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.**

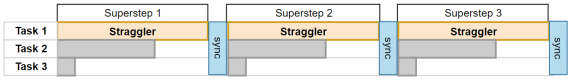
***Index Terms*—Bulk Synchronous Parallel jobs, Resource management, Cloud-native, workload forecasting**

I. Introduction

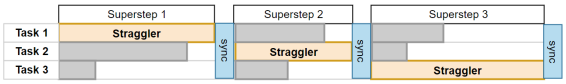
In the Bulk Synchronous Parallel (BSP) model [1] [2], work is accomplished in parallel by multiple distributed tasks (e.g., threads, processes, containers), but where the computation results per task must be synchronized periodically in what are known as *supersteps*. Examples include Google's Pregel [3] and digital twins, which involve interacting co-simulations [4].

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 [5] the tasks onto the computing resources. Although recent efforts address this problem [6], 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 1 illustrates the assumption made in the prior work and the relaxation of this assumption in this work.



(a) Static BSP application (prior work)



(b) Dynamic BSP application (this research)

Figure 1: Examples of (a) a static BSP application vs. (b) a dynamic BSP application.

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 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 describes a data-driven approach that identifies effective resource configurations for individual tasks of a BSP application to minimize both straggler behavior and resource reconfiguration costs. It supports a pluggable design where different reconfiguration planners can be strategized with several examples of such strategies. Empirical evaluations comparing our approach with a baseline static approach show between 10% and 25% improvement in performance over a baseline static approach [6] even in the presence of reconfiguration penalty.

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

979-8-3503-3902-4/23/$31.00 ©2023 IEEE

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 [7], 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 [8] 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. A recent effort [9] 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 [6], 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 [10] presented a strategy for making reconfiguration plans that minimize a given cost. 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 [11] to identify the optimal configuration.

The authors of [12] 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. Methodology

We define a four-step solution approach shown in Figure 2. We use a mock BSP application created using synthetic datasets to describe our approach.

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.

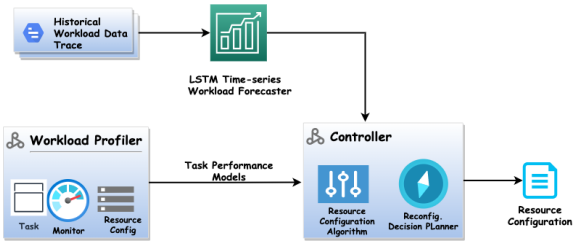


Figure 2: A Four-step Design Approach.

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 [13] 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: The benchmarking results reveal that larger workload sizes lead to higher durations, while more CPU shares lead to lower durations. From this we

[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.

learn how a task responds to changes in CPU resources and workload sizes. These insights are 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 [14].

B. Step 2: Workload Forecasting

To proactively configure resources in a BSP application, we also need to predict the task's workload in future supersteps. Thus, 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 [15]. 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 since they illustrate variability in workload patterns.

- **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 [16]:

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 3 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.

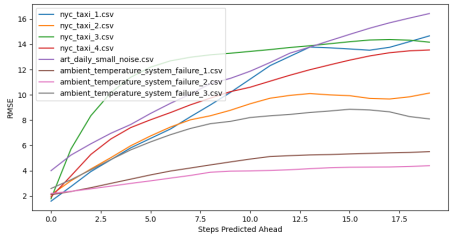


Figure 3: Forecasting Error for each Time-series Dataset with different Look-ahead Windows.

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.

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. Our algorithm extends the static configuration approach from [6] and greedily computes a resource configuration for dynamic workloads over an N-step horizon, where N is a configurable parameter. Our algorithm starts with the minimum amount of resources for each task in a superstep. It then aggregates all tasks that are predicted to be the slowest in a given superstep, and uses a "what-if" analysis approach to determine the improvement due to additional resources given to those tasks. We choose the straggler with the largest improvement to give additional resources. This process then repeats until there are no more available resources left.

