetes broadly follows two approaches. The first is to develop an efficient scheduler for controlling pod placement to minimize communication cost for MPI applications. The second is to develop a higher-level operator focusing on domain-specific management of applications.

Volcano™ scheduler [5] is a Kubernetes-native batch scheduling system for HPC workloads. It supports features like network topology-aware scheduling, priority-based scheduling, cross-cluster job scheduling, etc. The MPI operator has support for using Volcano for gang scheduling of pods.

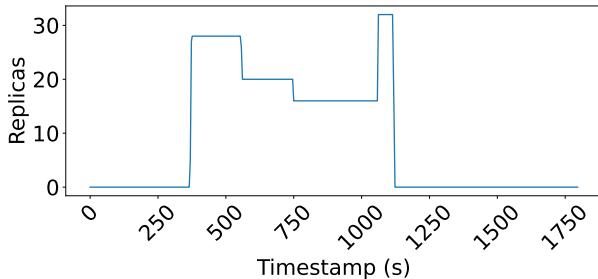
Kubeflux [23] is a plugin scheduler for Kubernetes for running HPC workloads. It demonstrated better scaling performance than the default kube-scheduler using the MPI operator.

More recent work on Kubeflux, now called Fluence, demonstrates the scalability of the scheduler by running production-grade MPI applications on up to 3000 ranks [22]. The authors implemented several optimizations in the MPI operator to improve scalability and reliability. They demonstrated large-scale runs of production-grade MPI applications with good strong scaling performance and low variability.

Kueue [3] is a job queueing and resource management system for managing batch workloads. They support various types of jobs,



(a) Cluster utilization profiles for the 4 schedulers we study in this experiment. Each color represents a different job



(b) Evolution of number of replicas with time for an xlarge job using the elastic scheduler

Figure 9: Results for scheduling performance comparison experiments on Amazon EKS

including Kubeflow's MPIJob. Kueue also supports other features such as partial admission with reduced parallelism based on user quota, topology-aware scheduling, etc.

Voda scheduler [13] extends the MPI operator for elastic distributed training. It supports locality-aware pod placement and addresses the issue of fragmented allocation after rescaling by supporting worker migration.

Kub [20] is a Kubernetes-based framework for elastic execution of MPI applications on the cloud. The authors demonstrate running MPI applications on the cloud with rescaling by checkpointing and restarting the application. However, they do not consider resource availability and use a static scaling protocol while making scaling decisions.

6 Future Work

The elastic scheduler presented in this paper has some limitations that we plan to address, and presents several avenues to explore in the future. When making scheduling decisions, we do not consider the cost versus the potential benefit of rescaling a job. The potential benefit of rescaling depends on several factors,

- *Number of replicas by which a job is scaling up.* A small increase in the number of replicas may not justify the overhead of rescaling.
- *Percentage of job already completed.* If only a small fraction of a job remains, scaling up may not provide enough benefit. Similarly, allowing the job to complete would be more efficient than scaling it down to start another job.
- *Parallel efficiency of the job.* If the job has a very low parallel efficiency at its current number of replicas, scaling up may not improve performance enough to offset the rescaling cost.

Incorporating some of these factors into the scheduler's decision-making requires input from the application during the decision-making process. This could be achieved by giving the application control to accept or decline a rescaling command sent by the scheduler. The application can then decline a rescaling command if only a small fraction of the application run remains. It can also decline a scaling-up command if the parallel efficiency of the job, as measured by runtime instrumentation, is lower than a specified threshold.

Another avenue to explore is running Charm++ applications as evolving jobs. Unlike elastic jobs, where the rescaling signal is sent from an external scheduler, evolving jobs can rescale at runtime based on internal, application-specific criteria without any external trigger [9]. This approach could be particularly useful in applications where the compute load changes dynamically over time, for example, due to dynamic refinement of the underlying grid in a numerical solver.

Charm++ also supports execution on GPUs [8]. However, shrink-/expand operations on GPUs aren't supported yet. In the future, we plan to support rescaling with GPUs and extend our operator to manage GPU resources. This will enable us to use the operator for running elastic distributed ML training workloads, in addition to other large-scale HPC applications that already use the GPU support in Charm++ [17, 27].

