Machine learning for high performance computing applications

by

Scott Hutchison

B.S., Texas A&M University, 2005

M.S., Air Force Institute of Technology, 2015

AN ABSTRACT OF A DISSERTATION

submitted in partial fulfillment of the
requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computer Science
Carl R. Ice College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2024

Abstract

The focus of this study was to apply state-of-the-art Machine Learning (ML) techniques to problems in the High Performance Computing (HPC) domain. The ML techniques included clustering, various types of regression, a recommendor system, and reinforcement learning using proximal policy optimization. Included are three different advancements applying these techniques.

The first application used K-means clustering and Gradient Boosted Tree Regression (GBTR) to predict estimated queue time for jobs submitted to an HPC system. This method achieved a 96% accuracy when predicting whether or not a job would start prior to a specified deadline.

The second application focused on optimizing hardware procurement for HPC systems while remaining under a fixed budget. Vendor quotes for new hardware were used with a custom Discrete Event Simulator (DES) to simulate the execution of a job workload on proposed hardware. An Extreme Gradient Boosting (XGBoost) regression model powers a recommendor system that provides a precision@50 of 92%.

The third application used Proximal Policy Optimization (PPO) with Invalid Action Masking (IAM) to train a Reinforcement Learning (RL) agent to schedule jobs on a simulated HPC system. The performance of this RL agent was compared to modern scheduling algorithms. The RL agent performed 18.44% better than the algorithmic baselines for one metric and comparably to the baselines for another.

Machine learning for high performance computing applications

by

Scott Hutchison

M.S., Air Force Institute of Technology, 2015

B.S., Texas A&M University, 2005

A DISSERTATION

submitted in partial fulfillment of the
requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computer Science
Carl R. Ice College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2024

Approved by:

Major Professor
Daniel Andresen

Copyright

Abstract

The focus of this study was to apply state-of-the-art Machine Learning (ML) techniques to problems in the High Performance Computing (HPC) domain. The ML techniques included clustering, various types of regression, a recommendor system, and reinforcement learning using proximal policy optimization. Included are three different advancements applying these techniques.

The first application used K-means clustering and Gradient Boosted Tree Regression (GBTR) to predict estimated queue time for jobs submitted to an HPC system. This method achieved a 96% accuracy when predicting whether or not a job would start prior to a specified deadline.

The second application focused on optimizing hardware procurement for HPC systems while remaining under a fixed budget. Vendor quotes for new hardware were used with a custom Discrete Event Simulator (DES) to simulate the execution of a job workload on proposed hardware. An Extreme Gradient Boosting (XGBoost) regression model powers a recommendor system that provides a precision@50 of 92%.

The third application used Proximal Policy Optimization (PPO) with Invalid Action Masking (IAM) to train a Reinforcement Learning (RL) agent to schedule jobs on a simulated HPC system. The performance of this RL agent was compared to modern scheduling algorithms. The RL agent performed 18.44% better than the algorithmic baselines for one metric and comparably to the baselines for another.

Table of Contents

List of Figures

List of Tables

Nomenclature

AI Artificial Intelligence

ANOVA Analysis of Variance

BFBP Best Fit Bin Packing

CPU Central Processing Unit

CSV Comma Separated Value

FCFS First Come First Serve

FN False Negative

FP False Positive

GBTR Gredient Boosted Tree Regression

GP-ARGO Great Plains Augmented Regional Gateway to the Open Science Grid

GPL GNU Public License

GPU Graphics Processing Unit

HPC High Performance Computing

HTC High Throughput Computing

IAM Invalid Action Masking

KSU Kansas State University

ML Machine Learning

MWF Modular Workload Format

OSG Open Science Grid

PBS Portable Batch System

PPO Proximal Policy Optimization

RL Reinforcement Learning

RMSE Root Mean Squared Error

SB3 Stable Baselines3

SJF Shortest Job First

SLURM Simple Linux Utility for Resource Management

SPEC Standard Performance Evaluation Corporation

SWF Standard Workload Format

TN True Negative

TP True Positive

XGBoost Extreme Gradient Boosting

Chapter 1

Introduction

Machine Learning (ML) has been a topic of much recent interest. In general, machine learning is a sub-field of Artificial Intelligence (AI) that enables machines to mimic human behavior and solve problems in a similar way. We have seen advancements in self driving cars with Tesla cars (and others)[3], large language models with GPT-3[4] which power Artificial Intelligence assistants like Bixby[5] or Siri[6]. In tandem with this trend, and in part because of it, there has been an increasing need for more and more computational power[7].

To meet the demand for increased computational power, High Performance Computing (HPC) systems have become larger and more powerful. HPC is the practice of combining computing power from multiple computers to solve large problems in science, engineering, or business. Numerous challenges exist when configuring these systems. Users of these systems must write and structure their code to take advantage of these distributed systems. For heavily utilized systems where jobs must be queued prior to their execution, it is often unclear to the users how long they must wait before their jobs even begin running. System administrators must determine the best equipment to procure which will meet the needs for their user base. Correctly configuring, maintaining, and running the HPC system itself often requires several full time staff members. Additional technical challenges exist, such as how to schedule jobs for execution on the HPC machines to maximize their utilization or

minimize the jobs' queue time.

As you can see, there are numerous challenges associated with procuring, configuring, using, and operating an HPC system. The goal of this work was to see if machine learning techniques could assist with finding solutions to some of these problems. Succinctly, the goals of this research are spelled out as follows:

1.1 Problem Statement

Though each of the contributions discussed in Chapters 2, 3, and 4 will have its own specific problem statement, the overarching problem statement for this work is as follows:

- How can state-of-the-art machine learning techniques be applied to problems in the HPC domain?

1.2 Contributions

The remainder of this document includes three previously peer-reviewed and published works. Chapter 2 includes an HPC job queue time prediction strategy that can predict whether or not a job will start by a specified deadline with an accuracy of 95%. Chapter 3 describes an HPC procurement optimization recommender system with a precision@50 of 92%. In Chapter 4, a reinforcement learning powered HPC scheduler is described that can learn different scheduling policies which performed 18% better than state-of-the-art scheduling algorithms for one metric and on par with scheduling algorithms for another. Finally, Chapter 5 offers some discussion on the expected scalability of these technique, discusses some potential future problems, and offers some final concluding remarks.

1.3 Survey of Machine Learning Techniques Utilized

There are many different state-of-the-art machine learning techniques used in different applications. What follows is a brief introduction to the specific techniques utilized for this research to provide familiarization prior to them being discussed in later chapters.

Unsupervised K-mean clustering

K-means clustering attempts to partition n observations into k clusters where each observation belongs to the cluster with the nearest mean. It is unsupervised, as there are no initial cluster labels for each of the observations. The centers of each of k clusters are initially randomly created, and all observations are assigned to a cluster based on the closest center. Then, the centroid (or geometric center) of all clusters is calculated, and the centroid becomes the new center. Moving the center may cause observations to shift into a new cluster, and this process repeats until convergence. Upon termination, the observations are grouped into k clusters based on the minimum distance to one of the k centers. K-means clustering is seen as part of the machine learning pipeline described in Chapter 2 which was used to predict the queue time of a job submitted to an HPC cluster.

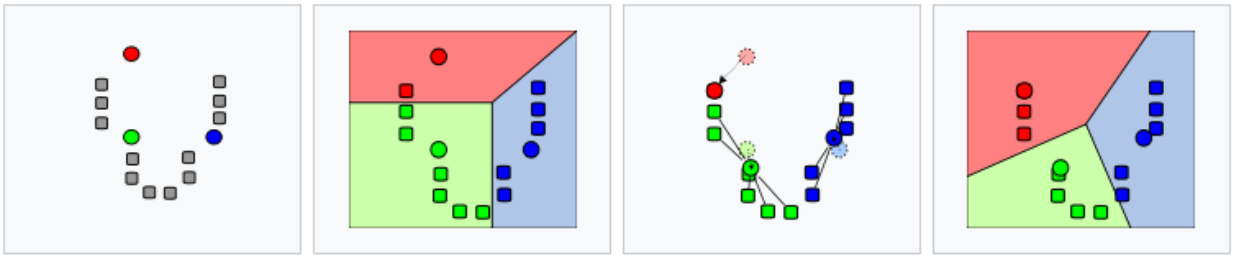

Figure 1.1: An illustration[1] of the steps of the k-means algorithm with $k = 3$. Clusters are represented by colors, observations are squares, and cluster centers are represented by circles.

Regression

Regression is seen in all three of the works presented ahead. In general, regression is a statistical process for estimating the relationships between one or more independent variables and a dependent variable. A simplified way of thinking about a regression model could be to picture the model as a "black box" function which maps a set of inputs to an output. Many different techniques have been developed to improve the accuracy of this function. During training, using various techniques, the model is refined to minimize the error of the model. Once the model has been trained, it can be used to predict a dependent variable for previously unseen independent variables. Thankfully, software libraries, like Scikit Learn[8], exist with implementations of these regression techniques, making it unnecessary to fully understand the details of the inner workings of the different regression techniques. The different regression techniques used for these research projects are described below.

Linear Regression

Linear regression, as was used in Chapter 2, assumes a linear relationship between the independent and the dependent variables. Linear regression, as shown in Figure 1.2, can accurately represent independent variable with linear relationships to dependent variables. Linear regression was used in Chapter 2 as part of the process to determine if the queue time of an HPC system was expected to increase or decrease in the future.

Gradient Boosted Tree Regression

Gradient Boosted Tree Regression (GBTR)[9] uses an ensemble of weak prediction models consisting of decision trees. Optimization during training is achieved by using a differentiable loss function. Gradient decent on this loss function allows the regression model to optimize its estimation of the dependent variable given the independent variables. GBTR was used in Chapter 2 as part of the machine learning pipeline to predict a job's queue time.

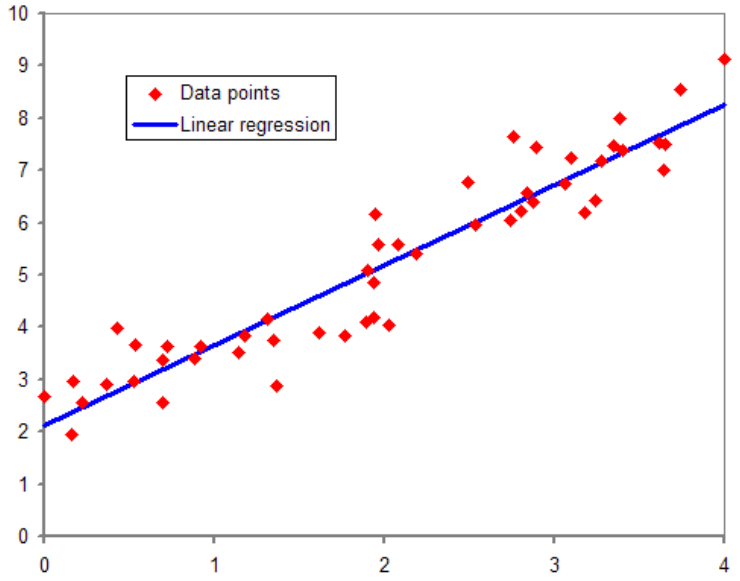

Figure 1.2: An illustration[2] of the steps a 2D linear regression line of best fit.

Extreme Gradient Boosting

Rather than using a differentiable loss function as GBTR does, Extreme Gradient Boosting (XGBoost)[10] uses Taylor approximation[11] in the loss function. The XGBoost library aims to provide a "Scalable, Portable and Distributed Gradient Boosting (GBM, GBRT, GBDT) Library". Both GBTR and XGBoost use an ensemble of decision trees to make their estimators, and XGBoost performs well in many different regression tasks. XGBoost is used in Chapter 3 to develop a regression model which can predict the average job wait time for jobs for a particular composition of proposed new HPC machines.

Reinforcement Learning

Reinforcement Learning (RL) attempts to learn a policy via rewarding the agent for good decisions during the training process and punishing it for bad ones. Figure 1.3 illustrates the general process for training a RL agent. In general, the RL agent observes the current state of the environment, S_t. The agent chooses an action, A_t, from among all the actions that are available to it in its action space. This action changes the current state of the environment

to S_{t+1}, and the agent then receives a reward, R_{t+1} based off how well the action achieved the desired policy. The agent adjusts its decision making to maximize the reward it receives by choosing the best action for a given state of the environment. Reinforcement learning was used in Chapter 4 when developing an application to schedule HPC jobs in the queue for execution on HPC resources.

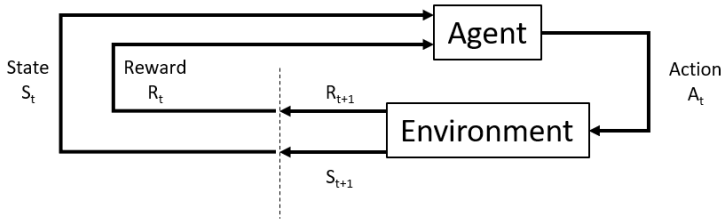

Figure 1.3: The general process for reinforcement learning

Proximal Policy Optimization with Invalid Action Masking

The specific technique utilized ahead in Chapter 4 was Proximal Policy Optimization (PPO)[12] with Invalid Action Masking (IAM)[13]. Proximal Policy is an on-policy technique, meaning that each update of the policy only uses data collected while acting according to the most recent version of the policy. The policy maps the states in the environment to actions taken when in those states. Generally, an objective function in RL returns the expected reward for an action in a given state. Rather than using gradient ascent on the objective function to optimize the policy (a computationally expensive process), PPO employs a surrogate objective function which gives a conservative estimate for how much the objective function will change as a result of a policy update. Large policy updates are penalized via clipping, resulting in quick training convergence and good performance for many tasks. Invalid Action Masking removes obvious invalid or unproductive actions from consideration during the training process. This has been shown to improve training convergence and decrease the time required for training[13].

1.3.1 Recommender Systems

An example of a recommender system might be a search engine, or a music, or movie recommendation system. If we are using a search engine to find pictures of cats, for instance, the search engine returns to us some pictures. Of all the pictures on the internet, only certain ones are pertinent to our query (i.e. the ones of cats). Only the pictures actually containing cats would be considered "hits" for our particular search. One of the metrics used to evaluate the performance of recommender systems is precision@k. Though Section 3.5 goes into greater detail, in general the user wants k recommendations returned to them. The precision@k represents the percentage of those k results that were returned by the recommender system which were "hits". For example, if precision@50 = 92%, then the recommender system would return 50 results and 92% of those (46 in this case) would be "hits". In Chapter 3, our recommender system is returning highly performant server sets under a given budget that will minimize the average job queue time.

1.3.2 Discrete Event Simulator

A custom Discrete Event Simulator (DES) was utilized for two of the works presented ahead. Additional details describing it can be found in Section 3.4.4. The DES simulates a HPC system, and keeps track of the available resources on individual machine in the simulated cluster. It also tracks the current simulation time step, and has several queues or priority queues containing the jobs submitted to the simulated HPC system. How jobs are assigned to the simulated machines is varied and can be done using algorithmic scheduling like the First Come First Server (FCFS), Shortest Job First (SJF), or the Best Fit Bin Packing (BFBP) algorithm. It can also use different techniques, such as using a scheduler powered by a Reinforcement Learning (RL) agent.

Chapter 2

High Performance Computing Queue Time Prediction using Clustering and Regression

2.1 Preface

This research was originally appeared in the proceedings of the First Workshop on Applications of Machine Learning and Artificial Intelligence in High-Performance Computing (WAML-HPC 2022)[14]. This article has been reproduced with permission from Springer Nature.

2.2 Introduction

When a job is submitted to a High Performance Computing (HPC) cluster, a scheduling application, like SLURM, PBS, LoadLeveler, etc., handles the allocation of HPC resources in the future to the job's requirements as specified by the submitter. If adequate HPC resources are currently unavailable, the job enters a queue for execution in the future, and future resources are scheduled for that job's use. While a job is queued and awaiting execution,

the job is making no forward progress toward its eventual completion. Worse still, it is often unclear to the user how long it will take until the job begins execution. The user knows the job is waiting to start, but there is often no way for the user to know if the job execution will begin in three hours, three days, or three weeks. Users with a time-critical application facing long queue delays may be willing to migrate jobs to commercial cloud infrastructure, like Amazon's AWS, Microsoft's Azure, Google's Cloud Computing, etc. Various techniques for predicting job queue times have been implemented in the past with different trade offs and accuracy. A machine learning pipeline which uses unsupervised K-Means clustering followed by Gradient Boosted Tree Regression has been used to lower error rates when used in other applications. This research investigates if this machine learning technique can also be used to a predict queue times for HPC jobs.

The goal of this research is to answer the following questions: For an HPC system with a given current utilization, can we provide an accurate queue time prediction for a job which factors in the future state of the HPC cluster? Can we predict the execution of a job prior to an assigned deadline?

