Dynamic Resource Management for Cloud-native Bulk Synchronous Parallel Applications

Evan Wang, Yogesh Barve, Aniruddha Gokhale
Dept of CS, Vanderbilt University
Nashville, TN, USA
evan618@gmail.com,{yogesh.d.barve,a.gokhale}@vanderbilt.edu

Hongyang Sun
Dept of EECS, University of Kansas
Lawrence, KS, USA
hongyang.sun@ku.edu

***Abstract*—Many traditional high-performance computing applications including those that follow the Bulk Synchronous Parallel (BSP) communication paradigm are increasingly being deployed in cloud-native virtualized and multi-tenant container clusters. However, such a shared, virtualized platform limits the degree of control that BSP applications can have in effectively allocating resources. This can adversely impact their performance, particularly when stragglers manifest in individual BSP supersteps. Existing BSP resource management solutions assume the same execution time for individual tasks at every superstep, which is not always the case. To address these limitations, we present a dynamic resource management middleware for cloud-native BSP applications comprising a heuristics algorithm that determines effective resource configurations across multiple supersteps while considering dynamic workloads per superstep, and trading off performance improvements with reconfiguration costs. Moreover, we design dynamic programming and reinforcement learning approaches that can be used as pluggable strategies to determine whether and when to enforce a reconfiguration. Empirical evaluations of our solution show between 10% and 25% improvement in performance over a baseline static approach even in the presence of reconfiguration penalty.**

***Index Terms*—Bulk Synchronous Parallel jobs, Resource management, Cloud-native, workload forecasting**

I. Introduction

In the Bulk Synchronous Parallel (BSP) model [1] [2], work is accomplished in parallel by multiple distributed tasks (e.g., threads, processes, containers), but where the computation results per task must be synchronized periodically in what are known as *supersteps*. A classic real-world implementation of the BSP model is Google's Pregel framework [3]. Pregel is a graph processing framework where computations are performed concurrently at individual vertices. After their computations, vertices can send messages to each other, which can be accessed at the beginning of the next superstep. The wait time for the vertices to send their messages forms the synchronization barrier of the BSP model. Another use case of the BSP model is predictive digital twins, which is a digital copy of a physical system. By running simulations on the digital twin, one can perform system analysis and make predictions about the behavior of its physical counterpart. Since the systems are very large, their digital twins often comprise multiple, interacting co-simulations that follow the BSP model.

Figure 1 illustrates the general architecture of a BSP computation comprising multiple executing tasks (also known as federates) that synchronize at a barrier after every superstep. An application comprises a sequence of such supersteps. Since the next superstep cannot start until all tasks from the previous

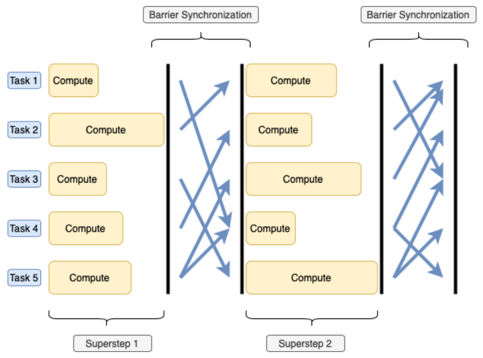


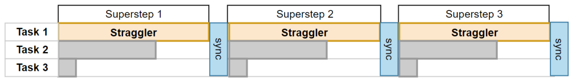
Figure 1: Bulk Synchronous Parallel (BSP) application.

step have completed and synchronized their results, the overall duration of the application execution depends on the slowest task (i.e., straggler) at each superstep. Thus, a resource manager should prioritize the resources for these stragglers (e.g., by giving them more CPU cores, memory, etc), thereby decreasing their execution times and reducing the wait time of faster tasks.

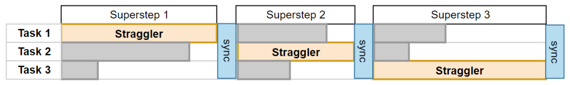
One challenge for BSP applications is to determine the right resource configuration, i.e., a partition of the available resources among all the tasks. Such a resource configuration is then used to gang-schedule [4] the tasks onto the computing resources. Although recent efforts address this problem [5], they make the simplifying assumption that the workloads and execution times of the tasks of a BSP application remain the same in all the supersteps. Thus, a resource configuration is computed only once and applied in every superstep thereafter.

This paper overcomes this significant limitation in the prior work by considering *dynamic* BSP applications with different workloads and execution times for individual tasks in successive supersteps. Hence, each superstep may give rise to a different straggler. Figure 2 illustrates the assumption made in the prior work and the relaxation of this assumption in this work.

The problem is further complicated by the fact that BSP applications are increasingly being transformed into cloud-native environments for easy deployment in container clusters, such



(a) Static BSP application (prior work)



(b) Dynamic BSP application (this research)

Figure 2: Examples of (a) a static BSP application vs. (b) a dynamic BSP application.

as those managed using Kubernetes. This move to the cloud, however, significantly hinders control over resource allocation strategies that has hitherto been under user control when these applications were hosted in controlled environments, such as traditional high-performance clusters. The cloud, in contrast, is a virtualized, shared environment where multi-tenancy is common and hence, delivering predictable performance via dynamic resource management is even more challenging.

To address these challenges, this paper makes the following contributions:

- We describe a data-driven approach to determine the effective resource configurations for individual tasks of a BSP application that simultaneously minimizes straggler behavior and resource reconfiguration costs;
- We present a pluggable middleware design where our resource reconfiguration planner can be strategized, e.g., by using dynamic programming or reinforcement learning, to meet our objectives;
- We provide an extensive empirical evaluation of our solution. The results show between 10% and 25% improvement in performance over a baseline static approach even in the presence of checkpointing penalty.

The rest of this paper is organized as follows: Section II compares our work with relevant prior efforts; Section III defines the problem formally; Section IV provides details of our approach; Section V describes the pluggable middleware design; Section VI provides the empirical evaluation results; and finally Section VII provides concluding remarks alluding to limitations and future work.

II. Related Work

We present a sampling of prior efforts related to our research along the directions of resource configurations for cloud-based BSP applications and model-based resource management.

