Resource Utilization Aware Job Scheduling to Mitigate Performance Variability

Daniel Nichols†, Aniruddha Marathe*, Kathleen Shoga*, Todd Gamblin*, Abhinav Bhatele†

†Department of Computer Science, University of Maryland, College Park, Maryland 20742 USA
*Lawrence Livermore National Laboratory, Livermore, California 94551 USA
E-mail: †dnicho@umd.edu, bhatele@cs.umd.edu, *{marathe1, shoga1, tgamblin}@llnl.gov

***Abstract*—Resource contention on high performance computing (HPC) platforms can lead to significant variation in application performance. When several jobs experience such large variations in run times, it can lead to less efficient use of system resources. It can also lead to users over-estimating their job's expected run time, which degrades the efficiency of the system scheduler. Mitigating performance variation on HPC platforms benefits end users and also enables more efficient use of system resources.**

In this paper, we present a pipeline for collecting and analyzing system and application performance data for jobs submitted over long periods of time. We use a set of machine learning (ML) models trained on this data to classify performance variation using current system counters. Additionally, we present a new resource-aware job scheduling algorithm that utilizes the ML pipeline and current system state to mitigate job variation. We evaluate our pipeline, ML models, and scheduler using various proxy applications and an actual implementation of the scheduler on an Infiniband-based fat-tree cluster.

***Index Terms*—performance variability, data analytics, machine learning, prediction models, scheduling**

I. Introduction

Performance variability has become a significant problem for end users, especially as high performance computing (HPC) systems grow in scale and complexity. It refers to the variation in performance (execution time) observed when a given executable is run with the same input parameters multiple times on an HPC system. Users may observe several times worse performance than expected for jobs submitted at different times that are otherwise identical. This can happen due to operating system (OS) noise or contention for shared resources such as the network or filesystem [1], [2]. Performance degradation negatively affects the end user as well as the operational efficiency of the system.

When faced with performance variability, users are unable to estimate run times for their jobs accurately, and hence may request nodes for longer times than may be required. Most HPC systems use batch schedulers such as Slurm [3] and LSF [4] to run jobs and assign allocated resources to them. These batch schedulers require a run time limit provided by the user that serves as an upper bound on the duration the job will be allocated resources. When the end user cannot predict the total run time of their application due to large variances, they will often over-estimate total job time input to the scheduler, which may result in longer queue wait times [5]. On the other hand, if the user underestimates performance degradation, their application may be terminated prematurely resulting in loss of job progress. Both these situations adversely affect resource utilization as well as job throughput. In addition, variability also makes it challenging to analyze the performance bottlenecks in a parallel application, and study the impact of performance improvements made to a code.

The overall operational efficiency of the HPC system also suffers due to performance variability as jobs take longer to complete on average. This results in fewer jobs being completed over time and causes the system's throughput to diminish. Additionally, the scheduler receives less realistic time estimates from users which inhibits its scheduling capability. Hence, it is important to tackle the performance variability problem not just at the individual user level, but at the system level.

With storage becoming relatively inexpensive, the amount of system related data being logged has increased considerably. Software such as the Lightweight Distributed Metric Service (LDMS) [6] are being used to collect and aggregate multiple data streams from system hardware and software on HPC systems. We believe that such system data holds clues about the performance variability of individual jobs. Moreover, we hypothesize that past historical data can give us a reasonable indication of the performance of jobs in the near future.

In this paper, we use historical job information and system monitoring data to accurately predict if a job in the scheduler queue will experience variation if scheduled right away. We observe that ML models trained on historical data for control jobs perform exceedingly well in predicting if a job in the queue will experience variation. Our models obtain an F_1 score of 0.95 in cross-validation. We use these trained models as an input to the job scheduler to influence scheduling decisions with a goal to reduce variability. Predictions from the trained ML models are used by the scheduling algorithm to delay scheduling of certain jobs in the queue if their run time may vary significantly. We design and implement an end-to-end system, which we call the Resource Utilization aware Scheduler for HPC (`RUSH` in short), for collecting application and system data, accurately modeling and predicting application variation, and intelligent adaptive scheduling based on such predictions.

