Establishing a Generalizable Framework for Generating Cost-Aware Training Data and Building Unique Context-Aware Walltime Prediction Regression Models

Swathi Vallabhajosyula
Computer Science and Engineering
The Ohio State University
Columbus, Ohio, USA
vallabhajosyula.2@buckeyemail.osu.edu

Rajiv Ramnath
Computer Science and Engineering
The Ohio State University
Columbus, Ohio, USA
ramnath.6@osu.edu

***Abstract*—This paper describes a generalizable framework for creating context-aware wall-time prediction models for HPC applications. This framework: (a) cost-effectively generates comprehensive application-specific training data, (b) provides an application-independent machine learning pipeline that trains different regression models over the training datasets, and (c) establishes context-aware selection criteria for model selection. We explain how most of the training data can be generated on commodity or contention-free cyberinfrastructure and how the predictive models can be scaled to the production environment with the help of a limited number of resource-intensive generated runs (we show almost seven-fold cost reductions along with better performance). Our machine learning pipeline does feature transformation, and dimensionality reduction, then reduces sampling bias induced by data imbalance. Our context-aware model selection algorithm chooses the most appropriate regression model for a given target application that reduces the number of underpredictions while minimizing overestimation errors.**

***Index Terms*—AI4CI, Data Science Workflow, Custom ML Models, HPC, Data Generation, Scheduling, Resource Estimations**

I. Introduction

Problem Statement: Access to high-performance computing (HPC) cyber-infrastructures (CI) like Ohio Supercomputer Center[1] (OSU), Texas Advanced Computing Center[2] (TACC), or Amazon Web Services has opened up venues for researchers/students/employees across different domains to participate in state-of-the-art research to develop computationally intensive scientific workflows like such as genome sequencing[3], weather forecasting[4], and high resolution-image processing at scale. These users do not understand the shared cyberspaces and the resource allocation policies. They build their models on commodity environments, roughly estimate the resource allocations for HPC environments based on their experience, and submit jobs with default, make-do, or workable allocation requests. Users often overestimate the resource requirements, especially when there is no history of executions, leading to longer wait times for resource allocations. On the other hand, underestimations can schedule the jobs sooner, but if the application does not execute till completion within the allocation, it gets forcefully terminated. The user ends up re-submitting the job, but this time with a high overestimation, so it executes till completion. The users are also billed for these utilizations (even the terminated jobs). This work analyzes the job scripts [18] or execution history and input features [2] to estimate an accurate walltime. The HPC clusters get frequent upgrades, and new software/hardware components are deployed to make them faster. This tips off the prior user estimations made on older environments. Hence we need a robust framework that can capture the resource requirements of a user application and predict its execution time given the context[5] to avoid underpredictions and reduce overestimations [16]. This requirement leads us to the first challenge, "How can we build a prediction model without sufficient training data?". We propose to build a pipeline that automatically generates training data given to a user (target) application by profiling its executions with different configurations to build a walltime estimator.

Training Data Requirements: The execution time of a target application depends on the execution environment, input data characteristics, and application configurations. We broadly classify them as "application-specific" and "environment-specific" features. In this work [1], the authors try to estimate the execution time by instrumenting the code against varying input and profiling the features (like no. of loops or branches). We treat the application as a "black box"

This work was partially supported by the National Science Foundation and the NSF AI institute for Intelligent Cyberinfrastructure with Computational Learning in the Environment(ICICLE) under grant agreements OAC-1945347 and OAC-2112606.

[1]https://www.osc.edu/

[2]http://www.tacc.utexas.edu

[3]https://www.cdc.gov/pulsenet/pathogens/wgs.html

[4]https://www.olcf.ornl.gov/2019/05/07/mapping-climate-patterns/

[5]An application execution context includes its configurations, input characteristics, and the execution environment

and profile its characteristics by executing it with varying application configurations and resource allocations (extrinsic features). The task of generating data draws our attention to the following challenges: "How many samples should we generate before training a regression model that can accurately predict walltime?", "How to make these models robust to changes in environment or application configuration?" and "How efficiently can we generate these training samples without incurring high data generation costs?". We propose to build a "context-aware" data generation module that samples the extrinsic feature with the help of a human in the loop. We distribute the executions such that we generate more scaled-down and a few full-scale runs to reduce the overall generation time and increase the sample space.

Tractability: Building a custom regression model for every target application is expensive and time-consuming. In our previous work [19], we explored the tractability of the initial framework by testing the application-specific regression models trained on data gathered from different execution environments. Applications in a family share the features but differ in their distributions. We explored the feasibility of re-using the pre-trained models to estimate the walltime of a new application in the family. With the growing popularity of the applicability of Artificial Neural Networks (ANNs) across domains, many scientific workflows are leaning towards incorporating these models into their frameworks. So the workload from various AI/ML models has tremendously increased in the past few years [8]. The CIs have reserved nodes customized to work efficiently with DNN models and frameworks. Building transferable models to predict execution times for the family of applications in ANNs can improve the efficiency of our framework. However, we fall back to creating application-specific training runs and models when applications do not share the feature space. Applications can also differ in resource utilization and have diverse configurations that make their walltime feature space unique. We need to select a suitable regression model w.r.t. the training data that adapts to the nature of its characteristics. We propose the next module of our framework, "Building Regression Models," for the target application that follows through a traditional data-science pipeline to prepare the training data and build regression modules.

Model Selection: In our previous work [19], we manually profiled two unique regression models against three different applications and their feature spaces to observe their applicability. In this work, a model ($model_t$) with the best accuracy for environment-specific runs was identified at specific points of new data availability. As new environment-specific executions become available, we re-trained these models and picked the new best model ($model_{t+1}$). In this paper, we how how to automate this process to add diverse regression models into the selection pool and select the best application-specific model *for a chosen policy* that filters out less useful models at each step.

Model Evaluation: Regression models learn to estimate target values by being trained against standard loss functions such as mean square error. These standard loss functions are indifferent to negative and positive predictions and only aim to get the loss as close to zero as possible. That is, a slight underprediction is also considered an accurate estimation. This approach, however, does not fit our goals because an underprediction of walltime will terminate the run, incurring the sunk cost of execution as well as the added cost of re-execution. Therefore, we need new metrics for evaluating the performance of walltime prediction models i.e. those that treat the direction of the errors differently. In other words, *we aim to reduce underpredictions while still minimizing overestimation errors.* To this end we present a "**policy**" that compares the regression models against this requirement when we evaluate their prediction performance.