2.3 Background

Improving HPC utilization, decreasing job queue time, and decreasing job turnaround time are all active areas of research at Kansas State University (KSU). These areas would typically apply and are of interest to any HPC cluster manager. Developing an accurate queue time predictor can not only help inform job scheduling, but it can also provide additional information to users about their expected job start times, and perhaps more importantly, about the length of time they can expect to wait for their results.

Kumar and Vadhiyar[15] performed a similar job queue time prediction by using k-nearest neighbors followed by support vector machines to classify jobs into time bins of various sizes with their probabilities. Though this technique showed promise, using regression allows for

a concrete prediction value for queue time, as opposed to the most likely time bin this job would fall into.

Jancauskas, et al.[16] conducted similar research on queue time prediction using Naive Bayes to return a list of probability estimates (t_i, p_i), where p_i was the probability that a job will start before t_i. They used similar features and achieved excellent accuracy, precision, and recall. An advantage of using clustering and regression over Naive Bayes is that a concrete start time prediction can be generated for the current load on the HPC system for an individual job with certain requirements. This research not only uses different machine learning techniques for prediction, but also factors in changes of the future state of the HPC system and the impact those changes have on job queue times, which Jancauskas, et al. considered outside the scope of their research.

Brown, et al.[17] used a very similar technique as this research, using k-nearest neighbors followed by gradient boosted tree regression, however they also did not factor in the future state of the HPC system.

Unsupervised K-Means[18] clustering followed by Gradient Boosted Tree Regression[9] (GBTR) has been used by various researchers to improve the accuracy of regression models. For instance, Zheng and Wu[19] used this technique to improve on short-term wind forecasting, and Liu et al.[20] used this technique to improve short-term power load forecasting. As this technique has shown promise in improving prediction accuracy in other areas of research, using clustering followed by regression could also improve the accuracy of predicting how long a job will be queued for execution on an HPC system.

2.4 Methodology

2.4.1 Data set

The HPC cluster at KSU is a Beowulf[21] HPC cluster called "Beocat". Beocat currently consists of 362 compute nodes with a total of 10980 compute cores and 5.57 Terabytes

of memory, and it uses SLURM[22] as the job scheduler. SLURM logs data from all jobs submitted to Beocat and retains 105 different features about each job. These features include job submission time, start time, end time, the number of CPUs requested by the user, the amount of memory and time requested by the user, etc. The data set used for this research consisted of all jobs submitted in 2018, which totaled approximately 730,000 jobs. Figure 2.1 shows the CPU and memory allocation over time for Beocat for 2018. The calculated CPU utilization for 2018 was roughly 60%. Jobs can remain queued due to lack of available CPUs or lack of available memory. This data set was thought to be a good representative data set with enough data to produce meaningful results.

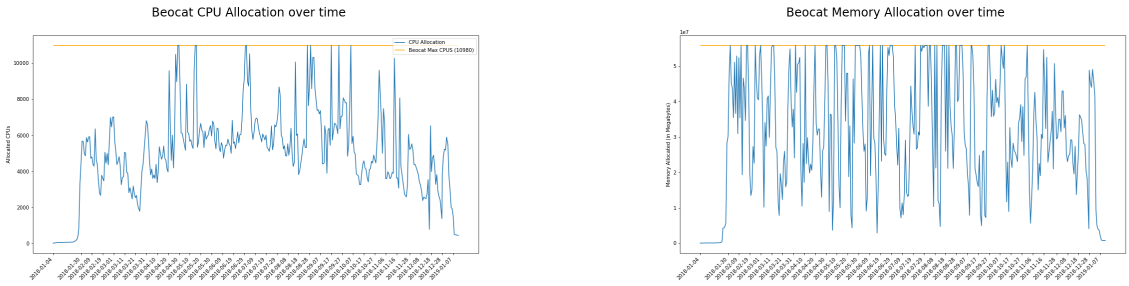

Figure 2.1: CPU and memory allocation over time for KSU HPC system for 2018

2.4.2 Feature Selection and Calculation

The queue time of a job depends primarily on two factors: the amount of resources available in the HPC and the amount of resources a job is requesting. Table 2.1 summarizes and describes the features used for this research.

To calculate `BeocatCPUsInUse`, the 2018 jobs from the log data were sorted chronologically by their start and end times. Each time a new job began, the number of CPUs in use by the cluster was increased by the number of cores allocated to that job. Each time a job ended, the number of CPUs in use by the cluster was decreased by the number of cores allocated the that job. The same strategy was employed to calculate `BeocatMemoryInUse`.

To calculate `QueueDepth`, jobs were sorted by their submit times and their start times.

Table 2.1: Features used

Category	Feature	Description
HPC features	BeocatCPUsInUse	Current allocated CPUs
	BeocatMemoryInUse	Current allocated memory
	QueueDepth	The queue depth when job was submitted
Job features	ReqCPUs	Number of requested cores for a job
	ReqMem	Amount of memory requested for a job
	ReqMinutes	Amount of minutes requested for a job
	OwnsResources	True if user has priority access to compute nodes; False otherwise
Dependent variable	QueueTimeInSec	Number of seconds from submit until start

Each time a job is submitted, the queue depth is increased by one. Each time a job starts, the queue depth is decreased by one.

The requested CPUs, requested memory, and requested time for each job were directly pulled from the log data to populate the `ReqCPUs`, `ReqMem`, and `ReqMinutes` features.

There are a number of compute nodes which are available for all Beocat users to use. Certain resources are owned by departments whose members have priority access and who can preempt running jobs. The `OwnsResources` feature was set to true if there were dedicated resources available which could run that job. If the job was submitted to only the queues common to all, the `OwnsResources` feature was set to false.

To calculate `QueueTimeInSec`, the submit time for each job was subtracted from its start time. This time delta object was converted into an integer that represented the number of seconds each job sat in the queue awaiting job execution.

2.4.3 Feature Normalization and Model Development

Min-Max scaling was used on Beocat CPU and memory allocation to return a value between 0 and 100 which represents the percentage of Beocat currently allocated.

A vector consisting of `ScaledBeocatCPUsInUse`, `ScaledBeocatMemoryInUse`, `QueueDepth`, `ReqCPUs`, `ReqMem`, and `OwnsResources` was constructed, and used to predict `QueueTimeInSec`. The log data containing roughly 730,000 jobs were randomly split into an 80% training batch

and a 20% testing batch. The training batch contained roughly 583,000 jobs, and the test batch contained roughly 146,000 jobs.

A base GBTR model was trained using 5-fold cross validation on the training data, which was then evaluated using the test data. This base model was used later to predict clusters that contained fewer than 100 elements. This model will be referred to as the GBT_{base} in various figures throughout the remainder of this report.

Since the optimal number of clusters required to group the training data was initially unclear, iterative K-Means was used to cluster the data using an increasing number of clusters from 2 to 150. The training data was fed into K-Means and n clusters were returned, where $n = 2, 3, 4, \ldots, 150$. After clustering, a GBTR model was developed using 5-fold cross validation for each each cluster containing more than 100 elements. A small cluster containing fewer than 100 elements would use the GBT_{base} model to make queue time predictions. These models were developed using the training data, and then evaluated using the test data. An unseen-before test input would first be classified by the K-means model, and then the appropriate GBTR model was used to develop a prediction of the job's queue time. The machine learning pipeline's error rate overall on the test data was used to determine an ideal number of clusters that minimized the error. The number of clusters producing a local minimum error rate was identified, and then that pipeline was selected for further evaluation. This machine learning pipeline is outlined in Figure 2.2.

During actual use, users may have individual and specific deadlines. This information is not currently solicited or collected on Beocat, so it was unavailable in the log data. An arbitrary deadline for each job was set to be the average queue time for all jobs in 2018, or 13423 seconds (HH:MM:SS = 03:43:43). A queue time prediction was made for each job, and it was assessed whether this prediction was met or exceeded the assigned deadline. Since the actual queue time was known, a confusion matrix was generated to determine the overall accuracy, precision, recall, and F1 scores for the machine learning pipeline.

The above mentioned queue time prediction represents a snapshot in time given the

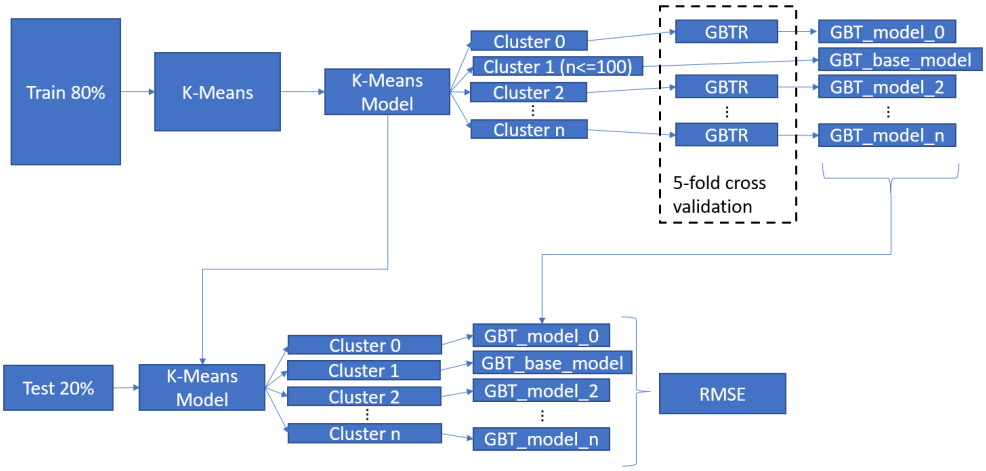

Figure 2.2: The machine learning pipeline used for this research.

overall HPC system resources allocated for a job with specific requirements. The future state of the HPC may also impact job queue time. For instance, if cluster allocation increases following a job submission when a large number of higher priority jobs are started, this queue time estimate may underestimate when a job would actually begin. Alternatively, an HPC allocation decrease following a job submission due to jobs finishing earlier than expected may cause a job to begin sooner.

Since the average queue time for Beocat in 2018 was roughly 3 hours and 45 minutes, a sliding time window of 4 hours was used to assess what impact a change in HPC CPU allocation would have on the change in queue times for HPC jobs. Figure 2.3 depicts how average Δ_{CPUs} was calculated. Average Δ_{queue_time} was calculated in the same manner. In 2018, there were 1,520 four-hour time windows containing submitted jobs. The time windows were randomly split into an 80% training batch and 20% testing batch. A linear regression model was trained using the average Δ_{CPUs} from the training data and used to predict the average Δ_{queue_time} of the test data. Again, the actual change in queue time was known from the log data, which enabled the calculation of RMSE, accuracy, precision, recall and the F1 Score for this linear regression model.

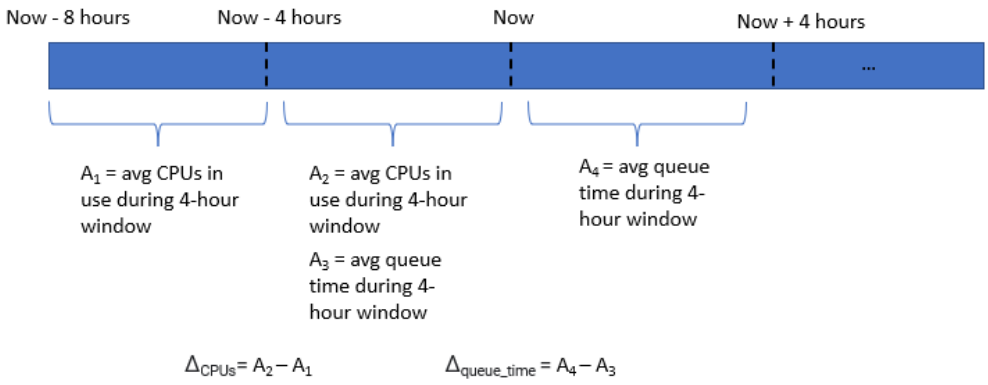

Figure 2.3: Depiction of Δ_{CPUs} calculation

2.4.4 Evaluation

Feature correlation was measured using the Pearson Correlation Coefficient[23]. This statistical measure produces a value between -1 and 1, where correlation coefficient values closer -1 or 1 indicate a stronger correlation between two features and a value closer to 0 indicates no or very little correlation.

Each regression model contained some N elements. The machine learning pipeline and each regression model was evaluated using the Root Mean Squared Error (RMSE) metric, which is calculated according to the following equation:

$$RMSE = \sqrt{\frac{\Sigma_{i=0}^{N}(\text{actual queue time}_i - \text{predicted queue time}_i)^2}{N}}$$

Additionally, the machine learning pipeline was used to compare whether or not the predicted queue time for each job exceeded the assigned deadline. A confusion matrix, along with the metrics of accuracy, precision, recall, and the F1 Score were utilized. The metrics and their descriptions are laid out in Table 2.2:

The metrics used to assess the change in queue time given the change in CPU allocation are laid out below in Table 2.3:

These metrics were calculated in the following way:

Table 2.2: Metrics for Deadline Classification

Metric	Description
True Positive (TP)	Model predicts job will start before deadline, and it does
True Negative (TN)	Model predicts job will start after deadline and it does
False Positive (FP)	Model predicts job will start before deadline, but job does not
False Negative (FN)	Model predicts job will start after deadline, but job does not

Table 2.3: Metrics for Future Queue Time Classification

Metric	Description
True Positive (TP)	Model predicts average Δ_{queue_time} will decrease 4 hours from now, and it does
True Negative (TN)	Model predicts average Δ_{queue_time} will increase 4 hours from now, and it does
False Positive (FP)	Model predicts average Δ_{queue_time} will decrease 4 hours from now, and it does not
False Negative (FN)	Model predicts average Δ_{queue_time} will increase 4 hours from now, and it does not

$$\text{Accuracy} = \frac{TP+TN}{TP+TN+FP+FN} \qquad \text{Precision} = \frac{TP}{TP+FP}$$

$$\text{Recall} = \frac{TN}{TP+FN} \qquad \text{F1 Score} = \frac{2*TP}{2*TP+FP+FN}$$

2.5 Results

PySpark is an interface for Apache Spark[24] for the Python[25] programming language. PySpark was utilized for data wrangling and analysis. PySpark's machine learning library, MLlib[26], was utilized for statistical analysis, clustering, and regression tasks. Matplotlib[27] was used to generate plots and charts.

2.5.1 Correlation of Features

Table 2.4 lays out the Pearson Correlation Coefficients for the features used. "Slightly correlated" values in Table 2.4 are displayed using orange text and the stronger "somewhat correlated" features are displayed using red text. Perhaps unsurprisingly, there is a slight correlation between the HPC CPUs in use and the HPC memory in use at any given time, as well as a slight correlation between the amount of CPUs requested by a user and the

Table 2.4: Correlation of Features

Feature	BeocatCPUsInUse	BeocatMemoryInUse	QueueDepth	ReqCPUs	ReqMem	ReqMinutes	QueueTimeInSec
BeocatCPUsInUse	1	0.195	0.008	0.008	-0.010	-0.067	-0.009
BeocatMemoryInUse	0.195	1	0.392	-0.047	-0.020	-0.036	0.131
QueueDepth	0.008	0.392	1	-0.061	-0.028	-0.091	0.326
ReqCPUs	0.007	-0.047	-0.047	1	0.119	0.057	-0.002
ReqMem	-0.010	-0.020	-0.027	0.119	1	0.036	0.003
ReqMinutes	-0.067	-0.036	-0.091	0.057	0.036	1	0.074
QueueTimeInSec	-0.009	0.131	0.326	-0.002	0.003	0.074	1

amount of memory requested by a user. The queue depth and queue time are somewhat correlated, and the queue depth and the amount of memory allocated on the HPC are somewhat correlated. This makes sense given the relatively large amount of time Beocat spends with its allocated memory near or at its maximum (See Figure 2.1).

2.5.2 K-Means Clustering and GBT Regression

The GBT_{base} model had a RMSE of 23229.92. This was compared to two naive guessing strategies of guessing zero queue time for all jobs and guessing the average queue time from 2018 for all jobs. Naively guessing zero seconds produced a RMSE of 42818.2, and naively guessing the average queue time (13423.21 seconds) produced a RMSE of 40659.8. It is clear that the base model has a lower RMSE than these two naive guessing strategies.

It was identified by iterating through the number of generated k-means clusters that 57 clusters produced a local minimum RMSE of 18119.23. As the number of clusters increased, there was not a significant improvement in accuracy, and it is thought that as the number of clusters continues to increase, the GBT_{base} model will be used for more and more clusters as the number of data points in each cluster decreases. Locating this "elbow" in the data[28] attempts to prevent overfitting and clustering beyond the point of diminishing returns. The RMSE of the machine learning pipeline as the number of clusters was varied is depicted in Figure 2.4.