7 Conclusion

In this paper, we extended the KubeFlow MPI operator to run Charm++ applications on a Kubernetes cluster. We added functionality for dynamic rescaling of jobs to the operator. We showed scaling performance results for two Charm++ applications. We also measured the overhead of rescaling operations and attributed the cost to different stages of rescaling. We found that the overhead is dominated by the restart time for a problem size of up to 4GB, with a very low overhead of in-memory checkpointing and restoring.

We presented a priority-based elastic job scheduling policy that can dynamically rescale jobs based on the state of the cluster to maximize utilization and minimize response times for high-priority jobs.

We evaluated the performance of the elastic scheduling policy against two rigid scheduling policies and a moldable scheduling policy on 4 different performance metrics. We wrote a scheduling simulator to compare the performance of the 4 scheduling policies across a wide range of job submission patterns and showed that the elastic scheduler performs better than the other 3 scheduling policies. We ran an example job set on an AWS EKS Kubernetes cluster and showed that the elastic scheduler performed better on all 4 performance metrics.

We showed that traditional rigid job scheduling policies can perform well under certain conditions - the `min_replicas` scheduler performs well under high-traffic conditions, while the `max_replicas` scheduler performs well in low-traffic conditions. The dynamic scheduling policies, i.e., elastic and moldable, show good performance in all traffic conditions, with the elastic scheduling policy outperforming the moldable scheduling policy on all metrics.

Acknowledgments

This work is supported by the IBM-Illinois Discovery Accelerator Institute (IIDAI).