In our prior work [19], we demonstrated the feasibility of "democratizing" an automated training data generation pipeline that can easily accommodate new target applications. We demonstrated the generalizability of the predictions with changes in the computational environment by using a training approach that mixes scaled-down and limited full-scale training. This paper proposes a more streamlined pipeline to explore the entire data science life-cycle for generating, analyzing, pre-processing, building and, finally, using these prediction models. In particular, we present model selection criteria to pick the best regression model from the candidate pool given a target application and the policy it must operate in. This research pipeline has three modules with unique functionality and concrete goals :

1) The "**Generating and Preparing Training Data**" module automatically and systematically generates comprehensive, diverse "scaled-down," along with limited, selective "full-scale" runs with minimal human intervention.
2) The "**Building Regression Models**" module standardizes and prepares the data, trains the selected regression models with the appropriate hyper-parameters, and stores them for later use.
3) The "**Selecting Appropriate Prediction Model**" module selects the most appropriate model from a pool of pre-trained models for a given policy.

Our work simultaneously explores the three dimensions influencing execution time -execution environment, application hyperparameters, and input data. Similar to Ernest [20], our work explores an application-independent approach to predict the execution time based on the history of executions. Unlike Ernest, we build only one regression model for different execution environments by characterizing their compute and memory features. To limit the control features, we explored the applications whose execution times are minimally influenced by the distribution of input samples, like Trimmomatic or unaffected by them, like training individual DNN models, unlike applications like CameraTraps [21] where the resolution and distribution of images would influence the training time. Our previous work [19] showed that existing models within a family could be used to make predictions for similar models in

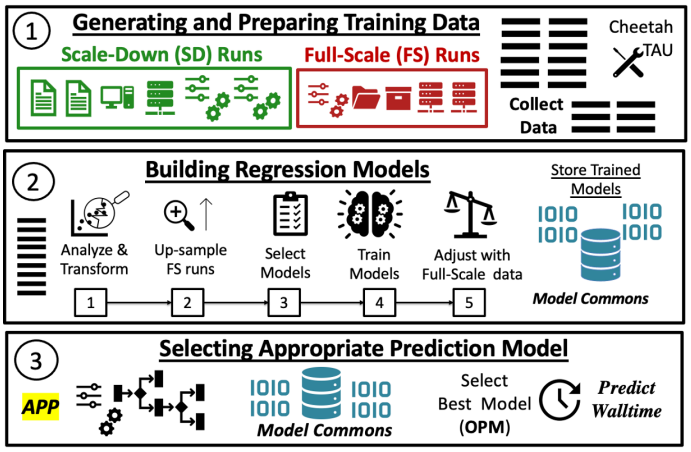

Fig. 1. Visualising the Modules in our Research Pipeline applied per Target Application

a family (like training on VGG16 and predicting for Resnet50). However, suppose the network's architecture has considerable changes; we need to treat the model as a unique application, generate scaled-down and full-scale data, and build customized predictive models.

Note that when submitting batch jobs, the (requested) wall-time refers to the maximum amount of resource allocation time needed for execution. Throughout this work, we use the terms walltime and execution terms interchangeably. The datasets and code could be found at GitHub[6]

II. Targeted Applications and Execution Environments

This section describes the characteristics of the three targeted applications and their execution environments.

A. Targeted Applications

These three applications were selected in order to represent different points in the HPC application spectrum, with respect to the intensity of their compute, input-output (IO), and memory requirements.

1) **Gray-Scott** [5] is a chemical diffusion simulator based on Gray Scott equations. It takes the diffusion coefficient of two reacting chemical substances, their densities, feed/kill rate, and the number of time steps in the reaction to simulate diffusion using a 3D array. This simulator initializes and manipulates the array to imitate the diffusion over a large range of time steps, and is therefore *compute-intensive*.
2) **Trimmomatic**: Pre-cleaning is essential in genome sequencing and assembly to achieve higher-quality assemblies. It is a popular "pre-cleaning" tool used to remove impurities from Illumina [6] genome sequencing short reads by trimming/removing the "noisy" bases that do not meet quality requirements while retaining the higher quality bases. Trimmomatic reads a set of short reads, processes them, and writes them back to the file. This application is therefore both *compute and I/O-intensive*.
3) **Family of DNNs**: We target three large sequential Convolutional Neural Network (CNN) models (each with a few million tuneable parameters) provided by Keras API on the TensorFlow Framework [9] for image classification (VGG16 [10], ResNet50 [11], and InceptionV3 [12]). For each of these models we pre-load the training data as red, green, and blue (RGB) values into memory before initializing and training them. These applications are therefore both *compute and memory intensive*.

B. Execution environments

Table I describes the execution environments (Linux workstation and OSC [17]) used for the generation of training data.

TABLE I
Execution environments for data generation

	Description
Linux workstation	2.3 GHz 8-Core Intel Core i9 with 16 GB 2400 MHz
Owens (CPU):	Dell PowerEdge C6320 two-socket servers with Intel Xeon E5-2680 v4 (Broadwell, 14 cores, 2.40GHz) 28 processors, 128GB mem
Owens (GPU)	Dual Intel Xeon 6148s Skylakes 40 cores per node @ 2.4GHz, 192GB mem
Pitzer (CPU)	Dual Intel Xeon 8268s Cascade Lakes 48 cores per node @ 2.9GHz 192GB mem
Pitzer (GPU)	Dual Intel Xeon 8268s - Dual NVIDIA Volta V100 w/32GB GPU memory 48 cores per node @ 2.9GHz, 384GB mem

III. Methodology

In this section, we describe the three modules of our research pipeline (Figure 1) viz. Generating and Preparing Training Data, Building Regression Models, and Selecting Appropriate Prediction Model. We profile three kinds of applications, as noted in Section II. Given an application and its input, the framework automatically profiles it in different execution contexts [7]. This training data then undergoes feature engineering, data preparation, prediction model building and model selection. We describes these steps next.