Using 57 clusters produces 42 GBTR models for clusters containing more than 100 elements, and the machine learning pipeline uses the base GBTR model for the remaining 15 clusters. Each test data point was clustered, and then the appropriate GBTR model was

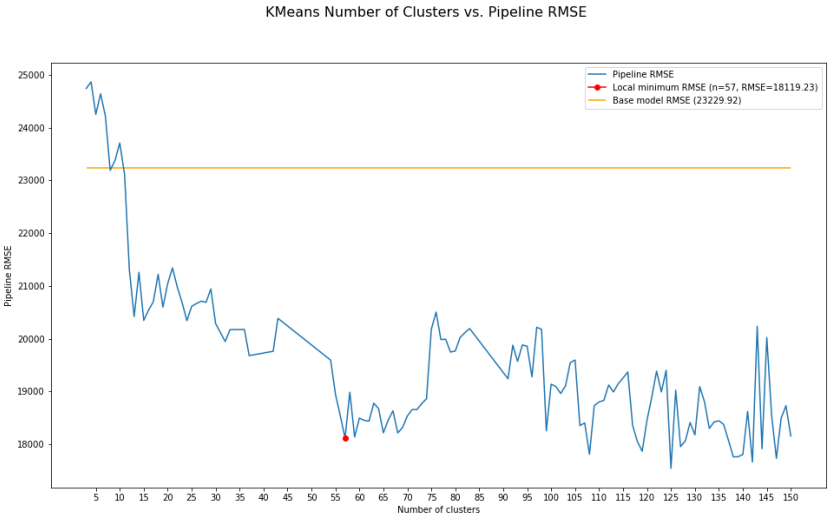

Figure 2.4: RMSE of machine learning pipeline as the number of K-Means clusters was varied.

Table 2.5: Confusion Matrix for Machine Learning Pipeline with Metrics

Total Jobs 145,658		**Actual**	
		Job runs before Avg Queue Time	Job runs after Avg Queue Time
Predicted	Job runs before Avg Queue Time	TP = 107,208	FP = 1,490
	Job runs after Avg Queue Time	FN = 3,366	TN = 33,594

Metric	**Value**
Accuracy	96.66%
Precision	98.63%
Recall	96.95%
F1 Score	97.79%

used to predict the queue time for a job. Each job's queue time prediction was compared to its actual queue time, and it was evaluated if the predicted and actual queue time exceeded the assigned deadline. The confusion matrix and evaluation metrics can be found in Table 2.5. Overall, the machine learning pipeline was excellent at predicting future queue times, and its accuracy, precision, recall, and F1 Score were all greater than 96%.

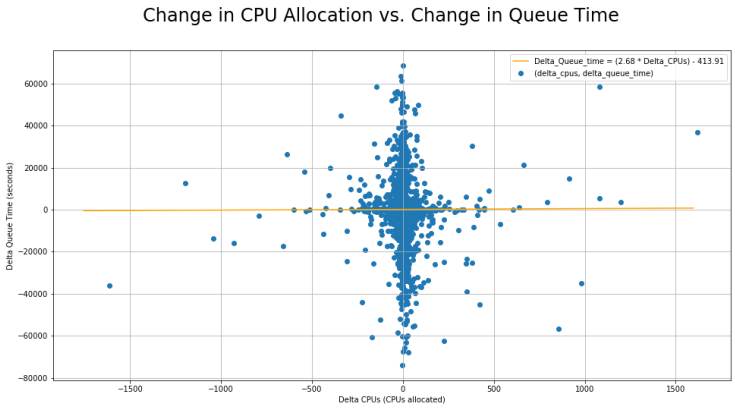

Figure 2.5: Change in average CPU allocation vs. change in average queue time

Table 2.6: Confusion Matrix for Queue Time Increase with Metrics

Total Time Windows 1,520		**Actual**	
		Future average Δ_{queue_time} decreases	Future average Δ_{queue_time} increases
Predicted	Future average Δ_{queue_time} decreases	TP = 702	FP = 26
	Future average Δ_{queue_time} increases	FN = 24	TN = 768

Metric	**Value**
Accuracy	96.71%
Precision	96.43%
Recall	96.69%
F1 Score	96.56%

2.5.3 Future HPC Queue Time Prediction

The $(\Delta_{CPUs}, \Delta_{queue\_time})$ points and the line-of-best-fit provided by the linear regression model are depicted in Figure 2.5. The model achieved a RMSE of 14691.17 seconds. The confusion matrix and evaluation metrics are found in Table 2.6. Overall, the linear regression model was excellent at predicting future queue times, and its accuracy, precision, recall, and F1 Score were all greater than 96%.

2.6 Discussion

The correlation between queue depth and the HPC memory in use for Beocat is supported by the memory in use over time depicted in Figure 2.1. This confirms the observations

made by Beocat's system administrators who have determined that more often than not, Beocat is constrained by its available memory rather than its available CPUs. This alone has informed the equipment requirements for purchases of new servers for the HPC system here at KSU. We now procure servers with larger memory to try to better accommodate our user's requirements. Doing a similar analysis might allow managers of other HPC systems to better identify hardware that can support the types of jobs their users often run.

As depicted in Table 2.5, the accuracy, precision, recall, and F1 Score were all greater than 96%. Although a somewhat arbitrary deadline was used for each user's deadline, this data could be provided by the users at submission time. This would give more meaningful information to the users of Beocat depending on how time sensitive their jobs are. Various other values for deadlines were used (1 hour, 8 hours, and 12 hours), all of which produced similar accuracy, precision, recall, and F1 scores exceeding at least 90%. It can only be concluded that the machine learning pipeline does a good job at predicting a reasonable start time for most jobs regardless of the pipeline RMSE.

Using clustering and regression as opposed to other techniques provides a concrete queue time estimate. The pipeline RMSE was roughly 5 hours of error, and the average queue time for jobs submitted to Beocat in 2018 was approximately 3 hours and 45 minutes. The HPC at KSU has comparatively low queue times for jobs, and other HPC clusters may have queue time measured in the range of days, or even weeks. An overall 5 hour error rate for the prediction for Beocat somewhat overshadows the average queue time in our case, but in other clusters, it might be more meaningful. In practice, queue times for Beocat are very left-skewed, and most of the jobs submitted to Beocat are executed after a very short period of time. Only very large jobs spend any significant amount of time in the queue waiting for resources.

It was shown that the average allocation of HPC CPUs over a 4 hour window was an effective predictor for an increase or decrease of future queue time for jobs. This information could further inform machine learning models attempting to predict queue time for jobs. For

instance, the linear regression model could be run before the queue time deadline assessment to determine if this contributes to an increase in the accuracy of the queue time prediction from the machine learning pipeline.

Finally, this queue time estimation tool could inform a decision to migrate a job to cloud resources instead of facing a long queue delay. Okanlawon, et al.[29] conducted research to better inform a user's decision to either resubmit a job with different resources or migrate that job to commercial cloud infrastructure. An accurate queue time estimation tool could offer another data point informing a user's decision.

2.7 Conclusion and Future Work

This research demonstrated that clustering and regression can also be applied to the task of queue time estimation for HPC systems. The machine learning pipeline described in this paper was more than 96% accurate at classifying whether a job would start before an assigned deadline. A simple linear regression model also achieved greater than 96% when attempting to predict if future queue times will increase or decrease. These pieces of information could prove vital to a researcher with a time critical application. It is also a meaningful metric for all HPC users, so they will be better informed about the start times of their jobs.

Additional analysis is needed to determine why certain jobs were grouped together into the clusters provided by K-Means. This research fed the cluster and job feature vector into the K-Means algorithm in search of the number of cluster producing a local minimum error rate. It is thought that additional analysis of clusters might shed light onto what is causing certain kinds of jobs to queue for longer times. Are there certain characteristics of jobs that cause them to sit in the queue longer? Are there certain characteristics or limitations of the HPC cluster itself which is contributing to longer queue times? Could additional HPC user education or better documentation mitigate queue time in some way? These remain open questions.

Our experience has been that users tend to drastically overestimate their job requirements at submission times. There is very little downside for a user who overestimates their resources at submission time. However, there is a very large downside if a job is killed before completion due to a user requesting insufficient resources at submit time. In the aggregate, however, mass overestimation of required resources leads to longer queue times for all users, which can negatively impact user experience overall. Tanash, et al.[30] have looked to machine learning to determine how actual allocated resources compared to what users have requested at submit time. Since this queue time predictor relied upon user submitted requirements for each job, adding a more accurate estimate of actual resources used would presumably improve the accuracy of a model predicting queue time.

Finally, informing the machine learning pipeline with the future queue time prediction may further improve the accuracy of the prediction made by the machine learning pipeline. It remains to be seen if first applying the future state of the HPC queue time prediction has measurable impacts on the accuracy of the clustering and regression pipeline.

Chapter 3

Optimized Hardware Configuration for High Performance Computing Systems

3.1 Preface

This research was originally presented at The Seventeenth International Conference on Advanced Engineering Computing and Applications in Sciences[31]. The results presented there have been expanded upon and further clarified and submitted to the International Journal On Advances in Systems and Measurements for consideration. The International Academy, Research, and Industry Association (IARIA) has granted royalty-free permission to any individuals or institutions to make the article available electronically, online, or in print.

3.2 Introduction

When faced with upgrading or expanding a High Performance Computing (HPC) or High Throughput Computing (HTC) system, administrators of these systems can be overwhelmed by options. It is a challenging task to get the best performance for a fixed budget. Server

capabilities (i.e., number and types of processors, amount of memory, and number and types of Graphics Processing Units (GPU) or other hardware accelerators) greatly affect their costs, and for a fixed spending ceiling, it is desirable to get the "best bang for your buck." For an HPC system, an optimal server package composition is dictated by its typical use. For instance, if many users rely upon a GPU-accelerated application or library, a higher GPU count may be desirable, even if this means fewer servers can be purchased. With many factors to consider, HPC administrators often rely upon their preferences, intuition, and experience to inform procurement decisions. This research uses historical job data from an HPC system, a discrete event simulator (DES), and a machine learning model to power a recommender system, which can help inform a hardware procurement decision. These techniques provide additional information to HPC system administrators about which set of budget-constrained hardware minimizes wait time for users' jobs, and provides quantifiable support for procurement decisions when upgrading or expanding existing HPC infrastructure. The contributions of this work can be summarized as follows:

1. A data set consisting of roughly 12,700 HPC scheduling simulations, each with a different HPC server set

2. An optimized XGBoost regression model for predicting average wait time when given a composition of servers

3. A recommender system with precision@50=92%, which can inform hardware procurement decisions

This paper is laid out as follows: Section 3.3 provides additional background on the problem and describes similar work done by others, Section 3.4 provides the methodology and some implementation details, Section 3.5 provides details of formulas for metric calculations, Section 3.6 provides the results of the experiments, and Section 3.7 provides additional details on how the recommendations of the system were evaluated across a wider time frame. Section

3.8 evaluates the performance of a model trained using all available data, and section 3.9 provides our final conclusions.

3.3 Background and Related Works

The Open Science Grid (OSG)[32][33] is a worldwide collaboration that offers distributed computing for scientific research. In the central United States, one of the organizations contributing resources to the OSG is the Great Plains Augmented Regional Gateway to the Open Science Grid (GP-ARGO)[34]. In part, GP-ARGO receives funding through governmental grants. These grants are often used to procure new equipment to expand or improve the capabilities of GP-ARGO's participating organizations. Consequentially, there is a fixed budget ceiling for HPC equipment procurement, and the administrator's goal is to purchase new equipment that will maximize computational performance for our typical applications while ensuring costs remain under the fixed grant budget. The research question for this work is as follows: for a planned HPC expansion, can experimental simulation provide an optimal set of hardware under a given budget that will minimize job wait time?

The challenge of optimal hardware procurement is not exclusive to our organization. Similar work was done by Evans et al.[35]. They collected benchmarks for various software applications on different hardware to optimize the ratio of Central Processing Unit (CPU) and GPU architectures for HPC jobs. Their work is similar to ours, but we took a different approach by using a scheduling simulator to evaluate the performance of a set of jobs that were actually submitted to an HPC system. We are solving a very similar problem as Evans et al., but using a different approach to arrive at an optimal hardware configuration.

Other researchers have attempted to optimize for a particular application, such as the work Kutzner et al.[36] did to improve the utilization of GPU nodes when using GROMACs. Although these techniques are not without their merits for HPC systems that run a large number of homogeneous applications, users of the GP-ARGO HPC systems run a wide

variety of jobs and applications. A more broad scheduler-based optimization was more appropriate for our application.

Various public HPC workloads exist[37], and have been used by HPC researchers in the past. However, as we are attempting to identify and evaluate new hardware for a specific HPC system, log data from that HPC system was utilized as the workload for this research.

Different scheduling applications like SLURM, HTCondor, or PBS, operate on HPC systems and perform the function of assigning HPC resources to jobs. This job-to-machine-assignment task is as an extension of the online bin packing problem[38]. For the bin packing problem, the goal is to pack a sequence of items with sizes between 0 and 1 into as few bins of size 1 as possible. Each job specifies the resources requested (the object sizes), and each HPC machine has a certain amount of available resources (the bins with their respective sizes). The scheduler is given the task to meet job requirements by assigning them to HPC nodes (pack the objects into the available bins) as efficiently as possible. This is an online problem as new jobs are submitted over time to the scheduler. The best fit bin packing (BFBP) algorithm has been shown by Dosa and Sgall[39] to use at most $\lfloor 1.7OPT \rfloor$ bins, ensuring this algorithm will provide a reasonably close to optimal average wait time when it is used as an HPC job scheduling algorithm. Since scheduling algorithms vary between applications, most being highly customizable, and others being proprietary, a discrete event simulator utilizing the BFBP algorithm served as a stand-in for our scheduling application in an attempt to make it more universally applicable. The BFBP scheduling algorithm is described in Figure 3.1.

Although various HPC simulators have been used for similar research, such as SimGrid[40], GridSim[41], or Alea[42], this experiment needed a simple discrete event simulator using the BFBP scheduler. The simulators mentioned above were either deemed overly complex for our purposes, or they failed to allow for the three limiting resources (memory, CPUs, and GPUs) we were interested in investigating. An HPC scheduler simulator was also considered, such as the Slurm simulator developed at SUNY University in Buffalo[43]. Although this

Algorithm 1 Best Fit Bin Packing Scheduling

```
1: while The simulation is incomplete do
2:     if Some job in the queue can be executed on some machine then
3:         Find the (job, machine) pairing which results in the fewest remaining resources for some machine. Begin executing that job on that machine.
4:     else
5:         Advance simulation time until a new job is submitted or a running job ends, whichever is sooner.
6:         Queue submitted jobs and stop ending jobs.
7:     end if
8: end while
```

Figure 3.1: Pseudocode for the best fit bin packing algorithm

option was investigated further, scaling a job's actual duration from the log data to the new machine once it is assigned to a machine was challenging. As such, a custom discrete event simulator was developed and utilized for this research. The simulator allows for three resource constraints in each machine: memory, CPUs, and GPUs. It is fairly lightweight, fast, and easy to understand.

A significant consideration when evaluating new server hardware is the performance increase newer technology or architectures can provide. Using log data, we know how long a job took on a machine with known hardware. Since the specifications for the new hardware under consideration are also known, the actual duration of the jobs from the historic log data was scaled using base performance of the processor as reported by SPEC CPU2017 benchmark, second quarter, 2023[44].

Knowing how a particular job performed on one set of hardware and estimating how it will perform on some other hypothetical set of hardware is challenging. Sharkawi et al.[45] successfully used a similar SPEC benchmark to estimate the performance projections of HPC applications. Other researchers, like Wang et al.[46] have pointed out that these benchmarks fail to account for all the variables affecting job resource utilization and should be avoided. Although CPU performance is not the only factor by which we could have scaled

job duration, and perhaps it is not the best factor by which to scale, it worked well for our purposes. The discrete event simulator was implemented such that the scaling factor could be easily changed if other researchers should find a different factor more relevant to their situation.

Various metrics are typically used when evaluating the performance of HPC scheduling algorithms. Some of these are average wait time, HPC utilization, average turnaround time, makespan, throughput, etc. Which metric is used depends on the application and function of the HPC system, and different organizations may value one metric over another. The metric used for this research was average wait time, or the average number of seconds each job spent waiting in the job queue for execution on HPC resources. We presume that the same techniques could be applied by other researchers using a different metric, should they prefer a different one.