References

- [1] [n. d.]. A batch scheduler of kubernetes for high performance workload, e.g. AI/ML, BigData, HPC. <https://github.com/kubernetes-retired/kube-batch>.
- [2] [n. d.]. Kubernetes Operator for MPI-based applications. <https://github.com/kubeflow/mpi-operator>.
- [3] [n. d.]. Kueue - Kubernetes-native Job Queueing. <https://kueue.sig.sigs.k8s.io/>.
- [4] [n. d.]. More performance and choice with new Azure HBv3 virtual machines for HPC. <https://azure.microsoft.com/en-us/blog/more-performance-and-choice-with-new-azure-hbv3-virtual-machines-for-hpc/>.
- [5] [n. d.]. Volcano - Cloud native batch scheduling system for compute-intensive workloads. <https://volcano.sh/en/>.
- [6] Bilge Acun, Abhishek Gupta, Nikhil Jain, Akhil Langer, Harshitha Menon, Eric Mikida, Xiang Ni, Michael Robson, Yanhua Sun, Ehsan Totoni, Lukasz Wesolowski, and Laxmikant Kale. 2014. Parallel Programming with Migratable Objects: Charm++ in Practice (SC).
- [7] Aditya Bhosale, Laxmikant Kale, and Sara Kokkila-Schumacher. 2025. Efficient and Cost-Effective HPC on the Cloud. In *The 34th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '25)*. ACM, New York, NY, USA, 5 pages. doi:10.1145/3731545.3744667
- [8] J. Choi, D. F. Richards, and L. V. Kale. 2020. Achieving Computation-Communication Overlap with Overdecomposition on GPU Systems. In *2020 IEEE/ACM 5th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*, 1–10. doi:10.1109/ESPM251964.2020.00006
- [9] Dror G. Feitelson and Larry Rudolph. 1996. Toward convergence in job schedulers for parallel supercomputers. In *Job Scheduling Strategies for Parallel Processing*. Dror G. Feitelson and Larry Rudolph (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–26.
- [10] Filippo Gioachin, Chee Wai Lee, and Laxmikant V. Kalé. 2009. Scalable Interaction with Parallel Applications. In *Proceedings of TeraGrid '09*. Arlington, VA, USA.
- [11] William Gropp, Ewing Lusk, and Rajeev Thakur. 1999. *Using MPI-2: Advanced Features of the Message-Passing Interface*. The MIT Press. doi:10.7551/mitpress/7055.001.0001
- [12] Abhishek Gupta, Bilge Acun, Osman Sarood, and Laxmikant V. Kale. 2014. Towards Realizing the Potential of Malleable Parallel Jobs. In *Proceedings of the IEEE International Conference on High Performance Computing (HiPC '14)*. Goa, India.
- [13] Tsung-Tso Hsieh and Che-Rung Lee. 2023. Voda: A GPU Scheduling Platform for Elastic Deep Learning in Kubernetes Clusters. In *2023 IEEE International Conference on Cloud Engineering (IC2E)*. 131–140. doi:10.1109/IC2E59103.2023.00023
- [14] Dominik Huber, Maximilian Streubel, Isaias Comprés, Martin Schulz, Martin Schreiber, and Howard Pritchard. 2022. Towards Dynamic Resource Management with MPI Sessions and PMIx. In *Proceedings of the 29th European MPI Users' Group Meeting (Chattanooga, TN, USA) (EuroMPI/USA '22)*. Association for Computing Machinery, New York, NY, USA, 57–67. doi:10.1145/3555819.3555856
- [15] Nikhil Jain, Eric Bohm, Eric Mikida, Subhasish Mandal, Minjung Kim, Prateek Jindal, Qi Li, Sohrab Ismail-Beigi, Glenn Martyna, and Laxmikant Kale. 2016. OpenAtom: Scalable Ab-Initio Molecular Dynamics with Diverse Capabilities. In *International Supercomputing Conference (ISC HPC '16)*.
- [16] Prithish Jetley, Filippo Gioachin, Celso Mendes, Laxmikant V. Kale, and Thomas R. Quinn. 2008. Massively parallel cosmological simulations with ChaNGa. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*.
- [17] Prithish Jetley, Lukasz Wesolowski, Filippo Gioachin, Laxmikant V. Kalé, and Thomas R. Quinn. 2010. Scaling Hierarchical N-body Simulations on GPU Clusters. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*. IEEE Computer Society, Washington, DC, USA.
- [18] LV. Kalé and S. Krishnan. 1993. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In *Proceedings of OOPSLA'93*, A. Paepcke (Ed.). ACM Press, 91–108.
- [19] Akhil Langer, Ram Venkataraman, Uddatta Palekar, and Laxmikant Kale. 2013. Parallel Branch-and-bound for Two-Stage Stochastic Integer Optimization. In *High Performance Computing (HiPC), 2013 20th International Conference on*.
- [20] Daniel Medeiros, Jacob Wahlgren, Gabin Schieffer, and Ivy Peng. 2023. Kub: Enabling Elastic HPC Workloads on Containerized Environments. In *2023 IEEE 35th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 219–229. doi:10.1109/SBAC-PAD59825.2023.00031
- [21] Eric Mikida, Nikhil Jain, Elsa Gonsiorowski, Peter D. Barnes, Jr., David Jefferson, Christopher D. Carothers, and Laxmikant V. Kale. 2016. Towards PDES in a Message-Driven Paradigm: A Preliminary Case Study Using Charm++. In *ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS) (SIGSIM PADS '16 (to appear))*. ACM.
- [22] Daniel J. Milroy, Claudia Misale, Giorgis Georgakoudis, Tonia Elengikal, Abhik Sarkar, Maurizio Drocco, Tapasya Patki, Jae-Seung Yeom, Carlos Eduardo Arango Gutierrez, Dong H. Ahn, and Yoonho Park. 2022. 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)*. 57–70. doi:10.1109/CANOPIE-HPC56864.2022.00011
- [23] Claudia Misale, Maurizio Drocco, Daniel J. Milroy, Carlos Eduardo Arango Gutierrez, Stephen Herbein, Dong H. Ahn, and Yoonho Park. 2021. 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)*. 10–16. doi:10.1109/CANOPIE-HPC54579.2021.00006
- [24] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: a distributed framework for emerging AI applications. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (Carlsbad, CA, USA) (OSDI'18)*. USENIX Association, USA, 561–577.
- [25] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).
- [26] James C. Phillips, Rosemary Braun, Wei Wang, James Gumbart, Emad Tajkhorshid, Elizabeth Villa, Christophe Chipot, Robert D. Skeel, Laxmikant Kalé, and Klaus Schulten. 2005. Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry* 26, 16 (2005), 1781–1802. arXiv:<https://onlinelibrary.wiley.com/doi/pdf/10.1002/jcc.20289> doi:10.1002/jcc.20289
- [27] James C. Phillips, David J. Hardy, Julio D. C. Maia, John E. Stone, JoÃ£o V. Ribeiro, Rafael C. Bernardi, Ronak Buch, Giacomo Fiorin, JÃ©rÃ©me HÃ©nin, Wei Jiang, Ryan McGreevy, Marcelo C. R. Melo, Brian K. Radak, Robert D. Skeel, Abhishek Singhary, Yi Wang, BenoÃ®t Roux, Aleksei Aksimentiev, Zaida Luthey-Schulten, Laxmikant V. KalÃ©, Klaus Schulten, Christophe Chipot, and