The execution time of an application depends on "application-specific" configurations such as the hyperparameters (e.g. the batch size, the number of training epochs, optimizer etc.) and input data characteristics (such as its size). It also depends on "environment-specific" characteristics such as the configurations of the compute nodes (e.g. GPU/CPU, no. of cores, memory, and IO bandwidth). We also capture features from the application's run-time "profiling", such as the max read/write bytes and bandwidth consumed. To build a machine learning model that accurately predicts an application's execution time, we need "sufficient" training data with a good distribution of these features. We run the targeted application against different inputs, hyperparameters, and environments to generate training data over a reasonable distribution over user-defined configurations. Note that these configurations consist of extrinsic parameters, i.e., parameters

[6]https://github.com/manikyaswathi/PreditingExecutionTime

[7]system configurations, inputs, and application hyperparameters

that change the application's behavior (execution time) without visibility into or modifying its implementation. A regression model must learn to predict continuous values (in this case, walltime) from these features. We captured 29 features for Gray-Scott, 26 features for Trimmomatic, and 16 features for our family of DNNs. Figure 2 shows the features influencing the execution time for each targeted application. Note that the feature space comprises both numerical and categorical values.

A. Research Pipeline

We use the following automation tools to execute and profile the target application against a set of configurations:

- **The Cheetah Experiment Harness and Campaign Management System** (Cheetah[8]) [3] enables running a target application with different application and system configurations. Configurations are specified in a campaign file for each job. Cheetah's runtime module, Savannah, converts the meta-data in the campaign files into execution configurations ("experiments") and runs these experiments on the target systems. Each experiment/run has a unique workspace endpoint, where it writes the experiment execution script and its configurations, experiment information logs, profile traces, and outputs. Cheetah also has a post-processing script that can be executed after each experiment to collect features from the workspace endpoint.
- **Tuning and Analysis Utilities** (TAU[9]) [4] is used within Cheetah to profile the runtime features that capture the performance-related information of a target application. We use the TAU command-line argument '-io' to observe the port and memory-mapped IO read/write bandwidths along with the maximum bytes written/read per execution. We also used Pytools[10] to capture static profiling information such as processor cores, memory, and other environment-related configurations.

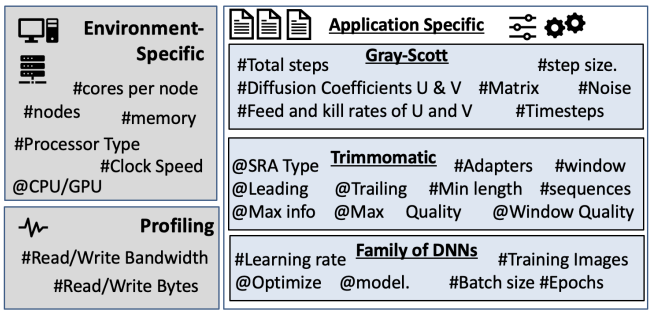

Fig. 2. Features used to predict walltime. # marks numerical values and @-marks categorical values

1) *Generating and Preparing Training Data:*

a) Feature Engineering: The primary goal of this module is to generate enough training data for our regressions models to predict appropriate execution time for a given an application context. Executing resource-intensive applications on shared, production research cyberinfrastructures (CI) to generate large amounts of training data is expensive in terms of resources, cost and time, especially when the application context keeps changing causing the need to regenerate the training data. To reduce the cost of data generation, we studied the feasibility of generating training samples on commodity environments like workstations or on low contention resources like single-node allocations. To reduce the execution time of the runs, we limited the input configurations and application features that influence execution time. For example, while training DNNs, we reduced the number of images in the training data. We call these runs "**scaled-down**" (SD) runs. We also ran the target applications for a few iterations at "**full-scale**" (FS) on every new CI to capture CI-specific information in training samples. We then ran and profiled the targeted application against a set of pre-identified configurations for both SD and FS runs. The SD executions cover more feature distributions and over several iterations (to exclude errors) and the FS executions are intended to capture the CI-specific characteristics. Graph 3 shows the estimated cost of generating the entire training data with FS configurations[11] vs. the actual cost of generating the combination of SD and FS runs. In III-A2, we generate different regression models with this combination of SD and FS training data.

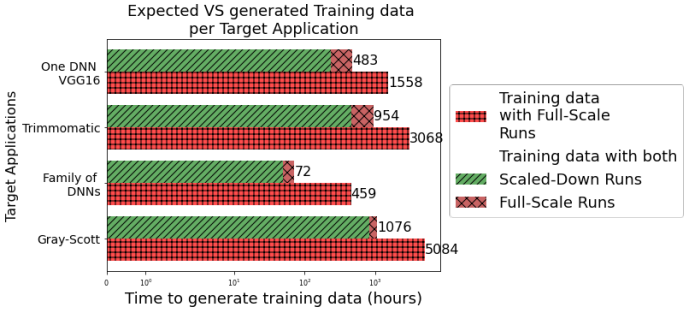

Fig. 3. The cost for preparing the training data with sufficient "full-scale" (FS) runs VS a combination of sufficient "scaled-down" (SD) and "full-scale" runs

An overview of the "extrinsic" control features used in generating training data is shown below[12]:

1) **Gray Scott**: We ran this simulator with different array dimensions, diffusion coefficients of the reaching chemicals, feed/kill rate, and number of steps, in different environments with different numbers of processor allocations. The SD and FS runs also differ in the array dimensions and in the time steps.
2) **Trimmomatic**: We use six different genome sequences as input "short-reads". A unique trim quality can be obtained by changing the leading and training quality, minimum quality per window size, strictness, and minimum length. A high-quality genome takes lesser time to be pre-processed than a low-quality one to achieve the

[8]https://github.com/CODARcode/cheetah
[9]https://www.cs.uoregon.edu/research/tau/home.php
[10]https://github.com/inducer/pytools

[11]estimated as (no. of scaled-down samples)*(avg full-scale execution time)
[12]we used more application specific configurations than those listed here

same trim configurations. The FS and SD run differ in the size of the genome sequence, the number of threads and configurations of the execution environments.

3) **Family of DNNs**: We train the three CNN models with different numbers of training samples and batch sizes. We run them on CPU and GPU nodes with different core allocations and observe the execution times as average epoch time. The SD and FS runs differ in the number of training samples, batch size, epochs, and number of cores used. FS runs are obtained by training the models on the entire dataset and larger batch sizes on all cores of a single-node CPU (or a GPU) HPC allocation.

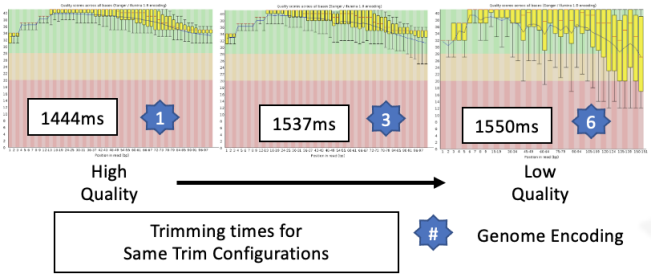

Fig. 4. Per Base Sequence Quality visualization using "fastqc" - Trimming time with constant configurations

Each experiment leaves its output traces in its workspace folder. A post-processing script collects the training data entry per experiment by parsing the different files in the workspace. Job scripts, input metadata, and logs have application-specific feature values; TAU profile logs provide runtime I/O values, and the CI database or the allocation history provides the values of the environment-specific features. Finally, we collect all the training data entries and store them as a ".csv" file that is used in the "Building Regression Models" component.