This research relied upon a regression model where: given the total CPUs, total memory, and the total GPUs for a composition of servers, the regression model will predict the average jobs wait time for the representative set of jobs. Various regression techniques were tried, but Extreme Gradient Boosting (XGBoost)[10] was the most effective of those tried. XGBoost is a scalable, distributed, gradient-boosted, decision tree machine learning library. It relies upon supervised machine learning, decision trees, ensemble learning, and gradient boosting. Similar to a random forest, multiple decision trees are created for the regression task, and these trees each make predictions of the average wait time given the three inputs (total server package CPUs, memory, and GPUs). The results from the multiple trees are combined via a weighted sum, and they are "boosted" by generatively adding new decision trees. The error of the objective function is minimized by gradient descent during the training process, resulting in quick convergence and accurate prediction results.

Recommender systems power a variety of applications like search engines and music recommendation systems. First, the "hits" for the system must be defined. Hits are the elements from the data set that are relevant to the user's search. Next, the user specifies the

number of recommendations, k, that they would like to receive. If the recommender system is precise, a large portion of the k items returned will be hits.

3.4 Methodology

The general plan for optimizing a hardware package for our fixed budget can be summarized as follows:

1. Receive vendor quotes with potential server options.
2. Generate potential server combinations to purchase under the specified budget which meet our procurement requirements.
3. Identify a typical set of jobs representing the workloads typically submitted to our HPC system.
4. Conduct simulations using a subset of the server packages to schedule the representative job set and compute metrics to determine their performances.
5. Use machine learning to train and refine a regression model that can predict the performance of un-simulated server combinations.
6. Develop a recommender system using the machine learning model and quantitatively evaluate its performance
7. Subjectively evaluate the recommended server packages and make a more informed procurement decision.

This pipeline is illustrated by Figure 3.2. First, we generate all possible combinations of servers we can purchase under our budget. Next, we uniformly sample 10% of these by selecting every tenth server combination and we use the DES to simulate the execution of a chosen set of jobs. We then use XGBoost to develop and train a regression model which will

map a sever package's total CPUs, memory, and GPUs to the predicted average wait time that these jobs will experience. We use the regression model to predict the average wait time of the sampled server combinations, sort them by the predicted wait time, and return the top k recommended server sets to the user. These recommendations can be quantitatively evaluated, as the actual average wait time has been simulated. Next, we can use the same regression model and recommender system to make predictions on the entire set of servers, and we can summarize them and subjectively evaluate them, as 90% of the have not be simulated and their actual average job wait time is unknown.

Finally, the recommendations of the system were simulated using workloads from 24 days across a calendar year to determine it the recommended server sets continued to be effective when faced with the varied workloads the HPC system experienced throughout the year.

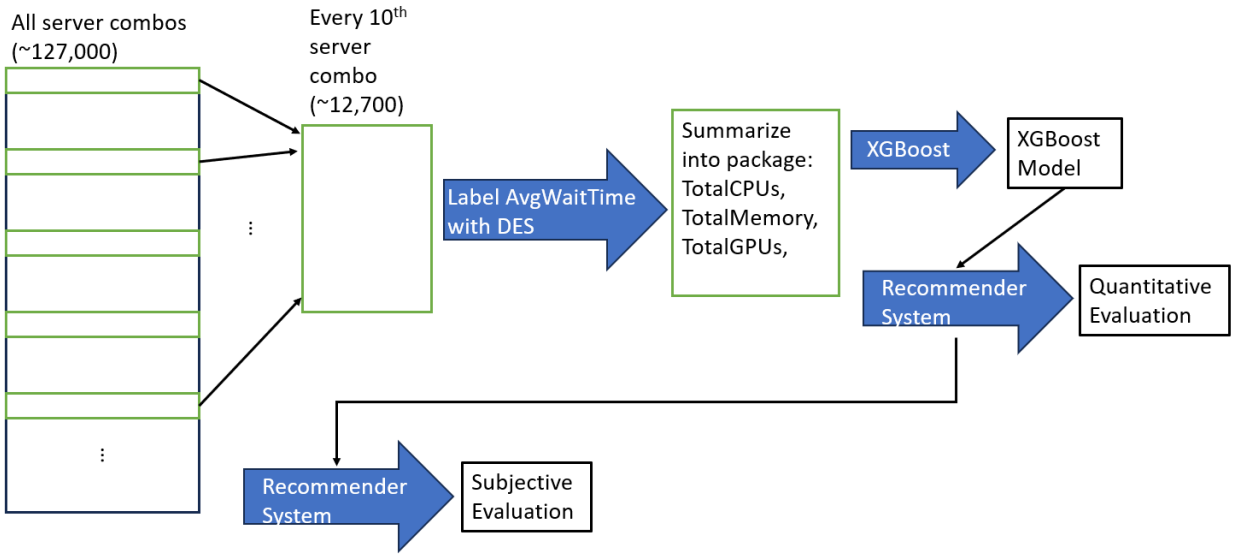

Figure 3.2: A pictorial representation of the methodology for this research.

3.4.1 Generate Server Options

To begin, we received several vendor quotes specifying the costs and capabilities of 21 potential servers to purchase. When considering upgrade options, we typically separate servers into one of three categories: compute nodes, big memory nodes, or GPU nodes. A compute

Table 3.1: Server capabilities and costs under investigation

Node type	*Distinct nodes considered*	*Memory range per node*	*CPUs range per node*	*GPUs per node*	*Cost range per node*
Compute	4	256-512 Gb	24-64 cores	0 GPUs	$6k-$10k
Big memory	2	1024 Gb	24-64 cores	0 GPUs	$11k-$13k
GPU	15	256-1024 Gb	24-64 cores	1-8 GPUs	$14k-$100k

node typically has a large number of processor cores, a moderate amount of memory, and no GPU. A big memory node will have a large amount of memory with a moderate amount of CPU cores and no GPU. A GPU node is any node which has a GPU. Table 3.1 lays out the options we received from several different vendors. The procurement budget was fixed at $1 million, and all possible server combinations were generated in the following way:

- Separate servers into three categories: compute nodes, big memory nodes, and GPU nodes.
- Choose all combinations of one node from each category.
- Determine all quantities of the three node types under a given budget such that there is at least one GPU node and there is not enough funding remaining to purchase another node.

In our selected job set, many jobs requested GPUs as a resource. These jobs would automatically fail if at least one GPU node were not included in a potential server package. Roughly 127,000 different server combinations met these requirements. Table 3.2 provides an illustrative example of how the server combinations were generated. Many server options and packages were omitted from the table for the sake of brevity.

3.4.2 Identify a Representative Set of Jobs

One typical days' worth of submitted jobs (roughly 16,000 jobs) was subjectively pulled from the log data of the local HPC system. As with most HPC systems, jobs were submitted

Table 3.2: Generated Server Combinations

ComputeNode1, $6,960 ea.	*BigMemNode1, $11,112 ea.*	*GPUNode1, $14,730 ea.*	...	*Package Cost*	*Funds Remaining*
141	0	1	...	$996,090	$3,910
139	1	1	...	$993,282	$6,718
138	2	1	...	$997,434	$2,566
⋮	⋮	⋮	⋱	⋮	⋮
0	1	67	...	$998,022	$1,978

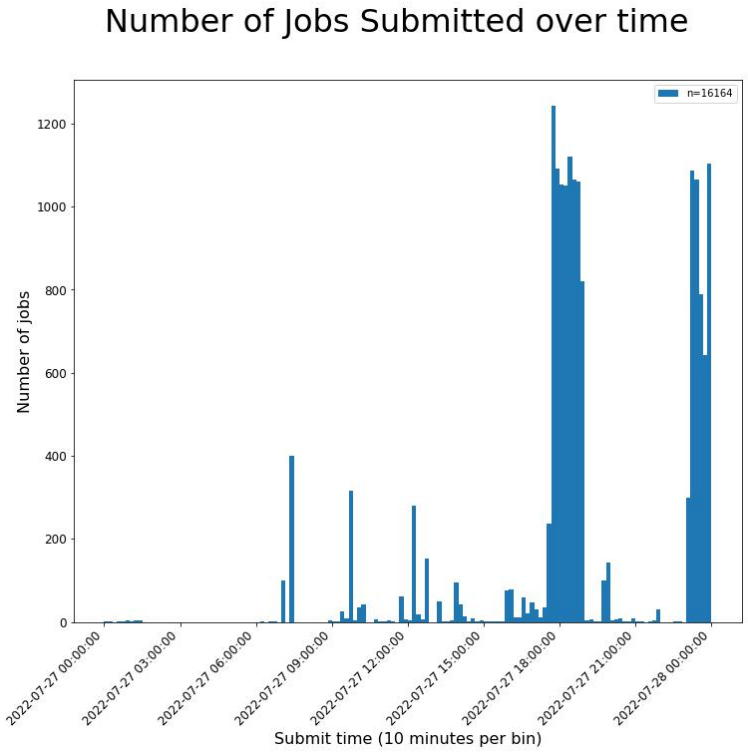

Figure 3.3: The number of jobs submitted over time for the selected day

in a bursty manner, and variety of resources were requested. Figure 3.3 and Table 3.3 display some descriptive statistics and information about the jobs used for this portion of this research.

3.4.3 Job Duration Scaling

The submitted jobs were scaled using the base performance of the processor on the SPEC CPU2017 benchmark suite. The requested duration was not modified, but the actual duration of each job was calculated using the following formula:

Table 3.3: Descriptive statistics for the pool of selected jobs

	Requested Mem (in Gb)	*Requested CPUs*	*Requested GPUs*	*Requested Duration (in hours)*	*Actual Duration (in hours)*
Mean	5.12	4.75	0.002	2.82	2.27
Std Dev.	16.73	3.33	0.055	1.02	13.67
Min	1	1	0	0	0
Max	800	64	4	11.20	11.20

$$\text{New duration} = \frac{\text{logged duration} * \text{logged processor performance}}{\text{new processor performance}}$$

3.4.4 Discrete Event Simulator

Since there are many different applications for scheduling jobs on HPC systems, the discrete event simulator using the BFBP scheduling algorithm acted as a generic substitute for the scheduling application for our HPC system. What was needed was a method for determining the average job wait time for selected jobs on specified hardware. Other simulators could have been used, but for this research, a discrete event simulator was implemented in Python that provides the following functionality:

- A global clock to keep track of simulation time.
- Several queues, priority queues, or lists to track jobs as they progress through the execution process: future jobs, queued jobs, running jobs, completed jobs, and unrunnable jobs.
- Jobs and machines are specified using comma separated value (csv) files, which is loaded prior to the simulation.
- Machines have three limiting resources: available memory, CPUs, and GPUs.
- Jobs are specified with the following attributes: submit time, actual duration, and requested duration, memory, CPUs, and GPUs. Jobs track their start time and end time as the simulation progresses to allow for metric calculation.

- Job end time is set when the job starts running as the job start time plus the job actual duration.
- When a job starts running on a machine, that machine's available resources are decremented by the resources requested by the job. Conversely, when a job completes, the machine executing it has its available resources increased by the amount requested by the ending job.
- Jobs with a submit time greater than the current global clock reside in the future jobs priority queue.
- Jobs with a submit time less than or equal to the current global clock, but not yet assigned to a machine, reside in the job queue.
- Jobs that have begun their execution and have an ending time less than the current global clock, reside in the running jobs priority queue.
- Jobs with an ending time less than or equal to the current global clock reside in the completed jobs list.
- If no node in the cluster has adequate resources to run a particular job, that job is moved to the unrunnable jobs list.
- In the event that no queued jobs can run on available resources, the simulation time "fast forwards" to the next event: either job submission or job ending.
- Jobs in the job queue are run as soon as there are available resources and are chosen using the best fit bin packing scheduling algorithm described in Algorithm 3.1.
- Actual job duration from logged job data can be scaled to allow for hardware improvement with newer hardware.

3.4.5 Machine Learning

Although each simulation completed fairly quickly, requiring no more than 30 minutes each, this particular combination of server quotes yielded roughly 127,000 combinations that need to be evaluated. To reduce the computational requirement, every tenth line from the file with the server combinations was sampled, and roughly 12,700 simulations for these server packages were completed in parallel using HPC resources. By sampling from the generated server packages uniformly, various quantities of each server under consideration were included in the simulated data. Each server package was summarized into the package total memory, total CPUs, and total GPUs, by summing the resources of every machine comprising the package. The average wait time for the simulation served as the label for each package. Using five fold cross validation, an XGBoost regression model was trained using training data. The regression model was evaluated using root mean squared error (RMSE) on the test data. An accurate regression model enabled the prediction of the average wait time for unsimulated server combinations and saved countless hours of additional simulation.

3.4.6 Recommender System

In our case, a hit was defined as a server combination with an average wait time in the lowest 5% of simulated combinations (or 632 hits out of the $\sim$12,700 simulated server combinations). The value of k was varied to evaluate the performance of the recommender system. Then, once confidence was gained that our recommender system was functioning properly, it was used to recommend systems from the entire server combination pool of 127,000 server combinations. The recommendations were summarized and evaluated subjectively before arriving at a final procurement decision.

3.4.7 Simplifying Assumptions

The current nodes comprising the HPC system were not added to the set of nodes simulating the selected jobs. The benefit current nodes would provide to the new servers under investigation would be common to all.

Any additional equipment required to install and operate the new servers (e.g., networking hardware, additional cooling equipment, server racks, power infrastructure, etc.) were not deducted from the total procurement budget. It was thought that these costs would be a relatively fixed regardless of the server package chosen. The same analysis described in this research could be done by reducing the total budget by the cost of additional hardware and then completing the analysis with a reduced budget.

3.5 Evaluation

Pearson's Correlation Coefficient[23] determined the extent of the correlation between the total memory, CPUs, and GPUs of a package and the average wait time. This coefficient provides a value between -1 and 1, where values closer to -1 or 1 indicate that the feature and the label are more strongly correlated. A coefficient of 0 indicates no correlation.

Wait time was calculated by analyzing the completed jobs output from each simulation. The wait time for each job was the number of seconds from the time the job was submitted until it began. For N jobs, the average wait time was calculated as follows:

$$\text{AvgWaitTime} = \frac{\Sigma_{i=0}^{N}(\text{Start Time}_i - \text{Submit Time}_i)}{N}$$

Root Mean Squared Error was utilized for regression model evaluation calculated accord-

ing to the following formula:

$$\text{RMSE} = \sqrt{\frac{\Sigma_{i=0}^{N}(\text{actual wait time}_i - \text{predicted wait time}_i)^2}{N}}$$

The performance of the final recommender system was evaluated using precision@k, recall@k, and F1@k. In general, precision@k is the proportion of recommended items in the top-k set that are relevant, and recall@k is the proportion of relevant items found in the top-k recommendations. F1@k is the harmonic mean of precision@k and recall@k, which simplifies them into a single metric. They were calculated according to the following formulas:

$$\text{Precision@k} = \frac{(\#\text{ of recommended items @k that are relevant})}{(\#\text{ of recommended items @k})}$$
$$\text{Recall@k} = \frac{(\#\text{ of recommended items @k that are relevant})}{(\text{total }\#\text{ of relevant items})}$$
$$\text{F1@k} = \frac{(2 * \text{precision@k} * \text{recall@k})}{(\text{precision@k} + \text{recall@k})}$$

3.6 Results

The correlation of features, the performance of the regression model and the recommender system, and some analysis about the recommended server compositions are described below.

3.6.1 Feature Correlation

The correlation between the features and the labels is shown in Table 3.4. For this set of jobs, the total CPUs in a server package were most strongly correlated to the average wait time. For the chosen jobs, the more CPUs a package had, the lower its average wait time.

Since we are constrained by our available budget of $1 million, choosing to buy one type of node over another is a zero-sum game. The more GPU nodes we purchase, and the more GPUs there are per node, the fewer compute nodes or big memory nodes we are able to afford. This is indicated by the positive correlation between GPUs and the average wait

time.

Table 3.4: Pearson Correlation Coefficients

	TotalMem	*TotalCPUs*	*TotalGPUs*	*AvgWaitTime*
TotalMem	1.00	0.14	-0.54	-0.23
TotalCPUs	0.14	1.00	-0.42	-0.70
TotalGPUs	-0.54	-0.42	1.00	0.44
AvgWaitTime	-0.23	-0.70	0.44	1.00

3.6.2 Regression Model

The XGBoost regression model had a RMSE = 150.13 seconds, indicating that the total memory, CPUs, and GPU features made excellent predictors for the average wait time for these jobs when simulated with the discrete event simulator. The predicted vs. simulated wait time is shown in Figure 3.4. If the regression model were perfect, all these points would lie upon the $y = x$ line, and it is clear that this model does a good job at predicting the average wait time for a given composition of servers.