A. Cloud-based Resource Configurations for BSP Jobs

In [6], the authors presented a Kubernetes co-simulation platform for cloud computing environments but this work does not use a gang-scheduling approach like we do. Authors of [7] proposed a hierarchical virtual machine (VM)-based workload-aware resource allocation for the federates of a simulation in cloud data center. However, the workload characteristics of the federates were restricted to single-threaded applications. In [8], authors presented priority-based scheduling of parallel jobs on high- and low-priority VMs to avoid under utilization of cloud resources and improve parallel job performance thereby resulting in only two categories of resource assignment. In [9], authors presented different vector bin packing algorithms and first fit decreasing (FFD) heuristics to allocate resources in a shared cluster, however, only for static workloads.

In [10], authors proposed a process migration scheme for dynamic load balancing to mitigate straggler scenarios in a BSP job. This load balancing technique utilizes idle times on workstations and migrates the processes when the load of the machines increases. The authors of [11], [12] presented MigBSP which also applies process migration techniques for load balancing tasks in a BSP job. It takes into account computation, communication and memory metrics of the running process to decide whether the process needs to be migrated to a different node in a cluster or not. Although this work allows more resource types than ours, it does not operate in shared, multi-tenant container clusters. A recent effort [13] has similar objectives as ours. It supports high performance applications with a dataflow architecture. The authors define a novel modeling approach that incorporates expected workloads, dataset sizes, resource scale-out impacts, and execution runtimes to make decisions on resource configurations. The BSP model we use is not a dataflow, however, our work can benefit from the additional parameters used by these authors in constructing the models.

This paper overcomes the limitations in our prior work on EXPPO [5], which assumes that tasks in each superstep have the same workload and computation cost. This is a significant limitation that we overcome with dynamic resource management approaches.

B. Model-based Resource Management

A significant amount of literature exists that uses machine-learned models of workload patterns or resource impact on application performance to conduct dynamic resource management. A common limitation in these works is that none of them are tailored to address the dynamic resource management of distributed and cloud-native BSP applications.

Our prior work in [14] presented a strategy for making reconfiguration plans that minimize a given cost. The cost is defined as a combination of response time that meets service-level objectives, cost of changing the number of machines, and cost of the machines themselves. The cost of changing the number of machines is similar to the checkpoint or reconfiguration cost in our problem. Like our prior work, our current work also uses time-series methods to make predictions about future workloads. Based on these predictions, prior work used a model-predictive approach based on receding horizon [15] to identify the optimal configuration.

The original work in [15] shows how the well-known receding horizon-based model-predictive control is applied to fine tune the behavior of systems, where the idea is demonstrated on two case studies: to manage the power consumed by servers and a digital signal processing system. Like [14], our work also leverages the idea from [15] where a receding horizon of N supersteps is used as part of our dynamic resource management strategy for BSP tasks.

The authors of [16] proposed Ernest, which predicts the performance of cloud-based applications for a given resource type and accordingly chooses the optimal resource configuration for the job. Such strategies can become part of future considerations for our work. However, since our work relies on the use of container clusters managed by Kubernetes, we are oblivious to the underlying physical hardware. Thus, additional efforts will be required to incorporate similar ideas.

III. Problem Statement

We use a brief description of a BSP use case to state the assumptions made in this paper and present a mathematical formulation for the problem we address.

A. Use Case of a Bulk Synchronous Parallel Application

We briefly describe a concrete BSP application use case to better articulate the challenges and the make it easier to understand the problem statement and our solution approach.

It is often the case that the algorithms and strategies designed to improve the functionality or various other properties of large-scale and complex systems require validation via simulations prior to actual deployment. However, seldom do such systems come with a single simulator that can handle all the behaviors and dynamics of a given system. Consider as an example the need to test algorithms for traffic congestion and safety enhancement in vehicular networks comprising both autonomous and human-driven vehicles. To validate the different algorithms will require simulations of vehicular movement, communication networking, the physics of the different entities involved and their interactions, and even simulations of human-computer interfaces.

No single simulation framework is available to test such use cases, which calls for co-simulations that are a classic case of a BSP application. In our use case, we may use vehicular simulators, such as SUMO or CARLA; communication networking simulators, such as Omnet++/INET or NS3; additional 5G networking simulators for user equipment, radio access networks and base stations; physics simulators such as Matlab; and other simulations for human behaviors.

In such a BSP application, the computational requirements of individual simulators in each superstep of the co-simulation will very likely be different depending on the property being simulated by each simulator within that superstep and the changing dynamics of the system. Hence, to ensure that the BSP application as a whole executes efficiently, reliably and with good performance, will require dynamic resource allocations to different simulators.

This paper addresses these challenges. To that end, we first outline the assumptions we make before formulating the problem and presenting our solution.

B. Assumptions

Deployment Environment: Considering the trends in application deployment strategies, we assume that BSP applications are deployed in multi-tenant cloud-hosted container clusters managed by frameworks such as Kubernetes (K8s). For instance, the different simulators from our use case can execute inside K8s pods as part of the cloud-native deployment of the BSP application. Our middleware supplies resource configurations for the BSP tasks to the K8s scheduler.

Dynamic Workloads: Unlike EXPPO [5], we assume that the tasks of a BSP application can have dynamic workloads, i.e., computations of different lengths at each superstep as explained in the context of our use case. A larger workload implies a longer computation time for that task, which calls for dynamic resource allocation at each superstep.

Reconfiguration/Checkpointing Cost: We assume some cost associated with resource reconfiguration. Changing the resource allocation implies altering the resources for the container. If multiple containers on a given node all require more CPU cores, that node may no longer have enough cores to satisfy the requirements and some containers would need to be migrated. This would require tearing down the containers and scheduling them onto a new node, which incurs an overhead. Additionally, some BSP use cases may require preserving the internal state of the task during migration, which requires checkpointing the container, thereby incurring additional overhead. For instance, each of the simulators from our use case may need to checkpoint their state before migration of their pods. For this paper, we assume a constant cost associated with reconfiguration and checkpointing.

Applying a Configuration: Any new resource configuration must be applied in the barrier synchronization stage, i.e., when a superstep ends and individual federates synchronize. We assume that on a change in resource configuration, the Kubernetes pod containing the container will automatically be checkpointed. The middleware will pause the application, checkpoint the impacted containers concurrently, and recreate them with the new configuration.