*2) **Building Regression Models**:* We generated training data that could be used to estimate walltime. This data is noisy and needs to be carefully cleaned/transformed to be usable by ML models. Data-Science workflow defines a set of rules are crafted to prepare the data after visualizing and analyzing it to identify noise and interesting patterns(features). Models trained on this data are analyzed, and data is adjusted or reprocessed to get better results. We replicate our pipeline to automate this process in two pieces - preparing the data and training models and selecting the best model by analyzing the accuracies (metrics). This section provides an overview of the selected regression models and evaluation metrics and describes the machine learning components used to prepare the generated data for training.

Several off-the-shelf regression models (Table II) may be trained to predict execution time. However, the choice of the regression model depends on the characteristics of the training data i.e. the types of its features, feature correlations, the distribution of the data (i.e. the ratio of SD and FS runs), and the model's sensitivity to outliers. We split the FS datasets into training, validation, and test sets in a 50:25:25 ratio. The FS training data is used along with SD samples to make predictions for the FS runs. We use a validation dataset consisting of only FS runs to adjust the predictions with respect to the FS feature distributions. Graph 5 (without upsampling) shows the difference in the distribution of execution times for SD and FS runs for the three applications. Given the nature of these datasets(SD-FS sample imbalance, availability of additional FS runs, feature types), training a single off-the-shelf regression model cannot accurately predict execution time (predictions with fewer errors) for all applications. Table II shows the sensitivity of regression models with respect to multicollinearity of features (i.e. features that are strongly correlated), outliers, categorical features, and linear separability of features for the Gray-Scott application. The machine learning pipeline pre-processes the training data, removes the sampling bias (through upsampling - see following section), and trains and stores a selected set of regression models over different training data distributions.

We use the following metrics to score the prediction performance of each regression model:

1) **Under Prediction Percentage (UPP)**: percentage of test samples that have predicted value (walltime) less than actual value (execution time).
2) **Mean Absolute Error (MAE)**: Mean of absolute errors between the predicted value (walltime) and the actual value (execution time). Outliers do not drastically affect this error.
3) **Mean Square Error (MSE)**: Mean of absolute errors between the predicted value (walltime) and the actual value (execution time). A single outlier can drastically increase this error. We compute Root Mean Square Error RMSE from the MSE.
4) **Mean Absolute Percentage Error (MAPE)**: Mean of absolute percentage errors between the predicted value (walltime) and the actual value (execution time).

The following are the components of our machine learning pipeline (Module (2) in Figure 1):

*a) **Feature Encoding & Dimensionality Reduction**:* Feature encoding converts categorical information into a numeric format. A unique genome is encoded with a number such that the genome with the highest quality gets 1, and the one with the lowest quality gets a 6 when we have six genomes in the sample space[13]. We encoded model types from VGG16, ResNet50, and InceptionV3 to 1, 2, and 3 for the Family of DNNs. Correlation analysis helps remove redundant features. By creating new independent features called principal components, Principle Component Analysis (PCA) reduces the feature dimensions and removes multicollinearity with minimal information loss. The feature space is reduced to 14 for Gray-Scott(29), 14 for Trimmomatic(24), and 9 for the Family of DNNs(16) principal components.

*b) **Up-sampling**:* From Graph 3, we intentionally limit the FS runs to save time and cost of data generation. The ratio of SD and FS runs in training data [41:1 (Gray-Scott), 51:1 (Trimmomatic), and 13:1 (Family of DNNs)] shows the

[13]we manually encoded this information by looking at the fastqc files4

TABLE II
THE LIST OF REGRESSION MODELS AND THE OBSERVED CHARACTERISTICS[a]

Regression Model	Handle Non-linearity?	Needs Huge Training Data?[b]	Sensitive to Outliers?	Handle Categorical Data?	Handle multicollinearity?	Needs Standardization?
(simple) Linear (SLR)	No	No	Yes	No	No	Opt
LASSO (LAR)[c]	No	No	Yes	No	Yes	Opt
Decision Tree (DTR)	Yes	No	Yes	Yes	Yes	No
Support Vector (SVR)	Yes	No	No	No	No	Opt
Neural Network (NNR)[d]	Yes	Yes	No	No	Yes	Yes

[a]Refer Section "Feature Encoding Dimensionality Reduction" under IV-B1 to see the models behaviour with respect to input data characteristics III
[b]Huge training data is needed to train DNNs.
[c]Linear Model trained with L1 prior as regularizer
[d]One Hidden Layer Neural Network

imbalance of this sample space. This training and FS test datasets have different distributions ((a) in Graph 5). Some regression models could treat these values as outliers and ignore them. Up-sampling FS runs in training to match the SD runs eliminates this sampling bias. Graph 5 (b) shows a more similar distribution between training and test data after upsampling.

*c) **Select Unique Regression Models** Given a list of available off-the-shelf regression models:* (Table II), we select only those models that exhibit unique behavior with respect to the predictions in terms of the evaluation metrics. After applying PCA and upsampling, LASSO and RIDGE regression model has the same accuracies as a simple Linear Regression model and could be removed from the candidate regression models (RMs). We finetune the model's hyperparameters by analyzing the bias-variance tradeoff using cross-validation over the training data.

*d) **Train Regression Models (RMs) on Different Training Data (TDs):*** We train each regression model on three subsets of generated training data to study the influence of the limited number of full-scale runs. The first training data set has only "scaled-down" (SD) runs. Then we add a few "full-scale" (FS) samples from the FS Training data (25%) after upsampling. Then we add more "full-scale" (FS) samples (50%) to training data after upsampling. Training models on these datasets caliber model's performance against unseen test datasets, i.e., FS Test data when the training samples have no (SD) to limited exposure (SD+50%FS) to Test data distribution. This study tests the model's scalability to new and unseen execution environments. It helps us answer another question, "how frequently should we add the available full-scale runs to training data?". For example, if a model predicts better than the previous model (built on SD+25%FS data) by adding more FS samples (going from 25%FS to 50%FS), then the newly added FS runs add information to our model. However, if the previous model's predictions are better than these predictions, the added FS samples are inducing noise and are not helpful at this point. The model should wait for the availability of more FS runs before retraining the model. We validate the models against the FS validation dataset.