Using real workloads and an implementation of our scheduling algorithm on a large allocation of the Quartz cluster at LLNL, we show that `RUSH` effectively reduces the variation and maximum run time of applications without significantly

affecting makespan or mean queue time. We see up to 5.8% improvement in maximum run time and no performance outliers. In our experiments, we see the average number of runs experiencing variation drop from 17 to 4 using `RUSH`. Additionally, we show that our ML model and scheduler can generalize to applications not included in its training data as well as different inputs for the same applications.

Our work makes the following important contributions:

- We create a pipeline for collecting longitudinal job performance data.
- We train machine learning models using system monitoring data to accurately predict the occurrence of variation for an application.
- We propose a new resource utilization aware scheduling algorithm that reduces application variation and lowers the maximum run time.
- We implement our scheduler and run it on a large allocation over several different workloads to show its actual improvement.
- We show that our scheduling algorithm generalizes to applications its ML model has never seen.

II. Related Work

In this section, we discuss related work on identifying and mitigating performance variability. We also discuss previous work on intelligent job scheduling.

A. Analyzing System Monitoring Data

The recent availability of extensive monitoring data on HPC systems has led to numerous studies looking at analyzing this monitoring data and studying longitudinal patterns. These studies try to identify performance trends and discover their root causes. A recent work [1] uses performance counters from the duration before a job runs to study the root causes of performance variation. Another work [7] uses performance counters to model expected run time of proxy applications and assess the quality of future runs.

B. Mitigating Performance Variability

Existing work [8], [9], [2], [1], [10] has extensively studied the causes of performance degradation and the degree that it hinders application performance. Recent work has looked at utilizing network control mechanisms and application aware machine learning to mitigate network contention [11]. Another approach shows that throttling messages in flight per-core can significantly reduce congestion and increase application performance [12]. This work also prescribes proactive congestion mitigation. Meanwhile, [13] looks at monitoring network health with "canary" jobs to prevent variability due to I/O contention.

C. Intelligent Job Scheduling

Previous work [14], [15], [16], [17] has used machine learning to predict how reliable user provided run times are and how much slowdown running a particular job will incur. These works use information provided in the job submission script and historical job log data to choose a scheduling strategy or queue order. Additionally, these works evaluate their models using simulations based on application trace and job log data.

Naghshnejad et al [14] propose a scheduling policy based on a meta-learning technique that predicts the reliability of a users predicted run time. This confidence is utilized to do more aggressive backfilling. Carastan-Santos et al [15] use historical job log data to similarly account for inaccurate user predictions. Additionally, this work uses a simulated environment and non-linear regression models to find optimal utility functions for scheduling workloads.

There is also existing work that uses current system performance to schedule jobs [18]. This work uses reinforcement learning to gradually learn an optimal scheduling policy with a weighted sum of makespan and queue time as an objective.

III. Data Collection and Modeling

Previous work [13] that used I/O performance predictors has shown that using current information about the file system to delay scheduling of I/O-intensive jobs can improve resource utilization. This shows that the relative health of shared resources is a meaningful predictor in determining if an application will experience performance variation in the near future. It also shows that delaying the execution of an application when shared resources are congested can lead to less variation and higher resource utilization. Thus, we hypothesize that deploying a resource utilization aware scheduler will also improve these two metrics.

An adaptive job scheduler would require *online* knowledge of system health and its relationship with application performance. Existing works have shown that system monitoring data can provide this meaningful insight into the health of shared resources (see Section II-A). Moreso, [7] shows that this data in conjunction with historical runs from proxy applications can accurately predict their relative performance. With this information available *apriori*, the job scheduler can alter its queue order to prevent variation and further congestion.

Thus, we collect system data, shared resource benchmarks, and proxy application profiles over time to build statistical models to be used in our scheduling algorithm. Below, we present the data used in our ML pipeline as well as our collection methodology. All of the data was collected on the Quartz system at LLNL. Quartz is a fat-tree cluster with 2,988 Intel Xeon E5-2695 compute nodes, connected by a Cornelis Networks Omni-Path fabric.