- Emad Tajkhorshid. 2020. Scalable molecular dynamics on CPU and GPU architectures with NAMD. *The Journal of Chemical Physics* 153, 4 (2020), 044130. arXiv:<https://doi.org/10.1063/5.0014475> doi:10.1063/5.0014475
- [28] Leah Shalev, Hani Ayoub, Nafees Bshara, and Erez Sabbag. 2020. A Cloud-Optimized Transport Protocol for Elastic and Scalable HPC. *IEEE Micro* 40, 6 (2020), 67–73. doi:10.1109/MM.2020.3016891

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

8 Overview of Contributions and Artifacts

8.1 Paper's Main Contributions

- C_1 A Charm++ build supporting shrink/expansion with the MPI communication layer.
- C_2 A modified MPI Operator implementation that can run Charm++ applications on a Kubernetes cluster with the 4 scheduling policies described in the paper.
- C_3 A simulator to model the performance of the 4 schedulers studied in this paper.

8.2 Computational Artifacts

- A_1 <https://github.com/adityapb/mpi-operator/tree/aws>
- A_2 <https://github.com/adityapb/mpi-operator/tree/aws/examples/v2beta1/charm/simulation>

Artifact ID	Contribution supported	Related Paper Elements
A_1	C_2	Figure 9 & Table 1
A_2	C_3	Figures 7, 8 & Table 1

9 Artifact Identification

9.1 Computational Artifact A_1

This artifact contains the source code for our modifications to the MPI Operator for running Charm++ applications with the elastic scheduling policy described in the paper. The following are the steps to run the experiment described in Section 4.3.2,

- (1) Set up an AWS EKS cluster with a single node group consisting of 4 c6g.4xlarge instances in a single subnet and a cluster placement group.
- (2) To deploy the elastic scheduling policy, edit line 8244 in the file `deploy/v2beta/mpi-operator.yaml` to point to the image `adityapb/mpi-operator:mpi`
- (3) Deploy the MPI Operator using the command -
`kubectl create -f deploy/v2beta/mpi-operator.yaml`
- (4) Generate the job YAML files by running the commands,
`cd examples/v2beta1/charm`
`python generate_jobs.py generate elastic`
- (5) Start the script for monitoring the cluster utilization by running,
`python track_utilization.py elastic`
- (6) Submit the jobs to the cluster by running,
`python generate_jobs.py submit elastic`
- (7) After all the jobs finish running, interrupt the utilization monitoring script. It will create a file called
`pod_utilization_elastic.log`
- (8) To deploy the moldable scheduling policy, edit line 8244 in the file `deploy/v2beta/mpi-operator.yaml` to point to the image `adityapb/mpi-operator:moldable`
- (9) Repeat steps 3 to 7 by replacing `elastic` with `moldable` in all commands to generate the file
`pod_utilization_moldable.log`

- (10) Using the same MPI operator deployment, get the results for the other 2 schedulers by replacing `elastic` with `min` and `max`
- (11) Run the following script to generate Figure 9 and print the actual metrics presented in Table 1,
`python plot_utilization.py`

The estimated execution time for the jobs of each scheduler to finish is around 50 minutes. For the 4 scheduling policies, the total execution time will be close to 4 hours.

9.2 Computational Artifact A_2

This artifact contains the source code for the simulator to model the performance of the 4 scheduling policies studied in this paper. The following are the steps to reproduce the experiments presented in Section 4.3.1,

- (1) Run the following command to generate the plots presented in Figures 7 and 8, and print the simulation metrics presented in Table 1,
`cd examples/v2beta1/charm/simulation`
`python run.py`

The total execution time for the simulator should be less than 5 minutes.