*e) **Validation Adjustment (VA=xAF):*** Upsampling FS training samples reduces the prediction errors by eliminating the oversampling bias from the SD runs. However, a model can still scale w.r.t SD distributions (50% of upsampled training comes from SD runs). We need to scale the predictions to the FS distribution further to reduce the potential underpredictions. Validation data[14] is used to obtain the adjustment factor, which is the MAPE of the validation data underpredictions. This Adjustment Factor (AF) increases the FS test data predictions to reduce underpredictions potentially. AF (as shown in (1)) always upscales all the FS test predictions (2), increasing the MAE. MSE and MAPE, especially for the overestimations. If adjusting predictions is not significantly reducing the underpredictions but increases these overestimation errors for a model, then we do not adjust its predictions.

$$AF = 1 + MAPE(y, \hat{y}), \forall \hat{y} < y \tag{1}$$

$$new\ \hat{y} = \hat{y} * AF \tag{2}$$

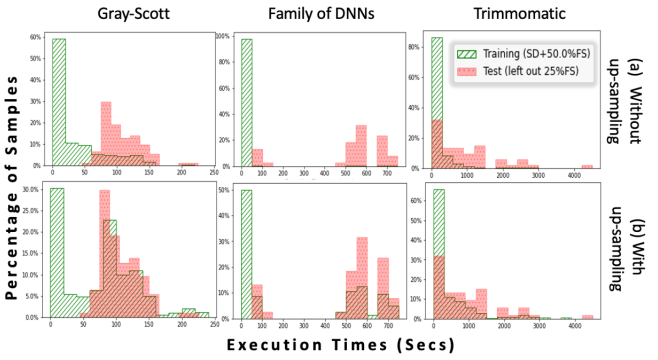

Fig. 5. Distribution of execution times for the scaled-down data (training) and full-scale data (test) without and with upsampling

We run all the selected regression models RMs={SLR, NNR, DTR, SVR} against the different training datasets TDs={SD, SD+25%FS, SD+50%FS} with and without validation adjustments (**VA**)={YES, NO} for an application and store the models and their metrics as "**model commons**".

*3) **Selecting an Appropriate Prediction Model:*** We have a set of candidate models that can predict the execution time for a given application. Using the FS validation dataset, we pick the best model using the "policy" (as shown in Figure6). We call this Optimal Predicted Model (OPM).

When we need to estimate the execution time for an application, provided the target environment and input configurations (inference data), this module looks through the existing pre-trained candidate models (TMs) for this application. Figure 7

[14]has the same distribution as test data

TABLE III
CHARACTERISTICS OF DIFFERENT COMBINATIONS OF "SCALED-DOWN" AND "FULL-SCALE" GENERATED SAMPLES WHEN USED AS TRAINING AND TEST SETS.

	Train & Test			
	FS & FS	**SD & SD**	**SD & FS**	**SD+n%FS & FS**
No PCA - "Raw" Generated Training Datasets	-limited training data -multicollinearity exists - limited distribution of samples over feature space	-no feature standardization -multicollinearity exists	-unseen data -no feature standardization - multicollinearity exists -training and test data have different distribution	*-still has unseen data*[b] *-no feature standardization* *-multicollinearity exists* *-imbalanced SD and FS training* *-training and test data have different distribution*
Features as Principle Components	-limited training data - limited distribution of samples over feature space	**IDEAL**[b] **Training and Test Data**	-unseen data -training and test data have different distribution	***-still has unseen data*** ***-imbalanced SD and FS training*** ***-training and test data have different distribution***
Principle Components + Training Data Outliers	-limited training data -outliers exist - limited distribution of samples over feature space	-outliers exist	-unseen data -outliers exist -training and test data have different distribution	***-still has unseen data*** ***-outliers exist*** ***-imbalanced SD and FS training*** ***training and test data have different distribution***

[a] This is the training data generated through our pipeline, and we want to train highly accurate regression models on this.
[b] The ideal characteristics for training data. We process the training data intending to achieve some of their characteristics.
SD and FS samples have different distributions for some environment-specific features and input-features.
SD data represents "huge" training data with good sample space. FS data has "limited" sample distributions over feature-space.

visualizes this model selection pipeline. This would replicate the model analysis part of the data-science lifecycle and check if the components added to address the noise induced by FS data (%FS inclusion, xAF) are helpful with respect to the validation data.

$$Better\ Model = \begin{cases} A, & MAE(A) < MAE(B) \cap \\ & MSE(A) < MSE(B) \cap \\ & MAPE(B) < MAPE(B) \\ B, & otherwise \end{cases}$$

(a) Default Criteria (DC)

$$Better\ Model = \begin{cases} A, & MAE_{change}(B,A) < w * UPP_{change}(A,B) \cap \\ & RMSE_{change}(B,A) < w * UPP_{change}(A,B) \cap \\ & MAPE_{change}(B,A) < w * UPP_{change}(A,B) \\ B, & otherwise \end{cases}$$

(b) Selection Criteria (SC)

Where: $w = 1$

$$(A,B) = \begin{cases} (TM1, TM2), & UPP(TM1) < UPP(TM2) \\ (TM2, TM1), & otherwise. \end{cases}$$

Fig. 6. The Default and Selection Criteria for a defined "Policy" for comparing two pre-trained models (TM1, TM2) selected from model commons to fetch a better model

Filter Components: To select the best performing model per application from the "model commons", we need to follow through an order of decisions w.r.t validation adjustment (xAF), availability of full-scale training runs, and finally, the regression models.

1) **Filter 1- Validation Adjustment Check:** This tests if "Adjusting predictions w.r.t. validation data is Better than No Adjustment?"
2) **Filter 2 - FS Training Runs Inclusion Check:** This tests if "Adding new FS samples to training data is Better than not adding them?"
3) **Filter 3 - Select Better Regression Model:** We know which training dataset works better, along with the adjustment factor (xAF) for each model. Select the model with the best accuracy.