C. Mathematical Formulation

Since the workload on individual tasks of the BSP application can change per superstep and individual containers/pods are allocated on multi-tenant cloud resources, different stragglers may manifest in different supersteps. Thus, identifying the stragglers and adapting the resource configuration in a proactive rather than a reactive manner is important. In doing so, our solution must be aware of the reconfiguration cost while ensuring that the generated resource configurations per superstep result in the minimum overall execution time of the application.

Formally, suppose the BSP application consists of n tasks (e.g., 4 or 5 in our use case) and m supersteps in total. Let R denote the total amount of cloud resources available (in discrete shares). At each superstep j, where $1 \leq j \leq m$, let $config^{(j)} = [r_1^{(j)}, r_2^{(j)}, \ldots, r_n^{(j)}]$ denote the *resource configuration*, i.e., the allocated resource shares for all tasks at that superstep. For each task i, let $t_i^{(j)}(r_i^{(j)})$ denote the resulting execution time. The straggler of superstep j is, therefore, the task that has the largest execution time, i.e., $t_{\max}^{(j)} = \max_i t_i^{(j)}(r_i^{(j)})$. Further, if the resource configuration of the superstep is different from that of the previous step, a constant reconfiguration/checkpointing cost C will be incurred.

The problem then is to find resource configurations for all supersteps that minimize the overall execution time of the application while considering both the stragglers and the reconfiguration/checkpointing cost:

$$\min \left(\sum_{j=1}^{m} t_{\max}^{(j)} + \sum_{j=2}^{m} \theta^{(j)} \cdot C \right)$$
$$\text{s.t.} \sum_{i=1}^{n} r_i^{(j)} \leq R, \ \forall j = 1, \ldots, m$$

Here, $\theta^{(j)} = 1$ if $config^{(j)} \neq config^{(j-1)}$, i.e., a reconfiguration is done at superstep j. The constraint ensures that the total available resources are not exceeded at all supersteps.

IV. Methodology

This section describes our methodology to solve the problem stated in Section III. To that end, we define a four-step approach shown in Figure 3 and described in detail below.

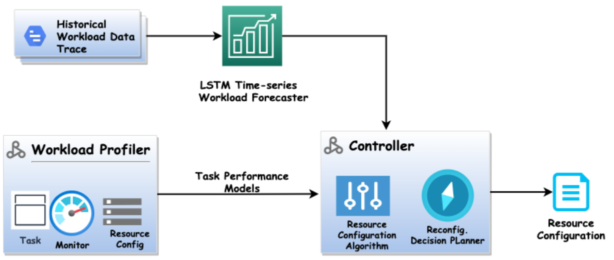

Historical Workload Data Trace
LSTM Time-series Workload Forecaster
Workload Profiler
Task
Monitor
Resource Config
Task Performance Models
Controller
Resource Configuration Algorithm
Reconfig. Decision PLanner
Resource Configuration

Figure 3: A four-step design approach.

1) A workload profiler that builds performance models for each task of the BSP application. These models can predict the execution time of a specific task.
2) A time-series model for each task to predict its future workloads.
3) An algorithm to define a resource configuration for the next N supersteps, where N is configurable.
4) A planner with pluggable strategies that decides how frequently resource reconfiguration should be done.

We describe these steps in detail using a mock BSP application that we created using synthetic datasets. We did not use the concrete use case from Section III-A to explain our solution because it was hard to reproduce the changing computational needs of each federate in successive supersteps of the BSP application and obtain the necessary straggler behaviors in a controlled manner. Instead, this was easier to do with a mock BSP application.

We choose synthetic datasets for the purposes of illustrating the approach and ease of reproducibility. Moreover, by showing the approach on a single concrete BSP application would have come across to the reader as not being generic but rather a point solution. In practice, we expect BSP application developers to provide the artifacts described below to our middleware prior to actual deployment. The rest of this section explains the four steps in detail.

A. Step 1: Workload Profiling

We profile each task of a BSP application for a variety of resource configurations and workload sizes. This provides the middleware information on how tasks perform with different workload and resource allocation combinations, which in turn is used to inform resource management decisions.

1) Benchmark Tasks and Profiling: To highlight the impacts of resource allocation and workload size, we use eight synthetic tasks from the *stress-ng* benchmarks [17] as shown in Table I.

Table I: The *stress-ng* Benchmarks used as BSP Tasks.

Task Name	Description
affinity	Rapidly change CPU Affinity
atomic	Exercise GCC __atomic_*() built in operations
bsearch	Performs binary search on sorted array
cap	Make calls to capget(2) system calls
chmod	Change file mode bits of a single file with chmod(2) and fchmod(2) system calls
memcpy	Copy data from shared region to a buffer
vecmath	Perform calculations on 128 bit vectors
zero	Make reads on /dev/zero

Each task (synthetic or real) is profiled with different CPU configurations: $1000M$ to $9000M$ CPU shares (with $500M$ increments). Here, $1000M$ CPU shares is equivalent to $1vCPU$.[1] We define the workload size of a task as an integer between 10 and 50 (with an increment of 5), representing a multiplier on the number of operations that task must execute. Thus, a workload size of 50 means that a task must execute 5 times as many operations as a workload size of 10. For each CPU configuration and workload size, the task is profiled a few times in K8s pods and the average execution time is recorded.

2) Utilizing Benchmarking Results: Figure 4 shows the heat maps on the profiling results for three tasks. The heatmaps for the remaining tasks show similar trends. As we would expect, larger workload sizes lead to higher durations, while more CPU shares lead to lower durations. By analyzing the profiling results, we can learn how a task responds to changes in CPU resources and workload sizes. The data is then used to train a model for each task's execution time, which in turn can predict task performance given any combination of workload size and CPU resources.

For the model prediction logic, we have used the SciPy interpolation libraries [18].

B. Step 2: Workload Forecasting

In our data-driven approach, training a model to predict a task's performance for a given workload and resource configuration is a necessary but not a sufficient step. To proactively configure resources in a BSP application, we also need to predict the task's workload in future supersteps. Thus,

[1]Note that in this work we have shown the impact of CPU resources only but the work can readily be extended to cover other resource types.