3.6.3 Recommender System

The regression model was used to predict the 12,700 labeled simulations, and their precision@k, recall@k, and F1@k for various values of k are displayed in Table 3.5. The goal was to reduce the number of possibilities from roughly 127,000 different possible combinations of servers down to a reasonable number which could be evaluated by an HPC system administrator and have a large percentage of the recommended server combinations be hits (among the best 5% of server combinations with the lowest average wait times). Although precision@10 was 100%, it is thought that seeing more server package options would allow system administrators a wider variety from which to choose. A system administrator could easily and quickly review up to 50 recommendations ($k = 50$), and more than 46 out of 50 of these recommendations returned by this system (92%) would be top performing server combinations, which is excellent. Recall@k when k is less than the number of total hits

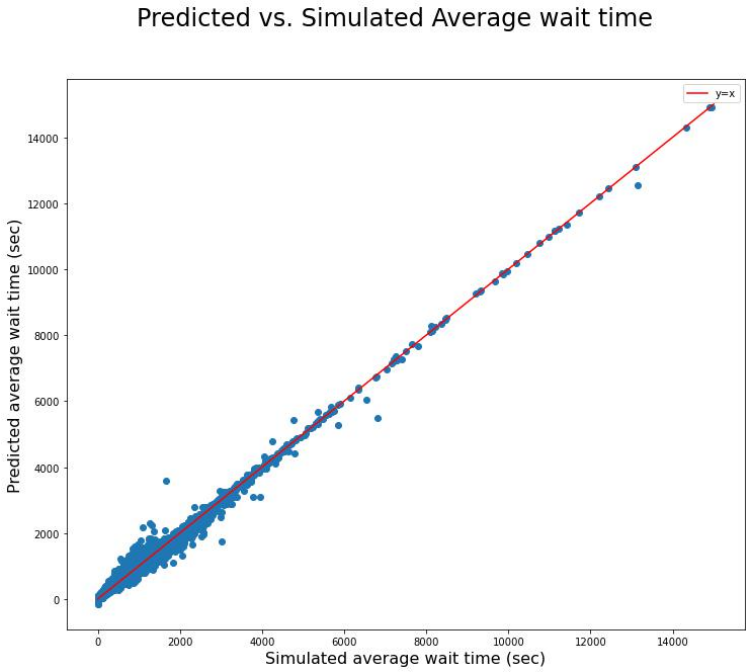

Figure 3.4: The predicted vs. simulated wait times showing the accuracy of our regression model.

(632 hits total) is unfairly penalized, but the recall@k above 632 is also excellent. When $k = 1,000$, the recall@1000 = 91%, meaning the recommender system successfully retrieved 91% of the top 5% performing server packages when returning less than 1% of the 127,000 different options.

Table 3.5: Precision@k and Recall@k for Test Data

k value	*Precision@k*	*Recall@k*	*F1@k*
10	1.00	0.02	0.03
50	0.92	0.07	0.13
100	0.81	0.13	0.22
500	0.74	0.59	0.66
632	0.72	0.72	0.72
1000	0.58	0.91	0.71

3.6.4 Recommended Compositions

Beyond looking at the individual server compositions recommended, we wanted to draw some conclusion about the types and quantity of nodes that the recommender system returned. The sum of the server quantities for the top 50 recommendations can be found in Table 3.6. Compute nodes with the larger number of cores were vastly preferred, and the recommender system did not recommend spending additional funds on more memory for the compute nodes. Additionally, the recommender system preferred the cheaper big memory node with fewer cores. Finally, for our typical workload, the recommender system did not recommended purchasing a large number of GPUs per GPU node, instead recommending servers with 2 GPUs per server most often. As shown in Figure 3.5, the recommender system suggests spending on average 58% of our total budget on compute nodes, 8% on big memory nodes, and 34% on GPU nodes.

Table 3.6: Recommendations Drawn from Model Predicted Results

Node Type	*Node Description*	*Sum of Servers Across Top 50*
Compute Nodes	Low Cost CPU w/ 256Gb	232
	Low Cost CPU w/ 512Gb	0
	High Cost CPU w/ 256Gb	3,467
	High Cost CPU w/ 512Gb	0
Big Memory Nodes	Low Cost CPU w/ 1024Gb	232
	High Cost CPU w/ 1024Gb	111
GPU Nodes	2 GPUs in one server	732
	4 GPUs in one server	267
	8 GPUs in one server	0

A boxplot showing the simulated average job wait time of the top 5% of recommended server sets (or k=638) is shown in Figure 3.6. It is clear that the recommender system was able to retrieve and recommend server sets which performed well on the representative set of jobs.

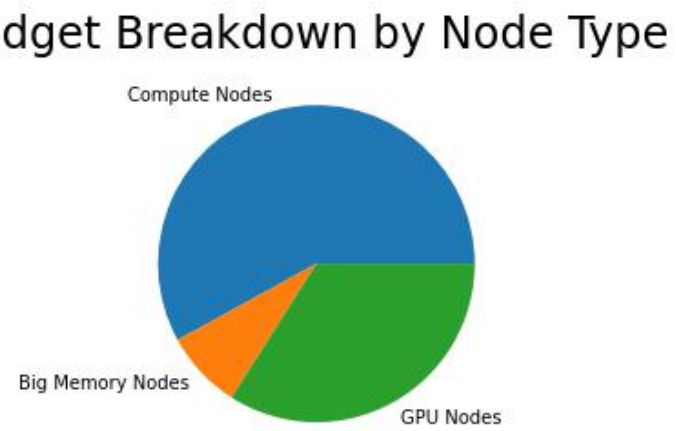

Figure 3.5: The recommended budget breakdown by node type.

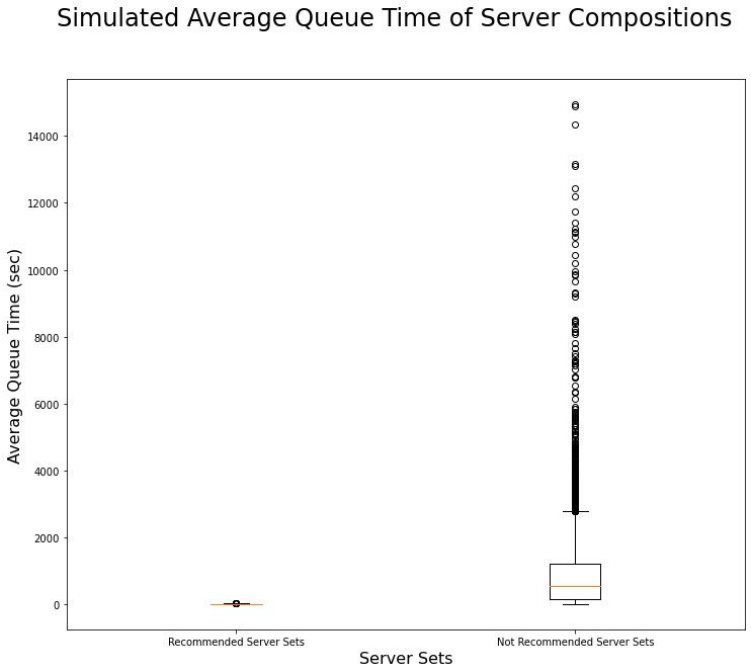

Figure 3.6: The average job wait time for the top 5% recommended server sets vs. the bottom 95%.

3.7 Evaluation of the Generalization of the Approach

The previously described regression model was developed using the results when using a single, representative workload of one days' worth of jobs from the local HPC system. For this technique to be viable, it must be demonstrated that the recommended server packages would perform well not only for the representative day, but also for many different days of HPC activity. To investigate this further, the following methodology was used:

- Subjectively pull log data an additional 24 days across the year (2 days per month)
- Use the discrete event simulator to simulate the execution of the workloads for the same subset of 12,700 server packages
- Identify the top 10% of server compositions with the lowest average wait time for each day
- Evaluate the performance of the initial recommended server sets using the additional labeled data

This computation was done in parallel using HPC resources and involved over 150,000 CPU hours.

3.7.1 Generalized Results

To begin, 24 different days of HPC log data from throughout the year were chosen subjectively (2 days per month). These days were scheduled using the DES using the roughly 12,700 server combinations which were originally used to train the regression model. See Figure 3.7 for the average wait times for each of the days simulated. Since the job characteristics for each day were different, this caused a re-ordering of the "hits" for each day. For instance, if a day had many GPU jobs, server combinations with more GPUs would have lower average job wait times. Each day's hits were defined as the server set whose average job wait time

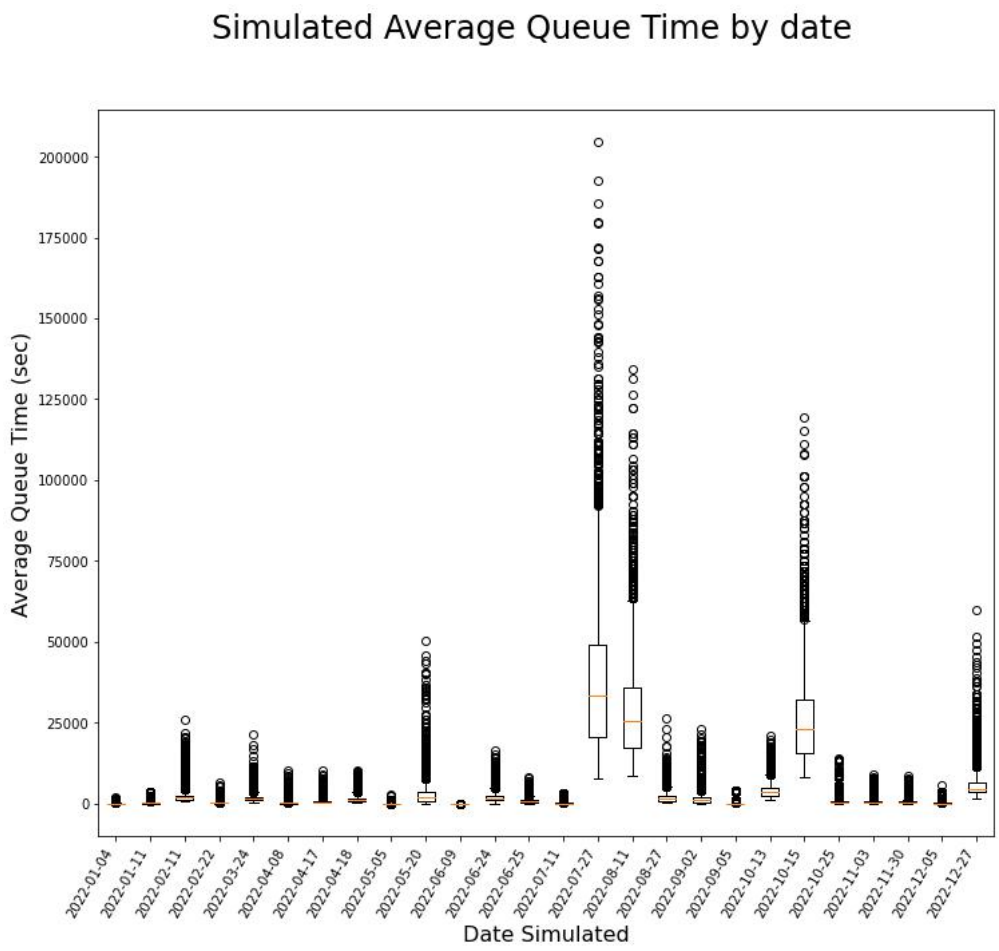

Figure 3.7: Boxplots of the average wait time for each of the days simulated throughout the year.

was in the lowest 10% for that day. By counting the number of days across the year for which that server combination was in the top 10% of performant server combinations, we were able to determine which server combinations were an overall "hit" for the recommender system. A hit threshold of $num\_hits > 6$ was found to produce good results, meaning that for 7 or more days of the 25 labeled days throughout the year, the server combination was in the top 10% of performant server packages. The results can be found in Table 3.7.

If $k = 50$, the recommender system achieves a precision@50=88%, which is slightly lower than the precision@50=92% when the day was evaluated on the same day on which it was

Table 3.7: Precision@k and Recall@k when Hit Threshold >6

Hit Threshold	k	precision@k	recall@k	F1@k
>6	10	1.00	0.005	0.01
>6	50	0.88	0.02	0.04
>6	100	0.83	0.04	0.08
>6	500	0.93	0.23	0.36
>6	1000	0.89	0.43	0.58
>6	2046	0.75	0.75	0.75
>6	5000	0.41	1.00	0.58
>6	10000	0.2	1.00	0.34

trained. Again, 50 recommendations is thought the be an easily human parsable amount which can be compared and evaluated by system administrators for purchase. In other words, given the top 50 recommendations returned to the user, 88% of them would be in the top 10% of performant server sets for 7 out of the 15 days throughout the year which were evaluated.

Increasing the hit threshold reduces the number of total hits that the recommender system can find, and consequentially lowers the precision@k. For instance, the threshold mentioned in Table 3.7 required a server set to be among the top 10% for seven or more days out of the 25 days simulated. In this case, there were 2,046 overall "hits" out of the $\tilde{1}$2,700 total server sets whose performance was measured. If the hit threshold is raised to 20, meaning 21 or more days found these server combinations in the top 10% of performing server sets, there are only 70 total hits for the recommender system to find. Figure 3.8 shows how precision@k degrades when the hit threshold is raised. When the hit threshold is raised to 20, the recall@500 is 67%, meaning that the top 500 results returned by recommender system contained 67% of the 70 hits that were found. Table 3.8 shows the results at this hit threshold. Though these results are less promising, there were relatively few server sets that performed well for this many days throughout they year. Using the recommender system trained on days' worth of representative jobs saved approximately 150,000 hours worth of computation time conducting the simulations on the various jobs submitted throughout the year.

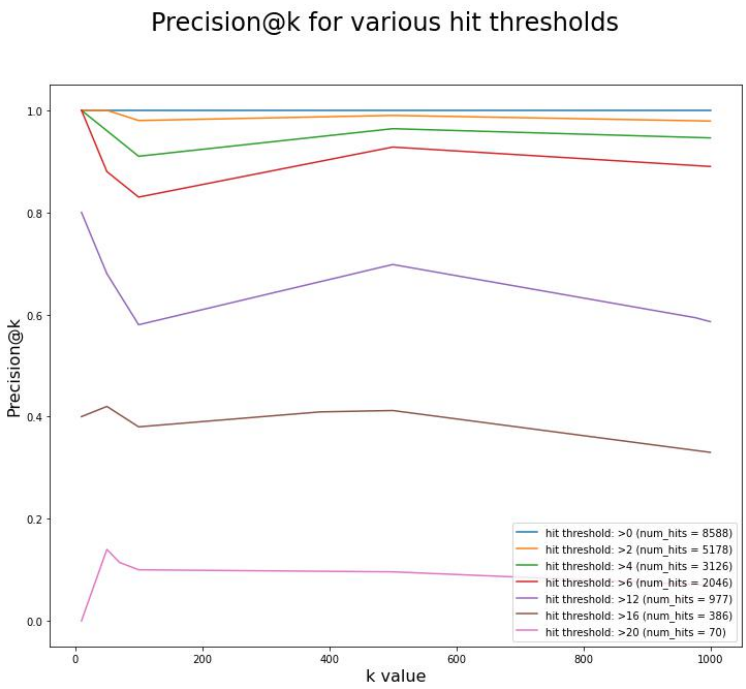

Figure 3.8: The precision@k as the hit threshold is varied.

Table 3.8: Precision@k and Recall@k when Hit Threshold >20

Hit Threshold	k	precision@k	recall@k	F1@k
>20	10	0.00	0.00	0.00
>20	50	0.14	0.10	0.12
>20	70	0.11	0.11	0.11
>20	100	0.10	0.14	0.12
>20	500	0.10	0.67	0.17
>20	1000	0.07	1.00	0.13
>20	5000	0.01	1.00	0.03
>20	10000	0.01	1.00	0.01

3.7.2 Time Savings for this Technique

Each simulation took around 30 minutes to complete, but they were conducted in parallel using HPC resources. The roughly 12,700 server compositions simulated to train the regression model took over 6,000 compute hours to complete. Each of the 24 additional days

took another 6,000+ hours, for a total of over 150,000 compute hours required to validate the recommendations across a representative sample throughout the year. An exhaustive search of all 127,000 server combinations for a single representative days' worth of jobs would have taken over 60,000 compute hours, and would have yielded a definitive answer on which server combination would have performed best on a single representative days' worth of jobs. Validating this across 24 days across a calendar year would have required over 1.5 million compute hours on HPC resources. The recommender system built using regression from a subset of the possible servers required a 99.96% decrease in the time required for computation while still achieving a precision@50 of 92%. This model achieved a precision@50 of 88% when using the threshold that for 6 or greater days, the server compositions had the lowest 10% average job wait time for each day. The additional validation step of computing 24 days across the year could even be omitted, as the results from the original trained model did quite well across the year.