A. System Monitoring Data from the HPC Cluster

Recent years have seen the growth of software stacks to collect and analyze system data. We utilize these to gather information about the state of the machine as proxy applications run. With this data we can infer causes of performance anomalies and predict future occurrences with statistical models.

We include two sources of system counters in our data: sysclassib and lustre_client. Sysclassib is a table of counters containing values for the endpoint traffic such as the xmit rate and recv rate. The lustre_client table of counters contains the number of system calls to the Lustre parallel filesystem as

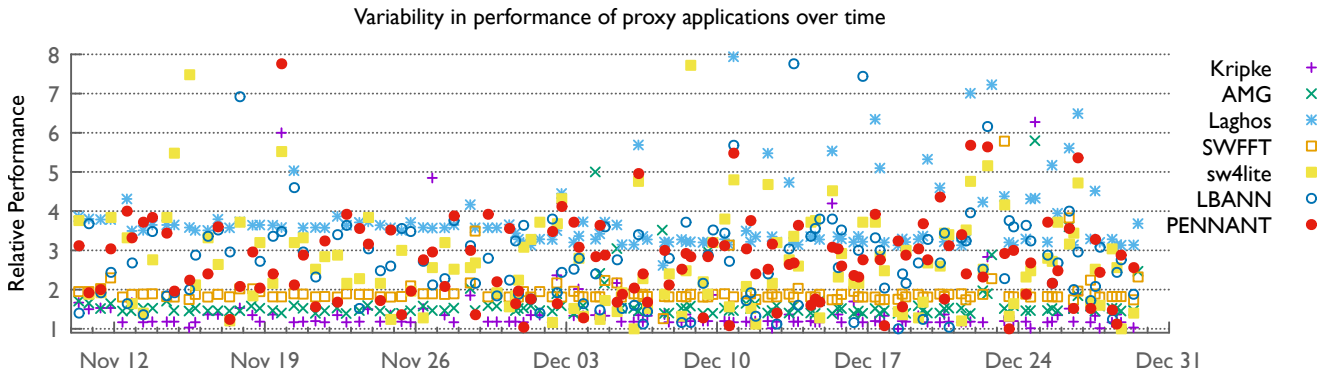

Fig. 1. Observed variability in the performance of proxy applications run in the production batch queue of Quartz at LLNL over a period of two months in 2020. Performance is relative to the lowest execution time per application. Several applications see over 2× performance variation and some even up to 14×. (Several data points over 8× not shown in the plot.)

well as the amount of data being written/read. These data are consistently collected by LDMS, which writes the aggregated data into Cassandra tables on the LLNL Sonar system. Each sample in the table is indexed on the hostname of its source node and the timestamp from when it was recorded.

We utilize aggregate data points in training rather than temporal data (see Section III-D). These aggregate data points are calculated by aggregating counter values over some duration before a job is run. In our training data we use five minutes as the duration. The counters are aggregated via minimum, maximum, and mean and, thus, each counter column becomes 3. For example, the `xmit_rate` counter in sysclassib becomes `max_xmit_rate`, `min_xmit_rate`, and `mean_xmit_rate`. This data is also aggregated over compute nodes as it is recorded on each node. In our dataset we include the aggregates over both all compute nodes and the nodes exclusive to the job being run, so that we can compare the results from training the ML models over data from the entire machine versus the nodes exclusive to each data sample.

B. Proxy Applications Used in Control Jobs

This system data provides insights into the state of the machine, while proxy applications can provide insight into how applications perform on that machine. Proxy applications are simpler programs that mimic the typical workload of a larger scientific code. As a result, they are ideal for generating data that is representative of historical workloads of HPC systems. We run seven proxy applications at frequent intervals to collect performance data: Kripke [19], AMG [20], Laghos [21], SWFFT [22], PENNANT [23], sw4lite [24], and LBANN [25]. These applications represent a range of computational and communication patterns in a variety of scientific domains. Each was compiled with the default build settings in their build documentation using the system intel compiler.