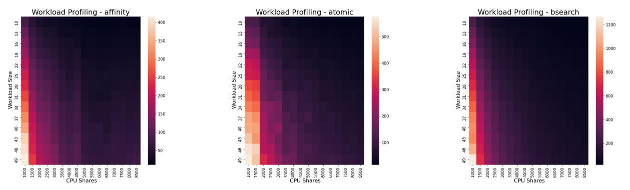


Figure 4: Workload profiling heatmaps shown for three of the eight synthetic tasks. Other heatmaps showed trends along expected lines.

we need variable workload traces to mimic workloads for our synthetic tasks. To that end, we have used time-series datasets from the Numenta Anomaly Benchmark that model fluctuations in workload sizes [19]. These datasets provide real-world time-series variations that exhibit realistic trends and patterns. Each of our eight synthetic tasks from Step 1 is assigned a separate sequence of time-series data for simulating changes in its workload. Every task also has its own time-series model that trains on historical patterns from that task's dataset.

1) Time-series Datasets: We use the following three time-series datasets from the Numenta Anomaly Benchmark with variability in workload patterns as depicted in Figure 5:

- **NYC Taxi:** Number of passengers for NYC taxi cabs. Data is recorded every 30 minutes.
- **Ambient System Temperature Failure:** The temperature in an office where data is recorded hourly.
- **Artificial Daily Small Noise:** Artificially-generated dataset which fluctuates daily with added noise.

Some of these datasets are split into multiple parts to be used by different tasks. For instance, both *nyc_taxi_1* and *nyc_taxi_2* originate from the NYC Taxi dataset but come from disjoint sections, and form individual datasets for different tasks. Each task's datasets has roughly 2,000 data points. Table II shows the assignment of these datasets to the synthetic tasks.

Table II: Each Synthetic Task and its corresponding Time-series Dataset used to model its Workload Fluctuations.

Task Name	Time-Series Dataset
affinity	nyc_taxi_1
atomic	nyc_taxi_2
bsearch	nyc_taxi_3
cap	nyc_taxi_4
chmod	art_daily_small_noise
memcpy	ambient_temperature_system_failure_1
vecmath	ambient_temperature_system_failure_2
zero	ambient_temperature_system_failure_3

2) Forecasting Methodology: We use Long Short-Term Memory (LSTM) neural networks from the Keras suite for forecasting workloads from the provided time-series. For each task, we process the corresponding dataset and train the model using the following steps [20]:

1) Convert dataset into a sequence of differences, where each value x_i is converted into the difference with previous value, i.e., $y_i = x_i - x_{i-1}$.
2) Normalize all difference values y_i's between -1 and 1.
3) Split the dataset into two halves, with the first half used for training and the second half for testing.
4) Convert training and test sets into supervised learning datasets. For each y_i in the sequence, create a mapping that has y_i as the input and $(y_{i+1}, ..., y_{i+s})$ as the output, where s is the size of the forecasting window.
5) Train an LSTM model on the (input, output) pairs from the supervised training set.
6) Make predictions on the supervised test set.
7) Invert the transformations in (1) and (2) to get predictions for the actual dataset.

For every point in each dataset, we now have the predicted values for the next s points, where s is the size of the forecasting window and is configurable. Although a larger forecasting window can provide more information, the prediction accuracy will suffer. We use a maximum forecasting window of $s = 20$.

3) Forecasting Error: To convert our predictions into actual workload sizes, we normalize them within our workload size range of $(10, 50)$. For each dataset and each look-ahead window s from 1 to 20, we measure the forecasting error by calculating the Root Mean Square Error (RMSE) between our predicted value and the actual value s steps in the future. Figure 6 plots the forecasting error for each dataset. As expected, the error begins relatively low but increases when we attempt to predict further into the future.

C. Step 3: Resource Configuration over an N-step Horizon

At the start of every superstep, the middleware needs to decide whether to keep the current configuration: keeping it will preserve the existing containers in the next superstep, however, performance may suffer if workloads change; whereas changing it will require a new configuration. An exponential number of ways to allocate resources among all tasks exist, so an exhaustive search is infeasible. Note that reconfiguration also incurs a cost as containers may need to be migrated and their states need to be checkpointed. As noted in Section III, reconfiguration also incurs a cost as containers may need to migrate and state checkpointed.

If the middleware decides to keep a configuration for multiple supersteps, it must optimize the configuration over that entire timeframe, i.e., minimize the cumulative duration over all these supersteps. An algorithm for creating such an optimal resource configuration needs to take into account the workload conditions at each superstep in that timeframe. To that end, Algorithm 1 extends the static configuration approach from [5] and greedily computes a resource configuration for dynamic workloads over an N-step horizon, where N is a configurable parameter.

The basis for the algorithm in [5] is to begin with a default configuration, $config = [1, 1, \ldots, 1]$, which allocates one unit of resource to each task and repeatedly add resources to the predicted slowest task until all resources are used. The

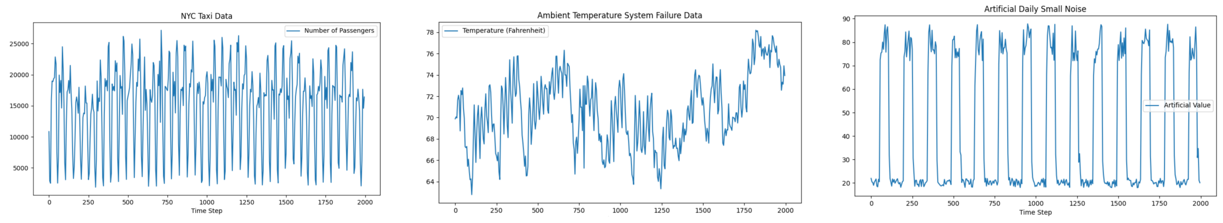


Figure 5: Workload patterns for the three time-series datasets.

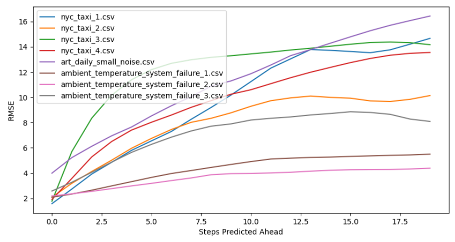


Figure 6: Forecasting Error for each Time-series Dataset with different Look-ahead Windows.