3.8 Training with All Data

Though we have shown it is sufficient to train using a single day's worth of representative jobs, we obtained simulated average wait times for jobs for 25 days throughout the year. We wanted to explore the performance of a recommender system trained on all data gathered and compare and contrast its performance with the recommender system described above. For each of the 25 days simulated, the average wait time varied depending on the jobs which were submitted on those days. Figure 3.7 shows boxplots of the average wait time by day depicting the these variations. As such, these values were scaled using min-max normalization prior to regression using the following formula:

$$Normalized\_value = \frac{(actual\_value - min\_value)}{(max\_value - min\_value)}$$

Performing this normalization across each day transforms the average wait time values into a a unitless value between 0 and 1 where values closer to zero represent the best perform-

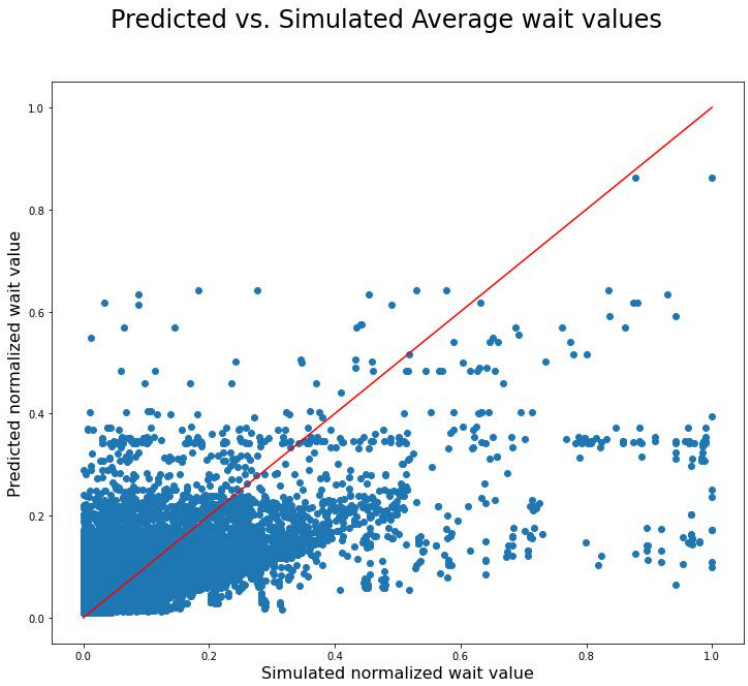

Figure 3.9: The predicted vs. simulated normalized average wait time values.

ing server sets with the lowest average wait time. Though the predictions by the regression model will no longer predict the number of seconds of average wait time a server compositions is expected to to have, predicted lower values still represent server packages with lower expected average wait time for jobs. The regression model was not as accurate as when training using a single representative set of jobs, as depicted in Figure 3.9. Again, if the regression model were perfect, all the predicted vs. actual values would lie upon the $y = x$ line of the graph. Though this model using normalization does not appear to be as good as the previous one, some loss of precision is expected when normalizing in this manner. We can still use the regression model to power a recommender system and evaluate its performance.

Table 3.9 shows the results of the normalized model when the hit threshold is greater than 16. Using the same $k = 50$ value from before, we achieve a precision@50 of 94%, which is thought to be excellent. In other words, 47 out of the top 50 recommendations returned by this model will be among the top 10% of performant server combinations for 17 or more days throughout the year. Though this model does slightly better than the model trained

on one representative days' worth of jobs, it took 24 times more computation to provide the data to train it. Though the original model trained using a single day achieved a lower precision, it was still able to return good results across the year and required substantially less computation time.

Table 3.9: Precision@k and Recall@k when Hit Threshold >16 for Normalized Model

Hit Threshold	k	precision@k	recall@k	F1@k
>16	10	1.00	0.026	0.05
>16	50	0.94	0.12	0.22
>16	100	0.78	0.20	0.32
>16	386	0.61	0.61	0.61
>16	500	0.54	0.70	0.61
>16	1000	0.32	0.83	0.46
>16	5000	0.08	1.00	0.14
>16	10000	0.04	1.00	0.07

The results of summing the top 50 recommended server sets can be found in Table 3.10. These differ slightly from the recommendations of the model trained using a single representative days' worth of jobs. In both models, the more expensive compute node with more cores was preferred, however the model trained on all the days prefers the compute node with more memory. The model trained on a single day preferred the cheaper big memory node, and the model trained on all the days preferred the more expensive one. Both models preferred GPU nodes with fewer GPUs. Though they differ slightly in the nodes types and quantities they prefer, they are fairly close with their recommendations, and it is thought that the original model using only a single representative day's worth of jobs would provide adequate enough recommendations for a HPC system administrator to evaluate and arrive at a good server combination to purchase.

Table 3.10: Recommendations Drawn from Normalized Model Predicted Results

Node Type	*Node Description*	*Sum of Servers Across Top 50*
Compute Nodes	Low Cost CPU w/ 256Gb	0
	Low Cost CPU w/ 512Gb	0
	High Cost CPU w/ 256Gb	1,667
	High Cost CPU w/ 512Gb	1,764
Big Memory Nodes	Low Cost CPU w/ 1024Gb	34
	High Cost CPU w/ 1024Gb	751
GPU Nodes	2 GPUs in one server	341
	4 GPUs in one server	108
	8 GPUs in one server	48

3.9 Conclusions

We began with roughly 127,000 different possible server packages which could have been purchased under our budget. We uniformly sampled 10% of these, leaving us with roughly 12,700 different server packages. We chose a representative days worth of jobs from local HPC log data, and simulated the scheduling of these jobs on the 12,700 server packages, so we could calculate the average jobs wait time for a given composition of servers. By developing an XGBoost regression model, we were able to predict the average job wait time for the unsimulated server compositions. Finally, we were able to verify that the recommender system made good recommendations by choosing different sets of jobs spaced throughout the year and simulating the scheduling of different job workloads using those server sets. An administrator considering purchase options has an 88% chance of selecting a top performing server packaged under their budget if they were to choose one from the top 50 recommendations returned by the recommender system. By simulating the performance of a small minority of server packages, our recommender system was able to make excellent recommendations.

The most benefit from using this system comes from the time saved doing simulations. The roughly 127,000 initial server combinations could be effectively summarized by simulating only 10% of them on a single representative set of jobs, and it proved unnecessary to

conduct simulations for days spaced throughout the year. This was done for the research in order to evaluate the performance of the recommender system, and explore its feasibility when used to optimize hardware procurement for HPC systems.

This recommender system is not intended to replace the expertise of HPC administrators when it comes to decisions for hardware procurement. It is our hope that this tool can provide a data-driven technique which will help narrow the search space with which administrators are confronted when they make procurement decisions. Returning to the research question: experimental simulation coupled with a regression model enabled a recommender system to return server compositions under a given budget with low average wait times with a precision@50 of 92%. Additionally, the discrete event simulator, job data set, machine learning code, and recommender system code are released under the GPLv3 license should other researchers find it useful (https://github.com/shutchison/Optimal-Hardware-Procurement-for-a-HPC-Expansion).

Chapter 4

Scheduling for High Performance Computing with Reinforcement Learning

4.1 Preface

This research was originally presented at 2024 OkIP International Conference on Advances in Parallel and Distributed Computing (APDC)[47]. This article has been reproduced with permission from Oklahoma International Publishing.

4.2 Introduction

Scheduling for High Performance Computing (HPC) systems is typically done using a batch scheduler. In most HPC systems, users will submit their jobs to a centralized job scheduler that will reserve and assign HPC resources according to the resources requested by the users at submission time. Typically, the system administrators managing the HPC system will configure the scheduler using an optimization goal, or metric, such as *maximizing HPC resource utilization, minimizing job wait time, maximizing job throughput*, etc. The opti-

mization goal of system administrators may change from one time period to the next, and different HPC administrators may value one metric over another. Correctly configuring the batch scheduler to optimize for different metrics is challenging, and optimizing for one metric may adversely affect another. This research shows that reinforcement learning can learn different scheduling policies to optimize for different goals while remaining competitive with or performing better than algorithmic scheduling baselines.

Although batch scheduling has been shown to be an NP-Hard problem[48], some job schedulers compute job priorities based on attributes of the submitted job. Examples of algorithmic scheduling based on job attributes include First Come First Serve (FCFS), Shortest Job First (SJF), Oracle SJF, and Best Fit Bin Packing (BFBP). More details of these algorithms are provided in Section II.A. More sophisticated schedulers use advanced techniques such as utility functions or machine learning to make their scheduling decisions. Recently, researchers have looked to Reinforcement learning (RL), one of the machine learning paradigms, to learn scheduling policies for HPC scheduling applications. With RL, we allow a machine learning agent to make scheduling decisions and provide feedback on its performance using a reward. When trained iteratively, the agent can learn a scheduling policy to maximize the reward it receives. When this reward is tied to the desired scheduling optimization goal, the RL agent makes scheduling decisions to optimize for the chosen goal. The questions this research set out to answer are as follows:

- Can RL provide a high-quality scheduling policy comparable to or better than algorithmic scheduling baselines?
- Is the learned policy only effective on the workload used for training, or can it generalize and remain effective on other previously unseen workloads?

The remainder of this paper is structured as follows: Section 4.3 provides details about the background and related works, as well as further details of the implementation of this work. Section 4.4 discusses the methodology and a comparison of the performance of a

RL scheduling agent with algorithmic scheduling baselines. Section 4.5 discusses the results of the experiment, the statistical analysis done, and the conclusions reached. Section 4.6 expands upon the challenges encountered in the work, as well as prospects for future work. Finally, section 4.7 offers final concluding remarks.

4.3 Background and Related Works

4.3.1 Algorithmic Scheduling Baselines

The baseline scheduling algorithms to which the RL agent's performance will be compared are First Come First Serve (FCFS), Shortest Job First (SJF), Oracle SJF, and the Best Fit Bin Packing (BFBP) Algorithm. For FCFS, jobs are scheduled strictly in the order in which they arrive. If no machine in the HPC cluster has sufficient resources to execute the first chronological job in the queue, the scheduler waits until there are adequate resources to begin the execution of the first job in the queue. SJF will sort the job queue by requested job run time and begin running the shortest job that some machine in the cluster has adequate resources to execute. In practice, users of the HPC system tend to overestimate the amount of time and resources their jobs require, as they do not want their job's execution to be halted by exceeding their requested resources. Overestimation of resources by the users, including requested job time, tends to degrade the performance of scheduling algorithms like SJF. Oracle SJF behaves like SJF except it sorts the job queue by actual run time instead of requested run time. This information cannot be known by the scheduler at job submission time when the scheduler is making its decisions, and scheduling using this information provides a decent lower bound for the average job waiting time. BFBP is perhaps the most relevant scheduling algorithm as it powers some scheduling applications in use on actual HPC systems today, like Slurm[22]. BFBP considers all jobs in the queue and the available resources for each machine in the cluster. BFBP selects the (job, machine) pair that will result in the fewest resources remaining for some machine in the cluster and begins executing

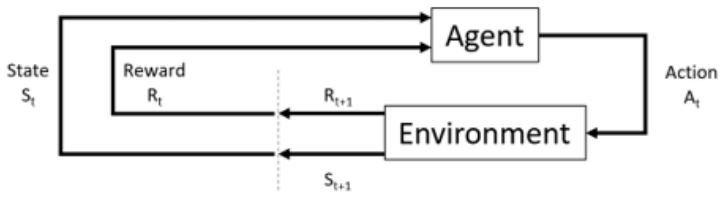

Figure 4.1: The general framework for reinforcement learning

that job on the chosen machine. This has been shown to take no more than $\lfloor 1.7 * OPT \rfloor$ machines[39], where OPT is the minimum number of machines required for a particular workload. Not only does this scheduling algorithm power some modern scheduling applications, but it also serves to provide a decent upper bound for the HPC cluster utilization metric for any particular workload.

4.3.2 Reinforcement Learning for Policy Optimization

In general, a RL agent is attempting to choose the best action to maximize its reward given the current state of the environment. The agent has knowledge of the environment through its observation space, and it sees the environment in state S at time t. The agent has certain actions available to it through its action space. The agent chooses action A at time t. This action will change the environment to a new state S at time $t+1$. The agent then receives some reward R at time $t+1$ for its action. Based on the reward, the weights for the neural network powering the agent's decision-making process are updated and the process repeats until the reward converges to a maximum. The policy the agent learns over time will maximize the expected reward for a particular action given the current state of the observation space. The general framework for RL is shown in Fig. 4.1.

In the context of using RL to schedule for HPC systems, the observation space consists of the jobs in the job queue and the resources available on each machine. The actions available to the agent are selecting a particular job from the job queue and a machine from the cluster on which to run it. When the reward is tied to the optimization goal (job wait time or HPC

utilization), the agent will learn to select the best job from the job queue and the machine on which to run it that will optimize for the desired metric. Metrics may be maximized (as with HPC system utilization) or minimized (as with job wait time). It was the goal of this research to ascertain if RL could learn multiple scheduling policies to optimize for different metrics, and if those policies could compete with algorithmic scheduling baselines on new, never-before-seen workloads. Additional details about the implementation used for this research can be found in Section II.E. Proximal Policy Optimization (PPO)[12] is an RL technique developed to improve upon and address some shortcomings in previous RL techniques. PPO is an on-policy technique, meaning that each update of the policy only uses data collected while acting according to the most recent version of the policy. The policy maps the states in the environment to actions taken when in those states. Generally, an objective function in RL returns the expected reward for an action in a given state. Rather than using gradient ascent on the objective function to optimize the policy (a computationally expensive process), PPO employs a surrogate objective function which gives a conservative estimate for how much the objective function will change as a result of a policy update. Large policy updates are penalized via clipping, resulting in quick training convergence and good performance for many tasks.

4.3.3 HPC Scheduling using Machine Learning

Scheduling for HPC systems using RL techniques has been a topic of much research interest recently. One of the first attempts to do HPC scheduling with RL was accomplished with DeepRM[49]. Mao et al. showed that RL can learn multiple scheduling policies to compete with state-of-the-art heuristics when scheduling HPC jobs. Their RL agent uses gradient decent on policy parameters to maximize the expected cumulative discounted reward. It was shown to adapt to different conditions, converge quickly, and learn sensible scheduling strategies. RLSchert[50] was a more recent project using RL to perform HPC scheduling. This project included a remaining time predictor to better estimate how long a job will take in

order to make better scheduling decisions. RLSchert incorporated requested memory and requested CPUs as resource constraints and learns a policy to select or kill jobs according to their status and estimated remaining time using the PPO technique. Another RL HPC scheduler is A2cScheduler[51]. This technique uses the actor-critic deep-RL technique to perform scheduling and resource management for HPC systems. A2cScheduler also considers two job resources constraints, requested memory and requested CPUs. The research most closely related this work is RLScheduler[52]. Zhang et al. showed the viability of using RL and PPO to learn multiple scheduling policies for both real and constructed HPC workloads. However, the neural network powering RLScheduler only selects the next job to be started from the job queue. The machine on which the job is run is not selected by the agent, and the job is handed off to the cluster for execution on some machine. Also, each machine in the cluster is limited to running only one job at a time, which is a technique employed by some HPC systems. However, many HPC systems allow the allocation of multiple jobs to a single HPC node as resources allow, resulting in increased job throughput and higher HPC utilization. Additionally, RLScheduler only considers two resource constraints for each job, the job's requested memory and number of requested CPUs. It was the goal of this research to build upon what was accomplished with RLScheduler by including the following:

- Increase the number of constraints each job can request from two to three by allowing jobs to request memory, CPUs, and GPUs.
- Allow multiple jobs to execute on a single HPC machine while still respecting the machine's total resource constraints.
- Allow the RL agent to select not only a job from the queue, but also the machine in the cluster on which to run it.

Accomplishing these should bring HPC scheduling with RL one step closer to implementation on an actual HPC system.

4.3.4 Workload Specification with Three Resource Constraints

The jobs given to the scheduler to be scheduled comprise the workload. There are numerous HPC workloads available for use, and we have access to the log data of a local university HPC system to construct workloads for our use. A trend with HPC scheduling research thus far seems to be the consideration of only two resource constraints per job: the amount of requested memory and the number of requested CPUs. We believe this stems from a limitation of the Standard Workload Format (SWF)[53] method of specifying workloads for HPC systems. The SWF has become the de facto standard for HPC scheduling research and is used extensively in this space. Although the resource constraints included in the SWF are certainly important, analysis of our local HPC system has shown that jobs requesting GPU resources have become increasingly prevalent. This trend will likely continue as future AI researchers and others capitalize on GPUs and other hardware accelerators. In fact, on our HPC system, we have frequently observed low GPU availability being a key factor in extending job wait time. Alternatives to the SWF have been proposed, such as the Modular Workload Format (MWF) (Corbalan and D'Amico, 2021). The MWF incorporates not only non-classical computing resources (like GPUs), but also additional new job profiles as well. For this reason, consideration of a third resource constraint, requested GPUs, was seen as important for this work, as it is often overlooked in prior HPC scheduling research. Workloads were specified using a simple comma separated value (CSV) format, which included the following information about each job: *JobName, RequestedMemory, RequestedCPUs, RequestedGPUs, RequestedDuration, ActualDuration*, and *SubmitTime*. The characteristics of the machines comprising the simulated HPC cluster were specified using another CSV file with the following attributes: *MachineName, TotalMemory, TotalCPUs*, and *TotalGPUs*.