We submitted jobs for each proxy application two to three times a day on the cluster from August 2020 to February 2021 with each job being run at various times in the day. Each application ran on 16 nodes using 512 cores in total. All of the applications use MPI for distributed memory parallelism and run in CPU only mode. Each run was profiled with HPCToolkit [26]. We use Hatchet [27] to read in the HPCToolkit profiles, and extract the inclusive run time of the main compute region in each code. Figure 1 shows the variation experienced by each application between November 12th, 2020 and December 31st, 2020 relative to each application's minimum running time. In mid-December, there was a significant spike in variation in all applications.

While all of the applications experienced some degree of variation, they may have different sources of this variation from their types of workload. Thus, the type of workload is included in the dataset as a one-hot encoded vector over compute, network, and I/O intensive. For the training data we hand selected these values. However, in production this data needs to be provided accurately to the scheduler. This can be given by the user, empirical methods, or binary analysis.

C. Benchmarks Used to Monitor System Health

Before and during proxy application runs we collected several metrics related to the health of the system as well as the performance of the job. Right as each job is scheduled we ran two MPI benchmarks with mpiP to gather information about the network health. These benchmarks are used to offer some information to the ML model as to how congestion is currently affecting running applications.

The first benchmark is a simple ring routine with send/recv that passes around a 100MB token for ten iterations. The second calls AllReduce on 100MB of random data for five iterations. Message sizes and iteration counts for these benchmarks were picked empirically such that there was sufficient variance for the ML models to learn from, but not enough to cause significant communication overhead.

Using mpiP we record the time spent waiting on the blocking Send, Recv, and AllReduce calls on each node. For the dataset we record the minimum, maximum, and mean of each of these values across used nodes. This becomes nine features in each data point.

D. Input to the Machine Learning Models

Each of these application runs becomes a sample in the final dataset. The input features for each sample consist of the minimum, maximum, and mean of every counter in the sysclassib and lustre_client tables, the user provided application type label, and the nine aggregated benchmark results. Finally, each sample has its run time and z-score as output labels. The resulting dataset and its features are presented in Table I.

TABLE I
DESCRIPTION OF DATA SOURCES AND THE NUMBER OF COUNTERS/FEATURES DERIVED FROM THEM FOR TRAINING THE ML MODELS.

Input source	# Counters	# Features	Description
sysclassib	22	66	InfiniBand counters
opa_info	34	102	Omni-Path switch counters
lustre2_client	34	102	Lustre client metrics
MPI benchmarks	3	9	Execution time
Proxy applications	-	1	Compute Intensive
	-	1	Network Intensive
	-	1	I/O Intensive

This collected data is designed to encapsulate the machine state during an application run and the relative performance of that run. The goal of the ML models is to analyze the machine state data and basic information about a job and predict if it will experience variation. Several recent works have explored using various models to learn over system data [28]. We find static models trained on aggregate statistics to work well for our purposes.

IV. RUSH: RESOURCE UTILIZATION AWARE SCHEDULER FOR HPC

We now present the two main components of RUSH: an ML-based variability predictor, and a model-based adaptive job scheduling algorithm, and also describe the design of the entire pipeline. Figure 2 highlights how each component of the pipeline fits together and their input/outputs.

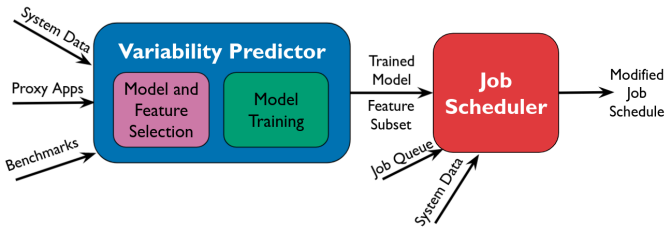

Fig. 2. Pipeline Overview. The ML model is trained offline on historical jobs and system data. Optimal features are selected and a trained model is exported. This trained ML model, current system data, and submitted jobs are provided as input to the job scheduler that, in turn, decides a new order for scheduling jobs and mapping them to system resources over time.