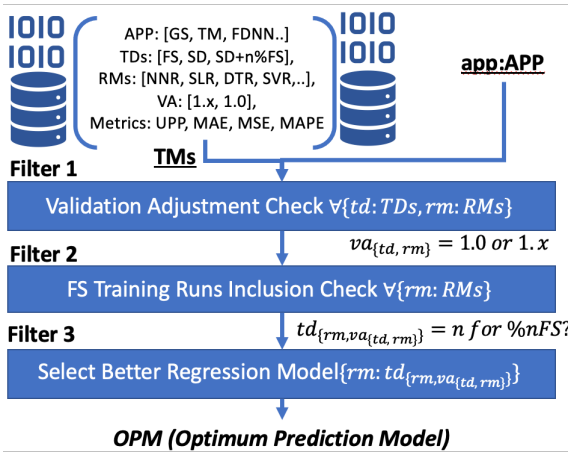

Fig. 7. The Per Application Model Selection Pipeline which selects the Optimal Prediction Model (OPM) by using "policy (Fig 6)" and validating the choices made in during Building Regression Models module.

The "default" policy 6 criteria could be used, or an environment-specific policy could be configured. The current policy has two comparison criteria w.r.t the scores generated on the (FS) validation dataset:

- If one model has fewer underpredictions and better overestimations (lower errors), select this model as the "Better Mode"(DC) (Figure 6**(a)**).
- If one model has a lower no. of underpredictions, and the other has more accurate predictions, apply (SC) and select the "Better Model" Figure 6 **(b)**.

The given "policy" 6 tolerates overestimation errors of the selected model as long as the reduction in underpredictions is higher than the other model. When comparing two models (TM1, TM2) using SC, we pick a model with lower UPP if

TABLE IV
THIS TABLE COMPARES THE ACCURACY OF THE SELECTED REGRESSION MODELSII WITH RESPECT TO THE TYPE OF TRAINING AND TEST DATA SETS

	Train & Test	FS & FS			SD & SD			SD & FS		
	RM	**MAE**	**MSE**	**MAPE**	**MAE**	**MSE**	**MAPE**	**MAE**	**MSE**	**MAPE**
Raw Training Data**	**NNR**	*18.22*	*513.00*	*18.76*	*14309.90*	*23197633*	*317985.47*	*125.65*	*16381.18*	*123.35*
	DTR	**2.18**	**35.06**	**2.07**	**1.53**	**18.67**	**7.72**	**49.19**	**3301.52**	**46.85**
	SLR	10.05	208.58	10.69	15.00	379.00	244.73	27.51	1095.06	24.97
	LAS	9.52	198.77	10.07	15.03	380.16	244.74	27.22	1078.22	24.67
Processed Training Data (Principle Components)	**NNR**	8.574	134.68	8.74	**3.91**	**50.92**	**32.00**	**11.99**	**286.02**	**12.39**
	DTR	**3.09**	**58.14**	**3.22**	**5.13**	**171.90**	**43.09**	50.54	3085.47	48.86
	SLR*	10.05	208.58	10.69	**15.51**	**409.35**	**252.09**	20.54	721.49	18.24
	LAS*	10.05	208.45	10.69	**15.51**	**409.35**	**252.05**	20.54	721.66	18.25
	SVR	9.65	209.77	10.17	**14.53**	**470.64**	**208.80**	30.81	1283.43	28.11
Principle Components + Training Data with Outliers	**NNR**	*491.82*	*4529327.55*	*373.27*	**5.27**	**94.24**	**31.02**	**25.16**	**1444.78**	**26.47**
	DTR	**4.28**	**115.95**	**4.09**	**6.07**	**442.67**	**68.44**	*53.47*	*3620.91*	*52.09*
	SLR*	195.27	61032.37	197.77	18.00	482.75	347.06	*16.73*	*518.62*	*14.98*
	LAS*	192.72	60341.17	193.31	18.00	482.72	347.00	*16.73*	*518.75*	*14.99*
	SVR	11.39	249.09	12.30	14.61	441.88	217.17	27.47	1084.32	24.83

Table III shows the characteristics of these "raw" datasets and the processed training data - principle components (PCs)
** Could not converge while training SVRs on "raw" training data.
* After standardizing the features, SLR and LASSO seem to have similar accuracies specially with large no of training samples.

the decrease in the no. of predictions is not increasing the overestimation errors drastically. In other terms, if the change in underprediction percentage from TM1 to TM2 is more significant than the change in overestimation errors (MSE, MAE, and MAPE) from TM2 to TM1.

IV. ANALYZING THE GENERATED TRAINING DATA AND COMPONENTS OF THE RESEARCH PIPELINE

This section analyzes the generated training data against the selected regression models with different sampling configurations. This analysis exposes the importance of applying the machine learning components used in the Build Regression Models. Then we show how the Selecting Appropriate Prediction Model works to fetch the Optimum Prediction Model (OPM) for each target application.

A. Analyzing the Generated Training Data

Table III shows the configurations and characteristics of "scaled-down"(SD) and "full-scale"(FS) runs used for training and testing datasets. We train the selected regression models II on these configurations to understand their applicability. See the section IV-B1 "Feature Encoding and Dimensionality Reduction" below for more details.

B. Analysing Build Regression Models

1) Feature Encoding Dimensionality Reduction: The following are the joint observations made on the applicability of selected regression models before and after applying feature encoding and dimensionality reduction (Tables III and IV):

- SLR models are sensitive to outliers in small training datasets (FSFS with Principle Components+Outliers) and cannot handle the sampling bias with SD+50%FS training data. LAS and SLR models have similar error patterns, especially after applying PCA. LAS uses L1 regularization over SLR to avoid overfitting the model to training data. But it still cannot make predictions for unseen test data (SD&FS)
- SVR models cannot handle multicollinearity. SVRs are sensitive to unseen test samples (SD&FS). They are not sensitive to outliers even when we have limited training data(FS&FS with Principle Components+Training Data Outliers). So they treat FS samples in SD+50%FS training data as outliers and are ignored. They do not learn anything about the FS runs and do not scale to FS test data.
- DTR models can handle multicollinear non-standardized data("raw" training data). They work well with small training datasets (FS&FS) but are sensitive to outliers in training data (Principle Components+Outliers in training). They fail to make predictions for unseen data(SD&FS). As long as they see few samples in distribution, they fair reasonably well with the sample imbalance problem(SD+50%FS in training data).
- NNR do not scale better when we have extremely limited data (FS) and outliers influence the model when there are no enough samples to learn from. There work really well with SD data with ample training data and these models scale relatively well for under data.

2) Up-sampling: Table V shows that Up-sampling reduces regression errors (MAE MSE, MAPE) for all the models across applications. It also tries to reduce the no. of underpredictions (UPP). We apply upsampling to the training data with FS runs for all the subsequent experiments.