Algorithm 1: Create an N-step Resource Configuration

Input: predicted workloads, N-step timeframe
Output: resource configuration for the timeframe

```
1  $config \longleftarrow [1, 1, \ldots, 1]$
2  while $resources\ available$ do
3      $slowest\_tasks \leftarrow$ get_slowest_tasks($config$, $N$, $predicted\_workloads$)
4      for $t = 1, 2, \ldots, N$ do
5          $slowest\_task \leftarrow slowest\_tasks[t]$
6          $improvements[slowest\_task]$ += $discount(t)$* improvement($slowest\_task$, $config$, $t$, $predicted\_workloads$)
7      end
8      $most\_improved\_task \leftarrow$ argmax($improvements$)
9      $config[most\_improved\_task]$++
10 end
```

same configuration is then kept throughout all supersteps. For our case of dynamic workloads, there may be a different "slowest task" at each superstep, in contrast to [5]. To deal with this, we also begin with a default configuration of one resource per task (Line 1). We then aggregate all tasks that are predicted to be the slowest at each superstep (Line 3). Since these tasks (or stragglers) will force other tasks to wait, we need to provision more resources towards them. We use their predicted workloads and task performance models to estimate the cumulative improvement by incrementing their provisioned resources. We then choose the straggler that has the largest improvement, and increment its resource allocation (Line 9).

Note that our algorithm prioritizes performance improvements in earlier supersteps over later ones in the future. Since the forecasting error will become increasingly inaccurate for later supersteps (as shown in Figure 6), this is accounted for by applying a discount factor for "improvements" at each superstep (Line 6). We use an exponentially decaying discount factor of the form $discount(t) = 0.95^t$ for each superstep t in the timeframe.

D. Step 4: Reconfiguration Decision Planner

While the previous step computes a resource configuration for an N-step timeframe, changes in predicted workloads could lead us to abandon the configuration earlier than anticipated. Hence, a decision we must make before the start of each superstep is whether to keep the current configuration or to compute a new one using Algorithm 1. If the decision is to compute a new one, then we need to determine for how much duration should the configuration be optimized for.

Since the longest timeframe the middleware could keep a configuration is s (the size of our forecasting window), there are $s+1$ possible actions we can take. We define these actions as $\{a_0, a_1, \ldots, a_s\}$, where a_0 denotes keeping the current configuration and a_j $(1 \leq j \leq s)$ denotes creating a new configuration optimized for the next j supersteps. We further define a *decision plan* as a sequence of actions to take at each superstep. Our middleware is designed to accept different algorithms as pluggable decision planners. Three decision planning strategies are designed and discussed in Section V.

V. Pluggable Strategies for Resource Reconfiguration Decision Planner

In this section we design three resource reconfiguration strategies: a static strategy, a dynamic strategy (with a model predictive control variation), and a reinforcement learning strategy, which can be used as pluggable decision planners for our middleware. Each strategy knows the predictions for the tasks' workloads (up to the next $s = 20$ supersteps), our current resource configuration, and performance models for each task of the BSP application. At each superstep, the strategy outputs an action from $\{a_0, a_1, \ldots, a_s\}$, which informs the system on the planning of next resource configuration.

A. Static Window Strategy

Our first strategy, called the static window strategy, is a naïve approach that automatically updates the configuration every K supersteps, where $K \leq s$ is the size of the static window. The new configuration is optimized for the next K supersteps (using Algorithm 1) and kept for exactly K supersteps. This means that we take action a_k every K supersteps and a_0 (i.e., keep the current configuration) the rest of the time.

The optimal size of the static window depends on the benefit of reconfiguration compared to the checkpoint cost. High checkpoint costs bias towards larger static windows, and vice versa. This static window strategy provides some adaptation to changes in task workloads, but it will often make sub-optimal decisions. For example, it automatically discards the resource configuration at the end of the window even if it is still performing well. Conversely, if the predictions are inaccurate, we can be stuck with a poor configuration that should have been abandoned earlier.

B. Dynamic Strategies

Recall that at each superstep t, we have $s+1$ possible actions to take: either keep the current configuration (a_0) or create a new configuration for anywhere between a single superstep (a_1) and the end of the forecasting window (a_s). If the BSP application consists of a large number of supersteps in total, we will also have a large number of reconfiguration plans to consider. To limit the search space, we first develop a dynamic window strategy, in which we assume that if action a_j (where $1 \leq j \leq s$) is taken, then we will always keep that configuration for exactly j supersteps, i.e., action a_j followed by action a_0 exactly $j-1$ times. This is similar to the static window strategy but the size of the configuration window is dynamically computed. We then develop a Model Predictive Control (MPC) variation, where a new configuration could be computed at any superstep but with a smaller time horizon for better predictive performance.

1) Dynamic Window Strategy: At any superstep t where a new configuration is needed (e.g., at the start of the application or when the existing configuration expires), the goal is to find an action a_{j^*}, where $1 \leq j^* \leq s$, that minimizes the duration of the application from step t to step $t+s-1$ (the end of the forecasting window).[2] We define q_t to be the resulting minimum duration, which can be computed via dynamic programming.

Let $d_t(j)$ denote the duration of the application from step t to step $t+s-1$ if action a_j is taken. We consider two cases:

Case (1): If action a_j is taken for any $1 \leq j \leq s-1$, then a reconfiguration needs to be done again at step $t+j$. Thus, the duration $d_t(j)$ can be expressed as:

$$d_t(j) = d_{t,t+j-1} + q_{t+j} + C \tag{1}$$

where $d_{t,t+j-1}$ denotes the execution time of the application from step t to step $t+j-1$ under the configuration decided by a_j using Algorithm 1, q_{t+j} denotes the minimum duration of the application starting from a reconfiguration at step $t+j$ until the end of the forecasting window, and C denotes the checkpoint cost.

Case (2): If action a_s is taken, then we are creating a configuration for the entire forecasting window (i.e., from step t to step $t+s-1$). Thus, there will be no checkpoint cost. The duration $d_t(s)$ in this case is simply given by:

$$d_t(s) = d_{t,t+s-1} \tag{2}$$

[2]If the application will complete before the end of the forecasting window (i.e., $m < t+s-1$), we can adjust the forecasting window to contain only the remaining supersteps of the application.

We can then compute q_t as the minimum duration among all of these possibilities:

$$q_t = \min_{1 \leq j \leq s} d_t(j) \tag{3}$$

and the optimal action a_{j^*} is the one that achieves the above minimum, i.e., $j^* = \text{argmin}_j d_t(j)$. The configuration decided by a_{j*} using Algorithm 1 will then be kept for exactly j^* steps before the next configuration is computed.