A. Variability Predictor Module

The first module in the RUSH pipeline uses system and control job data to predict if variation will occur from running a job on the current system state. There are three inputs to this module: system counters from Sonar, profiles from longitudinal runs of proxy applications, and timings from the MPI benchmarks. Within this module feature and model selection are done first followed by training and exporting the chosen model and features.

The ML models in the first component use the input data as described in Section III. We set up the ML problem as a classification task with the goal of classifying the occurrence of variation given the system and benchmark data. The input to this model consists of the 282 features listed in Table I. For model and feature selection we use binary classification and set the label of each data as 0 or 1. The first label, 0, is assigned when an application's run time is less than 1.5 standard deviations of its mean run time. This signifies no variation. On the other hand, we assign a label of 1 when the run time is greater than 1.5 standard deviations from its mean. These variations are computed per-application using the mean and standard deviation for each application's run times, but the model is trained on data from all applications. Instead of arbitrarily selecting an ML model we train a variety of models and use their F_1 scores to to compare their performance (see Section VI-B).

The set of classifiers used are standard models and we use the best performing in the pipeline (see Section V) based on F_1 score. The models used are Extra Trees, Decision Forest, K-Nearest Neighbors, and AdaBoost. Each is trained using stratified cross validation to preserve the imbalance of the data. To cross validate we split the data using six applications for training and one for validation. This is performed over every possible partitioning.

Features are selected after model selection using recursive feature elimination. Features are eliminated recursively and the set with the highest F_1 score are kept. For the Extra Trees and Decision Forest models, which have metrics for feature importance, the least import features are removed first during feature elimination.

After selecting the model and feature set the second component outputs the trained ML model that can be used offline. The chosen model is trained using the same data and k-fold cross-validation. However, this model is trained on three output classes: no variation, little variation, and variation. Here the variation label stays the same while the no variation label is assigned when an application's run time is less than 1.2 standard deviations from its mean run time. Little variation is when the application's run time is between 1.2 and 1.5 standard deviations of its mean run time. These labels are chosen based on our observations of application performance behavior.

B. Model-based Adaptive Job Scheduler

The second component of RUSH is an intelligent job scheduler that uses the models trained by the variability predictor as input. The scheduler has three inputs: the trained ML model, job queue, and systems data. It uses the ML model with the systems data as input to implement a scheduling policy and map jobs from the queue to system resources.

The proposed scheduler utilizes predictions provided by the ML model to delay scheduling of jobs that will experience variation. We do this by running the ML model on the current system counters whenever a new job is about to be scheduled. If the model predicts variation for this job we skip over it and look at the next one in the queue. The delayed job remains at the top of the queue and will be the first to be considered for scheduling next time resources become available.

Our job scheduling modifications are general enough that they can be used to modify other existing policies. For example, we show that we can easily modify the FCFS+EASY scheduling algorithm presented in Algorithm 1. The main and backfilling policies can be replaced with other queue ordering policies. One common example is Shortest Job First or SJF. This allows `RUSH` to utilize the benefits from other optimal queue ordering policies assuming they work by statically re-ordering the queue.

Algorithm 1 Scheduling Algorithm. This standard algorithm queues jobs using policy $\mathcal{R}_1$ and uses EASY to backfill smaller jobs. $Start(\cdot)$ is used to launch jobs when resources become available.

Input $Q \leftarrow$ queue of jobs
$M \leftarrow$ ML model
$S \leftarrow$ current machine state
SkipTable $\leftarrow$ Count of times skipped for each job
$\mathcal{R}_1 \leftarrow$ Queue ordering policy
$\mathcal{R}_2 \leftarrow$ Backfill ordering policy

```
1  sort Q according to R_1
2  for job j ∈ Q do
3     if j can be started currently then
4        pop j from Q
5        Start(j, Q, M, S, SkipTable)
6     else
7        Reserve j at earliest possible time
8        L ← Q \ {j}
9        sort L according to R_2
10       for job j' ∈ L do
11          if j' can be started currently without delaying reservation
            of j then
12             pop j' from Q
13             Start(j', Q, M, S, SkipTable)
14          end if
15       end for
16       break
17    end if
18 end for
```

This algorithmic change only affects the job queue ordering. It is agnostic towards resource mappings and network topology. These can be accounted for in the start function when jobs are being launched or by a separate software system. Therefore, the proposed algorithm only needs to modify the $Start(\cdot)$ function as shown in Algorithm 2. This function takes care of putting jobs back on the queue when they are being delayed.