4.3.5 OpenAI Gym Environment

OpenAI Gym[54] is an open-source Python library and Application Programming Interface (API) for developing and comparing RL algorithms. OpenAI Gym specifies methods for

implementing a custom environment to explore machine learning tasks. Within the custom environment, the observation space, action space, step function, and various others are defined. The RL agent's neural network provides a mapping from the observation space to the action space and allows the agent to maximize the expected return for its available actions. Invalid action masking[13] eliminates any impossible or clearly unproductive actions. In the context of our HPC Scheduling problem, invalid action masking removes any (job, machine) actions for which the machine does not have adequate available resources to run the job. Invalid action masking reduces the number of possible actions the agent must consider and has been shown to improve convergence time during training, as the agent will not attempt to schedule a job on a machine that cannot run it. If the agent were to choose an invalid action, the state of the observation space would not change, and the workload would not move any closer towards completion. Stable Baselines3[55] (SB3) is a Python library that provides a set of reliable implementations of RL algorithms, and this research implemented a custom OpenAI Gym environment using a self-implemented discrete event simulator (DES) representing the HPC scheduling problem. The SB3 PPO with invalid action masking algorithm was used to train a RL agent, and then its performance was compared to algorithmic scheduling algorithms implemented in the DES. The DES keeps track of the current simulation time step and maintains lists for future jobs, queued jobs, running jobs, and completed jobs. As the simulation progresses and simulation time advances, jobs move through the lists in the DES, machines track their currently available resources. Upon simulation termination, the desired metrics can be calculated and compared to one another depending on the algorithm used to schedule the jobs to the simulated HPC.

The custom OpenAI Gym environment provides the definitions of the observation space and the action space. The observation space consisted of the first n schedulable jobs in the job queue. A job was considered schedulable if at least one machine in the cluster had adequate resources to schedule it. Many scheduling applications limit a maximum queue depth to search for jobs to schedule, and the code for this research was written to allow

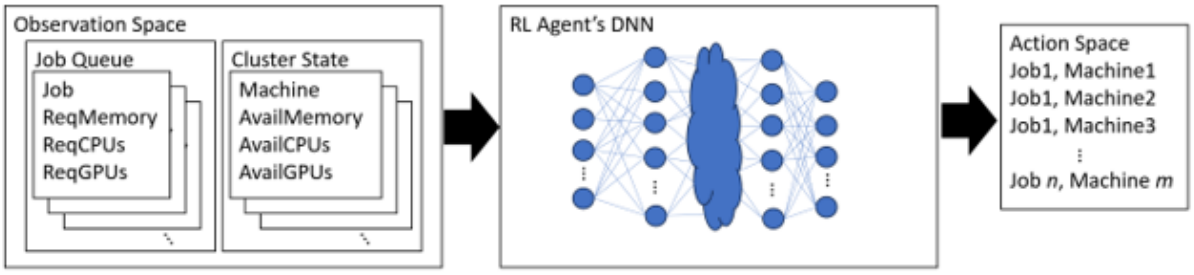

Figure 4.2: The observation space and action space of the custom OpenAI Gym environment

adjustment of the queue depth easily. The action space selects an individual schedulable job from the job queue and a machine on which to run it. This structure is depicted in Fig. 4.2. This was initially implemented as a multi-discrete action space, however, many scheduling algorithms require a discrete action space. This 2D action space was converted to a 1D action space similar to converting a 2D array into a 1D array using the following formula:

$$1D\_action = (machine\_index * num\_machines) + job\_index \tag{4.1}$$

The reward returned at each training step is tied to the desired policy. As implemented, the SB3 algorithm will maximize this episodic reward, so when minimizing job queue time, the reward per step was *reward = 0 - avg. job wait time*. When maximizing system utilization, *reward = HPC System utilization*. An episode consists of scheduling every job in a workload, and the cumulative reward across the entire episode is used to update the RL agent's learned scheduling policy.

4.4 Methodology

To evaluate the efficacy and the performance of using PPO with invalid action masking when scheduling jobs for HPC systems, the follow steps were taken:

- Construct 20 sets of jobs consisting of high utilization jobs (jobs requesting more than

20 CPUs) and modify the job's submission times to be the same. Add 2% of jobs requesting any number of GPU resources to the set.

- For each set of constructed jobs, train an agent using PPO with invalid action masking to minimize job wait time and another agent to maximize system utilization.
- Select 23 days' worth of jobs from HPC log data of days with approximately the same number of submitted jobs (1000 +/- 100 jobs).
- Schedule these 23 days using the algorithmic baselines.
- Use the trained agent to schedule these 23 days and compare its performance to the algorithmic scheduling baselines.
- Conduct analysis of variance (ANOVA) to determine if the differences of the means of the scheduling methods is statistically significant.
- For agents trained to minimize average job wait time, conduct pairwise t-tests between the agent's performance and the algorithmic baseline's average job wait time for the 23 days.
- For agents trained to maximize HPC system utilization, conduct pairwise t-tests between the agent's performance and the algorithmic baseline's average HPC system utilization for the 23 days.

4.4.1 Training Workload Construction

To construct the training workload, jobs from log data for the local HPC systems were sampled. Zhang et al. noted that the workload trajectory was important for productive training, and they utilized trajectory filtering to select only job traces that were productive for training their RL agent. Various training workloads from local HPC log data were tested for training, and the most productive and consistent training came from selecting "large

jobs" that requested more than 20 CPUs. Training workloads were constructed by randomly selecting 2000 jobs requesting more than 20 CPUs from HPC log data, and randomly adding 40 jobs requesting any number of GPUs. This 2% addition of jobs requesting GPUs aligns with the historic level of such jobs from our HPC system. Then the jobs were shuffled, and modifying the submission time for all jobs such that they were submitted simultaneously. Essentially, we wanted challenging workloads for the scheduler, which would reward good scheduling decisions and punish poor ones. Each of the 20 workloads constructed was used to train a RL agent using two metrics: minimizing the average job wait time and maximizing HPC system utilization.

4.4.2 Evaluation on Actual Workloads

Again, log data was used to find days which had roughly the same number of submitted jobs. We sought days with enough jobs that scheduling would be a non-trivial activity for our simulated cluster of nine machines. Searching log data for days with 1000 +/- 100 submitted jobs yielded 23 different days from the log data considered. The resources requested, the submission time, and the actual run time for these jobs remained unchanged from the log data, and the 40 different agents were used to schedule the 23 days using the metric on which they were trained. The algorithmic baselines also scheduled the 23 days to provide a basis for comparison of the performance of the RL agent's scheduling.

4.4.3 Statistical Analysis

ANOVA[56] was utilized to determine if there was a statistically significant difference between the means of the models and the algorithmic scheduling baselines. Next, pairwise t-tests[57] were conducted between each algorithmic scheduling baseline and the model to determine if difference between the means of the models and the algorithmic baseline was statistically significant. A significance level of 95% ($\alpha = 0.05$) was used for all statistical tests.

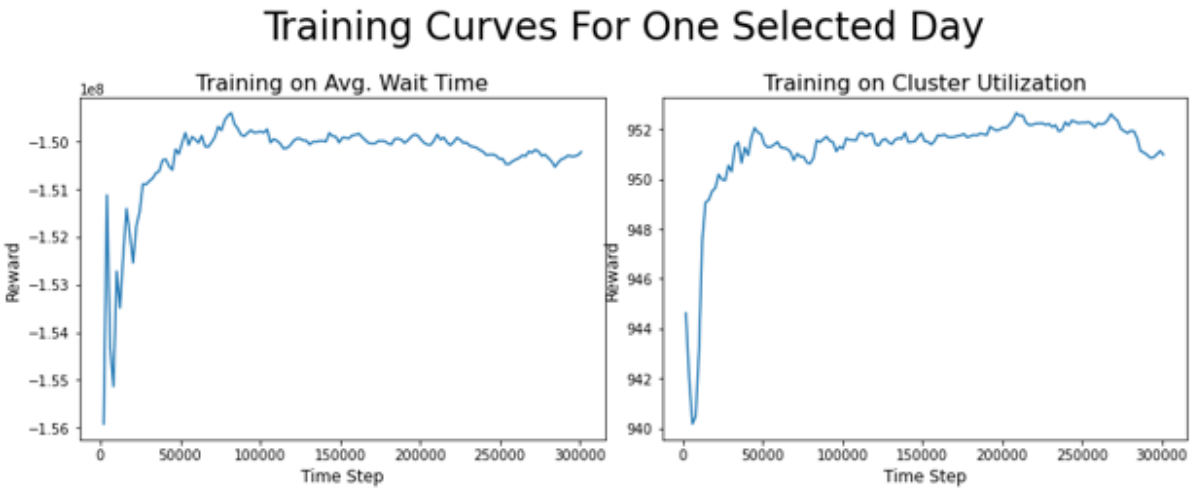

Figure 4.3: The training curve of two RL agents on a selected day when trained for 300,000 steps on each of the two metrics

4.5 Results

4.5.1 Training Convergence

Figure Fig. 4.3 shows the training curves for two agents trained on one set of constructed jobs. One agent was trained to minimize average job queue time, and the other was trained to maximize HPC system utilization. The training for the depicted day converges quickly to some maximal value, indicating that the agent has learned how to optimally schedule a chosen day for its given metric. Training was accomplished using HPC resources in parallel, and training each agent took no more than a few hours.

4.5.2 Minimizing Average Job Wait Time

Each of the 20 models trained to minimize average job wait time was used to schedule 23 days of jobs, and the performance of the model was compared to the algorithmic baselines of FCFS, Oracle SJF, SJF, and BFBP. Conducting ANOVA on the results showed there was a statistically significant difference between the average job wait times for the different

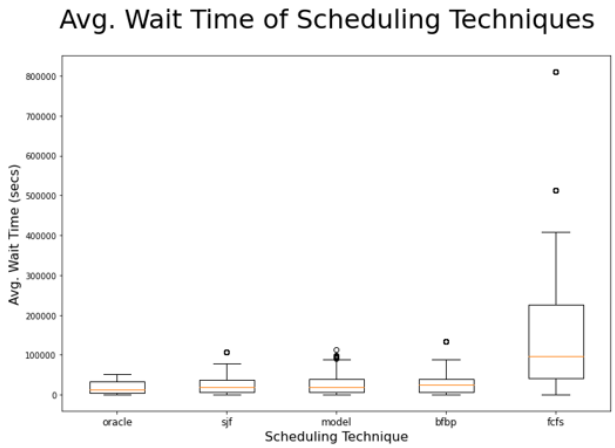

Figure 4.4: Boxplot of average job wait times for the different scheudling techniques (lower is better).

scheduling algorithms (p-value $=1.395*10^{-15}$). Doing pairwise t-tests comparing the different scheduling methods to the model showed significance between the model and Oracle, BFBP, and FCFS. When using the metric of minimizing average job wait time, both SJF and Oracle scheduling performed better than the model. This is unsurprising as these scheduling algorithms are designed to minimize this metric. The model was able to schedule jobs such that the average job wait time was 18.44% lower than if they were scheduled using the BFBP algorithm. Table 4.1 shows the mean average job wait time when scheduling all 23 days using the various scheduling techniques with statistically significant paired t-test p-values highlighted. A boxplot of the average job queue time can be found in Fig. 4.4.

Table 4.1: Results for Minimizing Average Job Wait Time

Scheduling Method	Avg. Job Wait Time in Minutes (Lower is Better)	T-test P-value vs. the Model
Oracle SJF	308.55	$3.125 * 10^{-10}$
SJF	454.05	0.9238
RL Model	456.79	n/a
BFBP	549.59	0.003718
FCFS	2779.20	$2.2 * 10^{-16}$

4.5.3 Maximizing HPC System Utilization

Next, each of the 20 models trained to maximize HPC system utilization was used to schedule 23 days of jobs, and the performance of the model was compared to the same algorithmic baselines. Conducting ANOVA on the results showed there was a significant difference between the HPC system utilization for the different scheduling algorithms (p-value $=2.2 * 10^{-16}$). Doing pairwise t-test comparisons between the different scheduling methods showed significance between the performance of the model and FCFS. In terms of maximizing cluster utilization, BFBP did the best with the cluster utilization of 45.63% compared to the model's cluster utilization of 45.61%. The BFBP algorithm was designed to maximize this metric by maximizing the utilization of each of the machines in the cluster, so this is also unsurprising. Additionally, the inconclusive t-test indicates that we cannot reject the null hypothesis that the difference in the means of BFBP and the model is due to random chance. We cannot conclude that the model does better, but it is at least able to perform comparably to the algorithmic baselines when trained to maximize this metric. Table 4.2 shows the mean HPC system utilization when scheduling all 23 days using the various scheduling techniques with statistically significant paired t-tests highlighted. A boxplot of the average HPC system utilization can be found in Fig. 4.5.

Table 4.2: Results for Maximizing HPC System Utilization

Scheduling Method	Cluster Utilization (Higher is Better)	T-test P-value vs. the Model
FCFS	41.26%	$2.2 * 10^{-16}$
SJF	44.73%	0.05552
Oracle SJF	45.18%	0.3567
RL Model	45.61%	n/a
BFBP	45.63%	0.9671

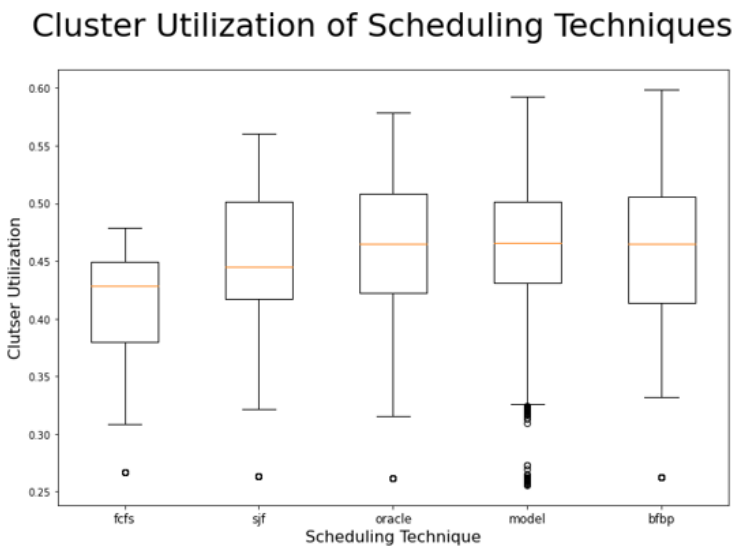

Figure 4.5: Boxplot of cluster utilization for the different scheduling techniques (higher is better).

4.5.4 Interpretation

The RL agent trained using PPO with invalid action masking was able to beat or at least perform comparably to the algorithmic scheduling baselines. The agent's performance when minimizing average job queue time was 18.44% better than the performance of BFBP, a scheduling algorithm used to power modern scheduling applications. When maximizing HPC system utilization, there was no difference between the agent's performance and that of the algorithmic baselines, including BFBP. Returning to the research questions, it was demonstrated that RL can provide a high-quality scheduling policy comparable to or better than algorithmic scheduling baselines. Additionally, since the RL agent was trained on constructed workloads, and then evaluated using different workloads actually submitted to our local HPC system, it was able to successfully apply the scheduling policy it learned on one workload to another. Additionally, RL was able to accommodate three resource constraints per job (requested memory, requested CPUs, and requested GPUs), and successfully schedule on par with algorithmic scheduling baselines. Furthermore, the RL agent was able to schedule multiple jobs per machine and respect the resource constraints on each machine. The RL

agent was able to select not only the job from the queue to schedule next, but also the machine on which to run it, similar to modern scheduling applications. Finally, this RL technique could learn different scheduling policies, one which was minimized average job queue time and one which was maximized HPC system utilization, showing its flexibility and applicability to the needs of different HPC system administrators.