2) Model Predictive Control (MPC): The effectiveness of the dynamic window strategy hinges on the optimality of configurations generated by Algorithm 1. However, as shown in Figure 6, the forecasting error increases as the time horizon increases and hence Algorithm 1 will not generate optimal configurations for a given superstep. To address this limitation, we present a Model Predictive Control (MPC) strategy. Compared to the dynamic window strategy, our MPC variation has several major differences as described below.

First, we consider a limited horizon where forecasting predictions have better accuracy. In this work, we select the size of the horizon to be $s' = 10$ supersteps. Second, we update the configuration plan at each superstep of the application. Specifically, at superstep t, we compute a new configuration plan that optimizes the execution time over the limited horizon (i.e., from step t to step $t+s'-1$) and then take the first action of the plan. Upon reaching step $t+1$, we update the predictions and the horizon shifts one step further into the future (i.e., from $t+1$ to $t+s'$). We then recompute the configuration plan and again take the first action of that plan. This process repeats at every superstep until the end of the application.

Finally, in the MPC strategy, we have the option of keeping the current configuration at any superstep (if one exists), i.e., by taking action a_0. At superstep t, let q'_t denote the minimum duration of the application in the horizon (i.e., from t to $t+s'-1$) if we keep the current configuration. Based on the dynamic programming approach described in Section V-B1, we can compute q'_t as:

$$q'_t = \min\left(\min_{1 \leq j \leq s'-1}\left(d'_{t,t+j-1} + q_{t+j} + C\right), d'_{t,t+s'-1}\right) \tag{4}$$

Here, j represents how long we will keep the current configuration, which could be anywhere from 1 superstep to s' supersteps. If $q'_t < q_t + C$, we get better performance by keeping the current configuration than creating a new one. In this case, we should take action a_0 instead of a_{j^*} at superstep t.

C. Reinforcement Learning Strategy

If we have perfect knowledge of the future workloads, the dynamic strategies from Section V-B can be used to compute the entirety of our configuration plan. However, it suffers when dealing with forecasting errors. Even though the MPC strategy adjusts the algorithm to work with forecasting predictions, it does not consider that predictions at some supersteps can have higher errors than others. Although this can partially be dealt with by discounting longer configuration windows (which rely on more distant predictions), it does not allow the model to learn from the actual forecasting error at each step.

We surmise that this issue could be resolved by reinforcement learning. For our problem, there is a discrete set of actions, i.e., $\{a_0, a_1, \ldots, a_s\}$, and a well-defined reward function, i.e., execution time. Additionally, reinforcement learning could learn from forecasting errors and estimate the trustworthiness of each prediction. This would allow the agent to make decisions based purely on strong predictions. To that end, we set the state as an $(n+1) \times m$ matrix, where n is the longest that we are willing to keep a configuration and m is the number of supersteps that we are looking ahead. Thus, we have $n+1$ possible actions $\{a_0, a_1, \ldots, a_n\}$ and each row corresponds to an action. In this matrix, the value at (row x, column y) is the predicted duration y steps from the current step with the configuration that is optimized for the next $x-1$ steps or resulting from action a_{x-1}. For example, consider (row 1, column 5) of a state. Row 1 corresponds to the configuration resulting from action a_0 or the current configuration. Column 5 means we are looking at the execution 5 supersteps from now. Therefore, this element represents the execution time 5 supersteps in the future using the current configuration.

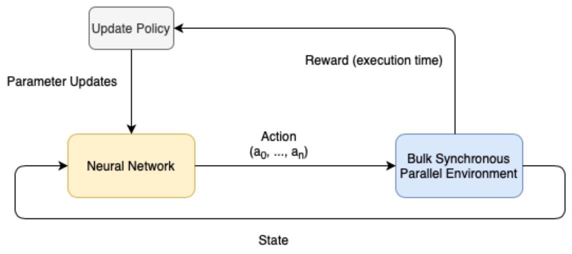


Figure 7: System for training a Deep Q-Learning agent. The neural network accepts a state and computes an action. The action is then fed into the environment, which returns the next state and the execution time (used as the reward for training).

We implemented the above strategy using OpenAI Gym and used a Deep Q-Learning agent from the stable baselines package [21], [22], [23]. The approach is depicted in Figure 7.

VI. Experimental Results

A. Experimental Setup

Our experimental setup uses four virtual machines that form the nodes of our Kubernetes (K8s) cluster instantiated on a powerful bare-metal instance of the Chameleon Cloud [24]. A single instance helps to create a shared environment and enables performance data collection at the host level. The K8s cluster specifications are shown in Table III.

The BSP tasks are run in K8s pods using custom Docker containers. Resource requirements and benchmark-specific parameters are specified in the job template. The BSP model is realized using Apache ZooKeeper (ZK) [25], where the ZK barrier is used for task synchronization between successive supersteps. Workload sizes are passed to individual tasks using ZK queues.

Our reconfiguration strategies are tested on a synthetic BSP application comprising eight tasks from the stress-ng benchmark as shown in Table I. Since the individual tasks take several minutes to run in each superstep leading to very long computation times for each experiment, particularly when there is a large number of supersteps, we simulated the task logic inside the K8s pods, where we used the task performance models from Section IV to estimate task run times by interpolating the data points from our workload profiles. This allows us to test our reconfiguration strategies on large numbers of supersteps in a much shorter timeframe.

Table III: Specifications of Experimental Kubernetes Cluster.

Kubernetes Cluster Specifications	
Number of Nodes	4
CPU Cores per Node	10vCPU
Memory per Node	10GB
Operating System	Ubuntu 20.04
Kubernetes Distribution	Microk8s

The BSP application is evaluated for 400 supersteps, where task workloads are modeled using the datasets in Table II. The checkpoint cost for these experiments is set to 15 seconds.

B. Evaluated Strategies

We evaluated the following strategies:

- **EXPPO:** This is the approach from [5], where task workloads are assumed to be constant in each superstep. We used the average workloads for each task to create a configuration by EXPPO, which is then used at every superstep of the execution. This strategy is used as the baseline for performance comparison.
- **Static Window (of size K):** This is the static window strategy (Section V-A), where a new configuration is created every K supersteps. In the experiment, we tested static windows from size 1 to size 9.
- **Dynamic MPC:** This is the dynamic model predictive control strategy (Section V-B2) that uses a horizon of 10 supersteps. It always outperforms the dynamic window strategy (Section V-B1), so the plain dynamic strategy is not evaluated in our experiments.
- **Deep Q-Learning:** This is the reinforcement learning strategy (Section V-C). We trained an agent with a set of 4 actions $\{a_0, a_1, a_2, a_3\}$ and a look-ahead window of 6 supersteps resulting in a Q-table with 4×6 entries.