3) Selecting Unique Regression Models: By looking at the previous experiments Table IV and Table V and observations (in the section "Feature Encoding Dimensionality Reduction"), we see that Linear, RIDGE[15], and LASSO models have similar behavior when training or large sample or on the pre-processed dataset. Reducing the models in selection spaces reduces the time complexity for training these models, subsequently picking the optimal model in the "Selecting Appropriate Prediction Model" phase. Similarly, when new

[15]Omitted RIDGE scores from tables to save space

TABLE V
TABLE 3: PERCENTAGE REDUCTION IN MAE, MSE, MAPE AND UPP AFTER UP-SAMPLING FS RUNS. TRAINING DATA = SD+50%FS

		Percentage Change after Up-sampling (rounded to nearest int)			
Apllicartion	**RM**	**MAE**	**MSE**	**MAPE**	**UPP**
Gray-Scott	NNR	-64	-76	-54	-5
	DTR	-11	-16	-7	**34**
	SLR	-26	-52	-13	-64
	SVR	-54	-66	-52	-46
Trimmomatic	NNR	-43	-66	-50	-11
	DTR	0	16	0	-13
	SLR	-55	-77	-50	55
	SVR	-65	-86	-74	-13
Family of DNNs	NNR	-77	-90	-82	33
	DTR	-28	-52	-25	**4**
	SLR	-20	-32	-49	-16
	SVR	-20	-20	-70	-12

models are added to the selection pool, we can train and analyze them on different sampling configurations and eliminate redundant models.

4) Validation Adjustment: This adjustment aims to reduce the no. of underpredictions by multiplying the prediction $\hat{y}$ with xAF ($1.x$). Increasing the predictions forces the model to overestimate and increase the positive errors (MSE, MAPE, and MAPE). We observe this change in Table X; we see that the adjustment factors lead to a positive change (increase) in the mean errors for overpredictions and reduce the underpredictions (percentages).

From now on, we report the MAE, MSE, and MAPE on only overpredictions along with UPP, as our goal is to reduce underprediction and minimize overestimation errors.

TABLE VI
THE TABLE SHOWS THE SELECTING APPROPRIATE PREDICTION MODEL PER TARGET APPLICATION ALONG WITH THE INTERMEDIATE SELECTIONS USING DEFAULT POLICY

SD+		Gray-Scott			Trimmomatic			Family of DNNs		
FS	VA	F1	F2	F3	F1	F2	F3	F1	F2	F3
NNR										
NO	Y	O	-	-	O	-	-	O	-	-
25%	N	-	-	-	O	-	-	-	-	-
	Y	O	-	-	-	-	-	O	-	-
50%	N	-	-	-	O	O	-	-	-	-
	Y	O	-	-	-	-	-	O	O	**O**
DTR										
SD	Y	O	-	-	O	-	-	O	-	-
25%	N	O	O	-	-	-	-	-	-	-
	Y	-	-	-	O	O	-	O	-	-
50%	N	O	O	**O**	-	-	-	-	-	-
	Y	-	-	-	O	-	-	O	O	-

*removing SLR and SVR from the list to save so cannot see results or Trimmomatic, Training Data = SD+n%FS
Adjusted w.r.t. Validation Y=Yes and N=NO

C. Optimal Prediction Model Selection

Table VI traces the "**Selecting Appropriate Prediction Model**" filtering steps to select an optimum prediction model per application.

After selecting the optimal model for a targeted application, we compare the accuracies with the baselines. The baseline model - Selected Baseline Model(SMB): Train all "selected" regression models on 50%FS training runs and evaluate w.r.t. FS test data. Fetch the best model w.r.t "Selection Criteria." This baseline represents the dataset, just the history of runs (FS) and no SD data availability.

Table VII shows that our model performs better than the the baseline w.r.t "**policy**" for all the three applications. Each application has a different regression model fetching optimal scores with different training data configurations (TDS).

Table VIII *shows the influence of the policy over the selection of OPM in the given context*. For example, if we want our model to be sensitive towards errors in overpredictions (MAPE_change $<= 10$), models with no validation adjustment (VA) were chosen as better models. For Trimmomatic and Family of DNNs, the framework picks models without validation adjustment (no xAF). Moreover, this behavior is as expected when we have FS samples (upsampled) in the training data. However, when we have no FS runs in training, like in Gray-Scott, the framework chooses a model that adjusts the predictions to match the distribution of FS test data. We also observe that the framework is sensitive to the FS samples added to the training data with respect to the selected policy. The regression models trained on fewer FS samples (training data SD+25%FS) were selected as better models (Trimmomatic for MAPE_change $<= 10$) even though we have an additional 25% FS samples available. Since by adding these samples to the training, the model tends to overestimate, causing the constraint MAPE_change $<= 10$ to fail, our framework chooses to use an older pre-trained model than its re-trained version.

TABLE VII
COMPARING THE OPTIMUM PREDICTION MODEL (OPM) TRAINED AND SELECTED BY OUR PIPELINE WITH THE BASELINE - SELECTED BASELINE MODEL(SMB)

APP	**APP**	**TDS**	**RM**	**VA**	**UPP**	**MAE**
Gray-Scott	**OMB**	**50%FS**	**SVR**	**No**	**38.30**	10.44 186.83 12.62
	OPM	**SD+ 50%FS**	**DTR**	**No**	**32.98**	**3.87** **120.57** **4.34**
Trimmomatic	**OMB**	**50%FS**	**SLR**	**No**	36.84	9.23 165.82 3.52
	TM	**SD+ 50%FS**	**SVR**	**Yes**	**7.89**	**34.42** **1840.32** **6.58**
Family of DNNs	**OMB**	**FS**	**DTR**	**No**	35.00	163.321 105159.259 176.672
	OPM	**SD+ 50%FS**	**NNR**	**Yes**	**15.09**	**50.359** **5294.590** **11.800**

VA - Validation Adjustment **Overestimation errors only
TM-Target Regression Model TDS-Type of training data

V. CONCLUSION AND FUTURE WORK

We demonstrate a robust pipeline that can generate training data, train a pool of selected regression models, and pick the best one to predict execution time for any given application. We propose selection criteria that compare these trained models to select one that reduces underpredictions while minimizing the overestimation. With the help of only a few validation samples representative of new execution

TABLE VIII
CHANGING THE POLICY TO SUIT THE CHARACTERISTICS OF THE EXECUTION ENVIRONMENT CHANGES THE BEST MODEL SELECTION (OPM)

	Gray-Scott			Trimmomatic			Family of DNNs		
Policy	**TDS**	**VA?**	**RM**	**TDS**	**VA?**	**RM**	**TDS**	**VA?**	**RM**
Default(w=1)	SD+50%FS	NO	DTR	SD+50%FS	YES	SVR	SD+50%FS	YES	NNR
MAPE_change <= 10	SD	YES	SVR	**SD+25%FS**	**NO**	**NNR**	**SD+50%FS**	**NO**	**NNR**
MAPE_change <= 100	SD+50%FS	NO	DTR	**SD+25%FS**	**NO**	**NNR**	**SD+25%FS**	**YES**	**NNR**
MAPE_change <= 50	SD+50%FS	NO	DTR	SD+25%FS	NO	NNR	SD+50%FS	NO	SVR

configurations for an existing application, our pipeline can dynamically pick a suitable regression model that generalizes to the new configuration.