4.6 Challenges and Future Work

Curiously, increasing the queue depth available to the RL agent beyond a certain point did not improve training convergence or the agent's performance. Queue depths of 10, 60, 100, 200, and 500 were investigated, and degradation of training value began when queue depth was greater than 100. One would think that being able to look deeper in the queue would allow for better scheduling performance. It is thought that size of the observation space was causing difficulty with the SB3 implementation of PPO with invalid action masking. The size of the observation space for our implementation was as follows:

$$observation\_space = queue\_depth * num\_HPC\_machines \tag{4.2}$$

As the queue depth, or the number of HPC machines increased, so did the size of the observation space. Beyond a certain point queue depth of HPC size, the sum of the probabilities of actions exceeded a threshold set in SB3. This challenge might be overcome with another implementation of PPO with invalid action masking, but it bears further investigation. To build the observation space also requires maintaining a list of schedulable jobs. This requires iterating through each machine in the cluster for every job in the job queue to see if any machine currently has adequate resources to schedule a particular job. This operation has *O(n*m)* time complexity, where n was the queue depth and m was the number of machines in the cluster, significantly slowed down the speed of scheduling using RL. This operation could be sped up significantly if the RL agent were first trained to choose only

schedulable jobs from the queue and then given the task to choose a schedulable job from the queue and machine on which to run it. Scheduling using BFBP requires the same *O(n*m)* operations to make its decisions, but with further algorithmic refinement, scheduling using RL could not only perform comparably to scheduling with algorithmic baselines in terms of HPC metrics, but it also would be faster and more performant. The speed of making scheduling decisions was outside the scope of what was investigated with this research, but it's likely that with some algorithmic optimizations, scheduling with RL could be faster than these algorithmic baselines as well.

Some scheduling applications, like Slurm, allow for custom user-provided plugins for scheduling. It would be of great interest to explore how a RL powered scheduling plugin for Slurm would perform on an actual or simulated HPC system. Much integration work would be required, but we believe this work demonstrates the viability of HPC scheduling with RL and moves it one step closer to an actual implementation on an HPC system.

4.7 Conclusions

This work showed PPO with invalid action masking can perform better than or comparable to algorithmic scheduling baselines when scheduling for HPC systems. The RL agent was able to select not only the next job from the job queue, but also the machine on which to run it all while accommodating multiple resource constraints on each machine. Also, within the discrete event simulator, multiple jobs could execute simultaneously on any machine in the simulated cluster while still respecting the machine's resource constraints. The code used to conduct this research has been released under the GPL v3.0 license should others find it useful.

Chapter 5

Final Conclusions

Returning to the original problem statement, machine learning techniques were critical in finding solutions for the presented problems in the HPC domain.

5.1 Scalability/Transferability of Methods

Chapter 2 described a method for the prediction of predicting queue time for jobs submitted to an HPC system, as well as a method for predicting whether or not the queue time for an HPC system will increase or decrease in the near future. It is thought that this method could be applied successfully to any other HPC system. More than 730,000 jobs were used to develop the regression machine learning pipeline as described. The speed and robustness of modern ML software libraries mean that training the pipeline and multiple GBTR models would not take a prohibitively long time.

For the optimal hardware procurement recommender system described in Chapter 3, it is thought that this method would also be easily scalable to larger budgets and HPC systems. The limiting and most time consuming portion of this research was conducting the simulations using DES. It was shown that a using only 10% of the potential server package options was sufficient to build a regression model to power the recommender system. An ablation study could be conducted to determine how few representative server packages would

be necessary to make a regression model with sufficient accuracy to still obtain adequate precision@k. An advantage of this method is that the simulations with the DES can be conducted in parallel. If more simulations were needed and adequate HPC resources were available for the simulations, the time required would not scale linearly as the difficulty of the problem increased.

The final contribution described in Chapter 4 would be more difficult to scale up. As discuss, as the size of the observation space the RL agent had visibility on increased beyond a certain point, the SB3 implementation of PPO with IAM began having errors and warnings. Further work would be required to track down the issues encountered. The SB3 library was sufficient for the size of the problem discussed, however, as the size of the problem increased, so too did the corresponding observation space.

5.2 Future Work

There are several extensions possible from the contributions mentioned. In Chapter 2, two different regression models were developed: one which predicts whether of not the queue time of an HPC system will increase in the future and one which predicts the expected queue time of submitted jobs. It does not take a great intuitive leap to see that the first could feed into the second. If average job queue time for the system is expected to increase, this could be an additional input into the second regression model predicting the job queue time which might increase the accuracy of the second regression model. Though this was not explored in the research as presented, it is expected that the prediction accuracy of the job time would increase with this additional input. It was beyond the scope of what was explored, however, it would be an interesting next step.

An additional possible future work can be derived from the RL powered scheduling agent described in Chapter 4. The scheduling application used at KSU is Slurm, and Slurm supporting using user provided plugins for scheduling. The research presented shows that

RL scheduling does work when scheduling with the self-developed DES, but in the presented work, it was not implemented using a production quality scheduling application. It would be of great interest if a RL agent powered scheduling agent could be implemented as a plug in for either a Slurm simulator or an actual production system to see if the benefits could be maintained closer to live HPC systems.

An area of research not previously mentioned in this work but showed great promise was High Performance Computing Scheduler Parameter Optimization using Simulation and Regression[58]. This contribution attempted to use machine learning to assist with discovering an optimal configuration of a scheduling application for HPC systems. This led to an international collaboration with the Federal University of Rio Grande do Norte in Brazil. Though this work failed to produce any results beyond the initial research poster stage, it was a promising area which would merit further investigation.

5.3 Final Remarks

Finally, the robustness and completeness of the modern machine learning libraries discussed throughout this work is nothing short of remarkable. The authors and maintainers of these libraries have provided well-documented and high-quality implementations of many different machine learning algorithms. These can be quickly applied and tested when determining the most effective technique to solve a problem regardless of the problem's domain. This work focused on solving problems for HPC systems and HPC system administrators, however, assisted by these ML libraries, anyone is able to quickly test the state-of-the-art ML techniques to accomplish whatever problem they are trying to solve. The author of this work is indebted to the authors, contributors, and maintainers of these ML libraries.

Bibliography

[1] k-menas clustering. https://en.wikipedia.org/wiki/K-means_clustering. Accessed: 2024-04-2.

[2] linear regression. https://en.wikipedia.org/wiki/Linear_regression. Accessed: 2024-04-2.

[3] Tesla autopilot. https://www.tesla.com/autopilot. Accessed: 2024-03-20.

[4] Gpt3. https://openai.com/blog/gpt-3-apps. Accessed: 2024-03-20.

[5] Samsung bixby. https://www.samsung.com/us/apps/bixby/. Accessed: 2024-04-2.

[6] Apple siri. https://www.apple.com/siri/. Accessed: 2024-04-2.

[7] Bruce Hendrickson et al. The future of computation science. In *SIAM Task Force Report: The Future of Computation Science*. SIAM, 2024. Accessed: 2024-04-02.

[8] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[9] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[10] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

[11] Brook Taylor. *Methodus incrementorum directa & inversa*. Inny, 1717.

[12] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[13] Shengyi Huang and Santiago Ontañón. A closer look at invalid action masking in policy gradient algorithms. *arXiv preprint arXiv:2006.14171*, 2020.

[14] Scott Hutchison, Daniel Andresen, Mitchell Neilsen, William Hsu, and Benjamin Parsons. High performance computing queue time prediction using clustering and regression. In *International Conference on Parallel Processing and Applied Mathematics*, pages 260–272. Springer, 2022.

[15] Rajath Kumar and Sathish Vadhiyar. Prediction of queue waiting times for metascheduling on parallel batch systems. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 108–128. Springer, 2014.

[16] Vytautas Jancauskas, Tomasz Piontek, Piotr Kopta, and Bartosz Bosak. Predicting queue wait time probabilities for multi-scale computing. *Philosophical Transactions of the Royal Society A*, 377(2142):20180151, 2019.

[17] Nick Brown, Gordon Gibb, Evgenij Belikov, and Rupert Nash. Predicting batch queue job wait times for informed scheduling of urgent hpc workloads. *arXiv preprint arXiv:2204.13543*, 2022.

[18] Aristidis Likas, Nikos Vlassis, and Jakob J Verbeek. The global k-means clustering algorithm. *Pattern recognition*, 36(2):451–461, 2003.

[19] João Henriques, Filipe Caldeira, Tiago Cruz, and Paulo Simões. Combining k-means and xgboost models for anomaly detection using log datasets. *Electronics*, 9(7):1164, 2020.

[20] Yahui Liu, Huan Luo, Bing Zhao, Xiaoyong Zhao, and Zongda Han. Short-term power load forecasting based on clustering and xgboost method. In *2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS)*, pages 536–539. IEEE, 2018.

[21] Donald J Becker, Thomas Sterling, Daniel Savarese, John E Dorband, Udaya A Ranawak, and Charles V Packer. Beowulf: A parallel workstation for scientific computation. In *Proceedings, international conference on parallel processing*, volume 95, pages 11–14, 1995.

[22] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Workshop on job scheduling strategies for parallel processing*, pages 44–60. Springer, 2003.

[23] Karl Pearson. Note on regression and inheritance in the case of two parents. *Proceedings of the royal society of London*, 58(347-352):240–242, 1895.

[24] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[25] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009. ISBN 1441412697.

[26] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

[27] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. doi: 10.1109/MCSE.2007.55.

[28] Robert L. Thorndike. Who belongs in the family. *Psychometrika*, pages 267–276, 1953.

[29] Adedolapo Okanlawon, Huichen Yang, Avishek Bose, William Hsu, Dan Andresen, and Mohammed Tanash. Feature selection for learning to predict outcomes of compute cluster jobs with application to decision support. In *2020 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 1231–1236. IEEE, 2020.

[30] Mohammed Tanash, Brandon Dunn, Daniel Andresen, William Hsu, Huichen Yang, and Adedolapo Okanlawon. Improving hpc system performance by predicting job resources via supervised machine learning. In *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (learning)*, pages 1–8. 2019.

[31] Scott Hutchison et al. Optimized hardware configuration for high performance computing systems. In *Proceedings of the 17th International Conference on Advanced Engineering Computing and Applications in Sciences (ADVCOMP 2023)*. International Academy, Research, and Industry Association, 2023. ISBN 978-1-68558-107-7. URL https://www.iaria.org/conferences2023/CfPADVCOMP23.html.

[32] R. Pordes et al. The open science grid. In *J. Phys. Conf. Ser.*, volume 78 of *78*, page 012057, 2007. doi: 10.1088/1742-6596/78/1/012057.

[33] Igor Sfiligoi et al. The pilot way to grid resources using glideinwms. In *2009 WRI World Congress on Computer Science and Information Engineering*, volume 2 of *2*, pages 428–432, 2009. doi: 10.1109/CSIE.2009.950.

[34] The great plains augmented regional gateway to the open science grid. URL https://gp-argo.greatplains.net/. https://gp-argo.greatplains.net/. Accessed 2023-01-18.

[35] Richard Todd Evans et al. Optimizing gpu-enhanced hpc system and cloud procurements for scientific workloads. In *International Conference on High Performance Computing*, pages 313–331. Springer, 2021.

[36] Carsten Kutzner et al. More bang for your buck: Improved use of gpu nodes for gromacs 2018. *Journal of computational chemistry*, 40(27):2418–2431, 2019.

[37] Dror G Feitelson, Dan Tsafrir, and David Krakov. Experience with using the parallel workloads archive. *Journal of Parallel and Distributed Computing*, 74(10):2967–2982, 2014.

[38] Silvano Martello and Paolo Toth. *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc., 1990.

[39] György Dósa and Jiří Sgall. Optimal analysis of best fit bin packing. In *Automata, Languages, and Programming: 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part I 41*, pages 429–441. Springer, 2014.

[40] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. Versatile, scalable, and accurate simulation of distributed applications and platforms. *Journal of Parallel and Distributed Computing*, 74(10):2899–2917, June 2014. URL http://hal.inria.fr/hal-01017319.

[41] Rajkumar Buyya and Manzur Murshed. Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *Concurrency and computation: practice and experience*, 14(13-15):1175–1220, 2002.

[42] Dalibor Klusáček, Mehmet Soysal, and Frédéric Suter. Alea–complex job scheduling simulator. In *Parallel Processing and Applied Mathematics: 13th International Conference, PPAM 2019, Bialystok, Poland, September 8–11, 2019, Revised Selected Papers, Part II 13*, pages 217–229. Springer, 2020.

[43] Nikolay A Simakov et al. Slurm simulator: Improving slurm scheduler performance on large hpc systems by utilization of multiple controllers and node sharing. In *Proceedings of the Practice and Experience on Advanced Research Computing*, pages 1–8. 2018.

[44] Second quarter 2023 spec cpu2017 results, 2023. https://www.spec.org/cpu2017/results/res2023q2, Accessed on June 14, 2023.

[45] Sameh Sharkawi et al. Performance projection of hpc applications using spec cfp2006 benchmarks. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12. IEEE, 2009.

[46] Yu Wang, Victor Lee, Gu-Yeon Wei, and David Brooks. Predicting new workload or cpu performance by analyzing public datasets. *ACM Transactions on Architecture and Code Optimization (TACO)*, 15(4):1–21, 2019.

[47] Scott Hutchison et al. Scheduling for high performance computing with reinforcement learning. In *Proceeding of OkIP International Conference on Advances in Parallel and Distributed Computing (APDC 2024)*, 2024.

[48] Jeffrey D. Ullman. Np-complete scheduling problems. *Journal of Computer and System sciences*, 10(3):384–393, 1975.

[49] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM workshop on hot topics in networks*, pages 50–56, 2016.

[50] Qiqi Wang, Hongjie Zhang, Cheng Qu, Yu Shen, Xiaohui Liu, and Jing Li. Rlschert: An hpc job scheduler using deep reinforcement learning and remaining time prediction. *Applied Sciences*, 11(20):9448, 2021.

[51] Sisheng Liang, Zhou Yang, Fang Jin, and Yong Chen. Data centers job scheduling with deep reinforcement learning. In *Advances in Knowledge Discovery and Data Mining: 24th Pacific-Asia Conference, PAKDD 2020, Singapore, May 11–14, 2020, Proceedings, Part II 24*, pages 906–917. Springer, 2020.

[52] Di Zhang, Dong Dai, Youbiao He, Forrest Sheng Bao, and Bing Xie. Rlscheduler: an automated hpc batch job scheduler using reinforcement learning. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2020.

[53] Steve J Chapin, Walfredo Cirne, Dror G Feitelson, James Patton Jones, Scott T Leutenegger, Uwe Schwiegelshohn, Warren Smith, and David Talby. Benchmarks and standards for the evaluation of parallel job schedulers. In *Job Scheduling Strategies for Parallel Processing: IPPS/SPDP'99Workshop, JSSPP'99 San Juan, Puerto Rico, April 16, 1999 Proceedings 5*, pages 67–90. Springer, 1999.

[54] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[55] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.

[56] Ellen R Girden. *ANOVA: Repeated measures*. Number 84. sage, 1992.

[57] Student. The probable error of a mean. *Biometrika*, pages 1–25, 1908.

[58] Scott Hutchison, Daniel Andresen, and William Hsu. High performance computing scheduler parameter optimization using simulation and regression. In *Research Poster presented at Super Computing 2022 (SC22)*, 2022.

Appendix A

Permission for use of Published Works

Oklahoma International Publishing, USA
OkIP Academia & Industry Convention

2024 OkIP International Conference on Advances in Parallel and Distributed Computing (APDC)
April 1-4, 2024, Online & Oklahoma City, OK, USA

----------0----------0----------

Subject: Requesting permission for the use of published work in the dissertation

Date: 2024-04-07 13:25
From: Scott Hutchison <scotthutch@ksu.edu>
To: "info@okipublishing.com" <info@okipublishing.com>
Cc: Daniel Andresen <dan@ksu.edu>
Whom It May Concern:

I am a doctoral student at Kansas State University. I was the primary author of the work "Scheduling for High Performance Computing with Reinforcement Learning," presented at the OkIP APDC conference from 1-4 April 2024. I am writing for permission to include all of this work as a chapter in my dissertation. My dissertation will be available online through the K-State Research Exchange (http://krex.ksu.edu). In addition, my dissertation will be microfilmed by ProQuest Information and Learning, and copies of the dissertation will be available for purchase. Please supply a signed letter permitting me to use the work. You can email the permission to Scott Hutchison at scotthutch@ksu.edu. Thank you for your assistance.

Very Respectfully,
Scott Hutchison
Computer Science PhD Candidate
Kansas State University
713-504-2933
scotthutch@ksu.edu

----------0----------0----------

PERMISSION GRANTED FOR THE USE REQUESTED ABOVE

OkIP Conferences aim to publish and present good technical papers for discussion when learning from others about the latest work and developments in the field. On behalf of OkIP and the conference organizers, we thank you for the quality of your presentation and look forward to your participation at our next event.

Sincerely,

Prof. Dr. Pierre F. Tiako
OkIP Conferences General Chair
tiako@ieee.org | +1 405 413-4145
April 08, 2024

1911 Linwood Blvd Ste 100
Oklahoma City Oklahoma 73106, USA

+1 405-510-6431
info@okipublishing.com | www.okipublishing.com