C. Results with Known Workloads

Since many strategies are influenced by forecasting errors, we first evaluate them when future workloads in the forecasting window are known (i.e., zero forecasting error). This is produced by substituting the predicted workloads with the real workloads and testing each strategy.

Figure 8 shows the execution times for the Static Window and Dynamic MPC strategies in terms of their percentage improvements over the EXPPO baseline, which takes approximately 65,126 seconds to complete the execution. The Deep Q-Learning strategy is not evaluated here since it is specifically designed to cope with forecasting errors. As expected, the

Static Window 1 strategy produces the best result (with ∼37% improvement over EXPPO) when we do not account for any checkpoint cost from reconfiguration. This is the strategy that creates a new configuration at every superstep. Therefore, it is also creating the most number of checkpoints, thus incurring a high checkpoint penalty. As the window size increases, the number of checkpoints reduces along with their cost, but at the expense of sub-optimal configuration for each superstep. The Dynamic MPC strategy performs the best when accounting for checkpoint cost. It improves upon EXPPO by around 31% and has an execution time that is 816 seconds faster than the best Static Window strategy (with window size 2).

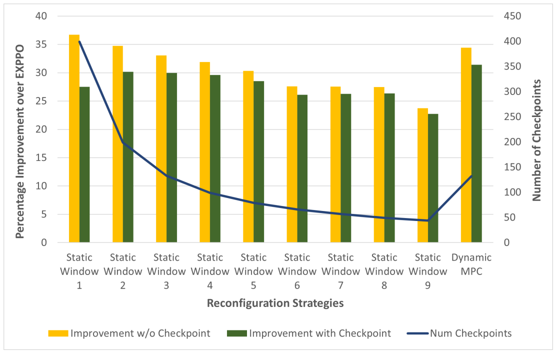


Figure 8: Results for the Static Window and Dynamic MPC strategies assuming perfect knowledge of future workloads. The left Y-axis shows the percentage improvement in execution time with and without checkpoint penalty over EXPPO (which takes approximately 65,126 seconds), and the right Y-axis shows the number of checkpoints.

D. Results with Model Predictions

We then evaluate all the strategies using model predictions. Each strategy now has to make decisions based on the predicted workloads. The results displayed in Figure 9 showcase how the strategies handle forecasting errors. Once again, all strategies outperform the EXPPO strategy, which never adapts the configuration. However, for the case when workloads are predicted, the non-EXPPO strategies all perform worse compared to their own performance when future workloads are known. In this experiment, the Dynamic MPC strategy improves upon EXPPO by around 24% and is 467 seconds faster than the best Static Window strategy (with window size 3) when considering checkpoint cost. It also reconfigures more often than in the previous experiment (194 vs. 132 times), showing that it has to frequently abandon poor configurations that result from inaccurate predictions.

The Deep Q-Learning strategy has the best performance with 24.4% improvement over EXPPO, is 159 seconds faster than the dynamic MPC strategy and takes 40 fewer checkpoints (194 vs. 154). The improvement likely stems from the shorter prediction window (6 time steps) and the smaller set of actions $\{a_0, a_1, a_2, a_3\}$ used to train the reinforcement learning strategy, which makes it less vulnerable to prediction errors.

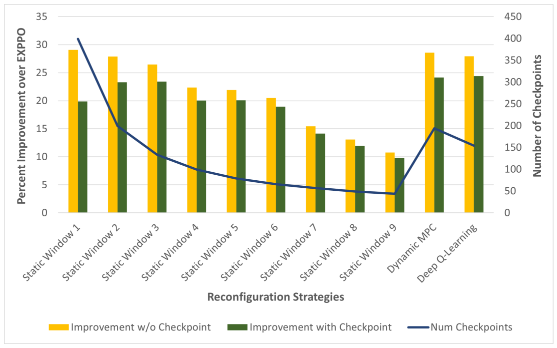


Figure 9: Results for each strategy using forecasted future workloads. The left Y-axis shows the percentage improvement in execution time with and without checkpoint penalty over EXPPO (which takes approximately 65,126 seconds), and the right Y-axis shows the number of checkpoints.

E. Summary and Discussion

Overall, our experimental results show the benefit of adapting resource configurations to workload fluctuations. All strategies perform better than EXPPO, which keeps only a single resource configuration. This improvement is more significant when we have perfect knowledge of the future workloads. In particular, the dynamic MPC strategy has the best performance with this information. However, the benefit of the dynamic strategy is reduced when decisions are made based on less accurate predictions. When the prediction uncertainty is high, reinforcement learning becomes an attractive choice. Finally, we point out that, when using the dynamic strategy, one must also consider the overhead of the algorithm itself. For long-running applications, the algorithm will likely provide more significant performance benefits. However, for short-running applications, the overhead associated with the algorithm may outweigh the performance gains it provides.

VII. Conclusions

This paper presents a dynamic resource management middleware for cloud-native Bulk Synchronous Parallel (BSP) applications, where individual tasks of the BSP application may become stragglers in individual supersteps owing to imperfect allocation of resources in a multi-tenant shared environment and differing workload size/computations of these tasks. This work overcomes limitations in the prior work [5], which assumed that individual tasks of a BSP application illustrate the same workload and computational requirements in every superstep.

The presented approach comprises a four-step resource management process that alleviates the straggler problem while being aware of reconfiguration costs. Empirical results evaluating our solution show between 10% and 25% improvement in performance over a baseline static approach [5] even in the presence of reconfiguration/checkpointing penalty.

A. Discussion

We used synthetic tasks and workloads only to demonstrate the approach. For real-world deployment, a BSP application developer must provide our middleware artifacts including the task structure and their computational needs, traces of workloads, Docker files for deployment in K8s, among others.

We expect our middleware to be deployed alongside the Kubernetes ecosystem. It interacts with the K8s ecosystem to manipulate resource allocations to the task pods. Moreover, we ensure that the middleware executes on a different set of resources than applications to alleviate interference effects.

Our work will benefit the critically important realm of Digital Twins, which are representative BSP applications.

B. Limitations and Future Work

The following limitations in this work become dimensions of our future work.