We plan to extend the current work in the following directions:

- Deploy the pipeline and models into production use within an HPC center.
- Integrate the scheduling policies of target environments into the selection criteria to include HPC policies into the feature space, to be able to include policy consequences, such as wait times into our predictions.
- Propose a new cost function that mimics the model selection criteria as loss functions that could be directly used in a regression model.
- Evaluate our pipeline for distributed applications. For example, evaluate whether the pipeline can appropriately generate training data for distributed executions and whether it can capture the communication-related features that influence execution time.
- By analyzing the internal execution of DNNs, explore the generalization to new DNNs in a family without generating additional training data.

ACKNOWLEDGMENT

We thank our peers at Oak Ridge National Laboratory(ORNL) for providing insights about the Cheetah Framework and guiding us through the understanding and implementation. We would also like to thank our colleagues from the Evolution, Ecology, and Organismal Biology department at the Ohio State University for providing insights about the CI usage that motivated us to build this pipeline.

REFERENCES

[1] Huang, L., Jia, J., Yu, B., Chun, B., Maniatis, P. & Naik, M. Predicting Execution Time of Computer Programs Using Sparse Polynomial Regression. (2010,1)

[2] Miu, T. & Missier, P. Predicting the Execution Time of Workflow Activities Based on Their Input Features. *2012 SC Companion: High Performance Computing, Networking Storage And Analysis*. pp. 64-72 (2012)

[3] Mehta, K., Allen, B., Wolf, M., Logan, J., Suchyta, E., Choi, J., Takahashi, K., Yakushin, I., Munson, T., Foster, I. & Klasky, S. A Codesign Framework for Online Data Analysis and Reduction. *2019 IEEE/ACM Workflows In Support Of Large-Scale Science (WORKS)*. pp. 11-20 (2019)

[4] Shende, S. & Malony, A. The TAU parallel performance system. *The International Journal Of High Performance Computing Applications*. **20**, 287-311 (2006)

[5] McGough, J. & Riley, K. Pattern formation in the Gray–Scott model. *Nonlinear Analysis: Real World Applications*. 5, 105-121 (2004), https://www.sciencedirect.com/science/article/pii/S1468121803000208

[6] Bolger, A., Lohse, M. & Usadel, B. Trimmomatic: a flexible trimmer for Illumina sequence data. *Bioinformatics*. 30, 2114-2120 (2014)

[7] Li, B., Gadepally, V., Samsi, S. & Tiwari, D. Characterizing Multi-Instance GPU for Machine Learning Workloads. *2022 IEEE International Parallel And Distributed Processing Symposium Workshops (IPDPSW)*. pp. 724-731 (2022)

[8] Samsi, S., Weiss, M., Bestor, D., Li, B., Jones, M., Reuther, A., Edelman, D., Arcand, W., Byun, C., Holodnack, J. & Others The mit supercloud dataset. *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. pp. 1-8 (2021)

[9] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y. & Zheng, X. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. (2015), https://www.tensorflow.org/, Software available from tensorflow.org

[10] Simonyan, K. & Zisserman, A. Very deep convolutional networks for large-scale image recognition. *ArXiv Preprint ArXiv:1409.1556*. (2014)

[11] He, K., Zhang, X., Ren, S. & Sun, J. Deep residual learning for image recognition. *Proceedings Of The IEEE Conference On Computer Vision And Pattern Recognition*. pp. 770-778 (2016)

[12] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J. & Wojna, Z. Rethinking the inception architecture for computer vision. *Proceedings Of The IEEE Conference On Computer Vision And Pattern Recognition*. pp. 2818-2826 (2016)

[13] Klusáček, D. & Chlumský, V. Evaluating the Impact of Soft Walltimes on Job Scheduling Performance. *Job Scheduling Strategies For Parallel Processing*. pp. 15-38 (2019)

[14] Soysal, M., Berghoff, M., Klusáček, D. & Streit, A. On the Quality of Wall Time Estimates for Resource Allocation Prediction. *Proceedings Of The 48th International Conference On Parallel Processing: Workshops*. (2019), https://doi.org/10.1145/3339186.3339204

[15] Peternier, A., Binder, W., Yokokawa, A. & Chen, L. Parallelism Profiling and Wall-Time Prediction for Multi-Threaded Applications. *Proceedings Of The 4th ACM/SPEC International Conference On Performance Engineering*. pp. 211-216 (2013), https://doi.org/10.1145/2479871.2479901

[16] Klusáček, D. & Soysal, M. Walltime Prediction and Its Impact on Job Scheduling Performance and Predictability. *Job Scheduling Strategies For Parallel Processing*. pp. 127-144 (2020)

[17] Center, O. Ohio Supercomputer Center. (1987), http://osc.edu/ark:/19495/f5s1ph73

[18] Soysal, M., Berghoff, M. & Streit, A. Analysis of job metadata for enhanced wall time prediction. *Workshop On Job Scheduling Strategies For Parallel Processing*. pp. 1-14 (2018)

[19] Vallabhajosyula, M. & Ramnath, R. Towards Practical, Generalizable Machine-Learning Training Pipelines to build Regression Models for Predicting Application Resource Needs on HPC Systems. *Practice And Experience In Advanced Research Computing*. pp. 1-5 (2022)

[20] Venkataraman, S., Yang, Z., Franklin, M., Recht, B. & Stoica, I. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. *13th USENIX Symposium On Networked Systems Design And Implementation (NSDI 16)*. pp. 363-378 (2016)

[21] Beery, S., Morris, D. & Yang, S. Efficient Pipeline for Camera Trap Image Review. *ArXiv Preprint ArXiv:1907.06772*. (2019)