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 3), this is accounted for by applying a discount factor for "improvements" at each superstep. 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 our algorithm. 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.

IV. 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 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.

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) Model Predictive Control (MPC): The effectiveness of the dynamic window strategy hinges on the optimality of configurations generated by our algorithm. However, as shown in Figure 3, the forecasting error increases as the time horizon increases and hence the algorithm will not generate optimal configurations for a given superstep. To address this limitation, using MPC we consider a limited horizon where the forecasting predictions have better accuracy. We select the size of the horizon to be $s' = 10$ supersteps. We then update the configuration plan at each superstep of the application and take the first action of the plan. We repeat this process at every superstep until the end of the application.

C. Reinforcement Learning Strategy

If we have perfect knowledge of the future workloads, the dynamic strategies above can be used to compute the entirety of our configuration plan. However, it suffers from forecasting errors, and even though MPC copes with forecasting errors to some extent, it does not consider that predictions at some supersteps can have higher errors than others, and thus does not allow the model to learn from these forecasting errors. To that end, we develop a reinforcement learning approach based Deep Q-Learning. We use the execution time as the reward function, and consider m look-ahead supersteps as the set of states and

$\{a_0, a_1, \ldots, a_n\}$ as the set of actions. We implemented this approach using OpenAI Gym.

V. 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. The K8s cluster specifications are shown in Table III.

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 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) [17], 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 Table I.

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 [6], 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 IV-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 IV-B2) that uses a horizon of 10 supersteps. It always outperforms the dynamic window strategy (Section IV-B1), so the plain dynamic strategy is not evaluated in our experiments.
- **Deep Q-Learning:** This is the reinforcement learning strategy (Section IV-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 4 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).

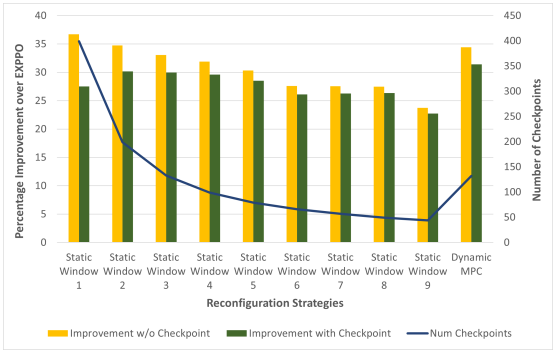


Figure 4: Results for the Static Window and Dynamic MPC strategies assuming perfect knowledge of future workloads. The left Y-axis shows the percentage improvement in execution time with and without checkpoint penalty over EXPPO (which takes approximately 65,126 seconds), and the right Y-axis shows the 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 5 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.

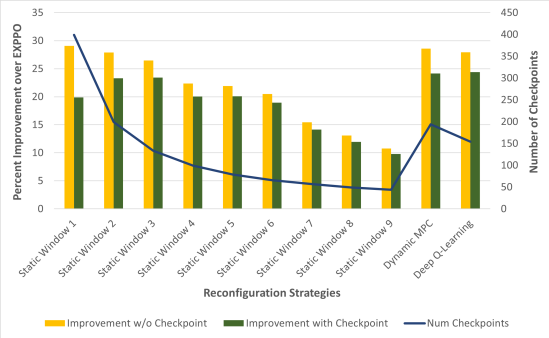


Figure 5: Results for each strategy using forecasted future workloads. The left Y-axis shows the percentage improvement in execution time with and without checkpoint penalty over EXPPO (which takes approximately 65,126 seconds), and the right Y-axis shows the number of checkpoints.

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.

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.

VI. 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.

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 [6] even in the presence of reconfiguration/checkpointing penalty.

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] K. N. Amponsah, "A Framework for Evaluating the Impact of Communication on Performance in Large-scale Distributed Urban Simulations," Ph.D. dissertation, University of Nottingham, 2021.

[5] 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.

[6] 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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.