- **Checkpoint Costs –** Checkpointing support within containers/pods has yet to reach maturity [26]. In the future, we will also include variable checkpointing costs.
- **Memory Considerations –** This work focuses only on the CPU resource. However, some tasks may benefit more from increasing memory than CPU shares. By adding additional resource types to our resource configuration, we could better accommodate these types of tasks.
- **Compare with Reactive Approach –** This research takes a proactive approach towards resource scaling by using predicted workloads for each task. However, we may not be able to get accurate predictions for certain types of tasks. Thus, we will add support for reactive strategies.
- **Evaluate In-place Pod Updates –** Our reconfiguration relies on creating a new configuration from scratch, which requires all containers to be checkpointed and recreated. We will explore resource reconfigurations that do not incur a checkpoint penalty, e.g., if multiple tasks are scheduled onto the same virtual machine, resources may be passed between these tasks without the need to delete any containers.
- **Account for Vertical Interference –** Some tasks may run into interference issues when scheduled onto the same node contending for the same shared resources thereby degrading performance. Future work will include more extensive task profiling to discover which combinations of tasks will experience interference. This could lead to scheduling configurations which inform the cluster about tasks to avoid scheduling together.
- **Dynamic Task Architecture –** This work assumes that the number of tasks per superstep remains the same although their workloads may change. In future work, we will also consider dynamic changes in the number of tasks, which may be the case in digital twin models of complex cyber-physical systems.

The software artifacts in this work are available at github.com/doc-vu/Kube_Gang_Scheduling.

References

[1] L. G. Valiant, "A Bridging Model for Parallel Computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[2] T. Cheatham, A. Fahmy, D. Stefanescu, and L. Valiant, "Bulk Synchronous Parallel Computing - A Paradigm for Transportable Software," in *Tools and Environments for Parallel and Distributed Systems*. Springer, 1996, pp. 61–76.

[3] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A System for Large-scale Graph Processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010, pp. 135–146.

[4] D. G. Feitelson and L. Rudolph, "Parallel Job Scheduling: Issues and Approaches," in *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 1995, pp. 1–18.

[5] Y. D. Barve, H. Neema, Z. Kang, H. Vardhan, H. Sun, and A. Gokhale, "EXPPO: EXecution Performance Profiling and Optimization for CPS Co-simulation-as-a-Service," *Journal of Systems Architecture*, vol. 118, p. 102189, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S138376212100134X

[6] K. Rehman, O. Kipouridis, S. Karnouskos, O. Frendo, H. Dickel, J. Lipps, and N. Verzano, "A Cloud-based Development Environment using HLA and Kubernetes for the Co-simulation of a Corporate Electric Vehicle Fleet," in *2019 IEEE/SICE International Symposium on System Integration (SII)*. IEEE, 2019, pp. 47–54.

[7] Z. Li, X. Li, L. Wang, and W. Cai, "Hierarchical resource management for enhancing performance of large-scale simulations on data centers," in *Proceedings of the 2nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. ACM, 2014, pp. 187–196.

[8] X. Liu, C. Wang, B. B. Zhou, J. Chen, T. Yang, and A. Y. Zomaya, "Priority-based consolidation of parallel workloads in the cloud," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 9, pp. 1874–1883, 2012.

[9] M. Stillwell, D. Schanzenbach, F. Vivien, and H. Casanova, "Resource allocation algorithms for virtualized service hosting platforms," *Journal of Parallel and distributed Computing*, vol. 70, no. 9, pp. 962–974, 2010. [Online]. Available: https://doi.org/10.1016%2Fj.jpdc.2010.05.006

[10] O. Bonorden, "Load balancing in the bulk-synchronous-parallel setting using process migrations," in *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2007, pp. 1–9.

[11] R. da Rosa Righi, L. Graebin, R. B. Avila, P. O. A. Navaux, and L. L. Pilla, "Combining multiple metrics to control bsp process rescheduling in response to resource and application dynamics," in *2011 IEEE 17th International Conference on Parallel and Distributed Systems*. IEEE, 2011, pp. 72–79.

[12] R. da Rosa Righi, L. L. Pilla, A. Carissimi, and P. O. Navaux, "Controlling processes reassignment in bsp applications," in *2008 20th International Symposium on Computer Architecture and High Performance Computing*. IEEE, 2008, pp. 37–44.

[13] D. Scheinert, L. Thamsen, H. Zhu, J. Will, A. Acker, T. Wittkopp, and O. Kao, "Bellamy: Reusing performance models for distributed dataflow jobs across contexts," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2021, pp. 261–270.

[14] N. Roy, A. Dubey, and A. Gokhale, "Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting," in *IEEE 4th International Conference on Cloud Computing*, 2011, pp. 500–507.

[15] S. Abdelwahed, N. Kandasamy, and S. Neema, "A control-based framework for self-managing distributed computing systems," in *Proceedings of the 1st ACM SIGSOFT Workshop on Self-Managed Systems*, ser. WOSS '04. New York, NY, USA: ACM, 2004, p. 3–7.

[16] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient Performance Prediction for {Large-Scale} Advanced Analytics," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016, pp. 363–378.

[17] C. I. King, "Stress-ng," *URL: http://kernel.ubuntu.com/git/cking/stressng-.git/*, 2017.

[18] P. Virtanen, R. Gommers, et al, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[19] A. Lavin and S. Ahmad, "Evaluating Real-time Anomaly Detection Algorithms–The Numenta Anomaly Benchmark," in *IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2015, pp. 38–44.

[20] J. Brownlee, "Multistep Time Series Forecasting with LSTMs in Python," https://machinelearningmastery.com/multi-step-time-series-forecasting-long-short-term-memory-networks-python/, 2017.

[21] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[22] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI Gym," *arXiv preprint arXiv:1606.01540*, 2016.

[23] A. Hill, A. Raffin, E. Maximilian, and et. al, "Stable Baselines," https://github.com/hill-a/stable-baselines, 2018.

[24] K. Keahey, J. Anderson, Z. Zhen, and et. al, "Lessons Learned from the Chameleon Testbed," in *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX, July 2020.

[25] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-Free Coordination for Internet-Scale Systems," in *Proceedings of the USENIX Annual Technical Conference*. USA: USENIX, 2010, p. 11.

[26] "Minimal Checkpointing Support," https://github.com/kubernetes/kubernetes/pull/104907, 2022.