However, continually delaying jobs can lead to starvation. To prevent job starvation the modified schedule also includes a hard limit on the number of times a job can be skipped over. In our experiments we set this to 10, but the threshold was never met. This parameter could be extended to be per-job and used to enforce priorities or even ignore the scheduling delay entirely for certain jobs.

Algorithm 2 Modified $Start(\cdot)$ Function. This is called to launch jobs when resources are available. This modified version in `RUSH` puts jobs back on the queue if they will vary in performance significantly.

Input $j \leftarrow$ job
$Q \leftarrow$ scheduler queue
$M \leftarrow$ ML model
$S \leftarrow$ current machine state
SkipTable $\leftarrow$ Count of times skipped for each job

```
1  if SkipTable[j] < j.skip_threshold and
   M(j, S) ∈ variation labels then
2     SkipTable[j] ← SkipTable[j] + 1
3     push j after front of Q
4  else
5     launch job j
6  end if
```

This leads us to the following design of the $Start(\cdot)$ function in Algorithm 2. It first checks if a job j is past its skip threshold (line 1). When j is past the threshold, then the **and** is short-circuited and j will be run (line 5). If j is within its skip threshold, then `RUSH` will evaluate the ML model $M(j, S)$ (line 1). Variation being predicted will lead to j being put back on the queue (lines 2-3). Otherwise, j will be run (line 5).

V. Implementation

In its entirety, `RUSH` requires recording large amounts of system data, training ML models, and modifying an existing batch scheduler implementation. This section discusses how these components of the pipeline were implemented.

A. Variability Predictor Implementation

To facilitate our scheduler, we utilize a data pipeline (Figure 2) that controls running the proxy jobs, collecting the performance and run time system data, and training the ML models. This pipeline needs to be portable and efficient, so that the same experiments and scheduler adaptations can be used on other machines.

To make our pipeline portable we designed it entirely using bash to control job launches and Python to collect and analyze data. Jobs are launched using configurations from environment variables and can launch using either LSF or Slurm based job schedulers. Once the jobs run the data from these jobs are aggregated and analyzed using Python.

In addition to portability the pipeline needs to be efficient in its storage and analysis of large amounts of data. Collecting 32 HPCToolkit profiles per day can create several million files in a short amount of time. To alleviate this storage burden we only store database files from `hpcprof-mpi` in addition to the hatchet dataframes of them.

Given a set of application runs we collect a unified dataset depicted in Table I. To build contained datasets we query the Sonar tables for aggregated LDMS data. We collect counter information for the duration prior to a job running. In our tests this was the five minutes prior to each proxy application's run.

TABLE II
DESCRIPTION OF EXPERIMENTS RUN IN A SYSTEM RESERVATION OF 512 NODES OF QUARTZ TO COMPARE RUSH TO THE BASELINE.

Experiment	Name	Applications	# of Jobs	Description
ADAA	All Data All Applications	All	190	ML model trained on data from all running applications
ADPA	All Data Partial Applications	Laghos, LBANN, PENNANT	150	Subset of 3 applications running
PDPA	Partial Data Partial Applications	Laghos, LBANN, PENNANT	150	ML model trained on AMG, Kripke, sw4lite, SWFFT
WS	Weak Scaling	All	190	Jobs run on 8, 16, and 32 nodes – weak scaling
SS	Strong Scaling	All	190	Jobs run on 8, 16, and 32 nodes – strong scaling

The counters were reduced over this interval with the minimum, maximum, and mean of each being included as a column in the dataset. Next the profiling information, including the wall clock time, is added into our table. This table is stored in a Pandas dataframe, which is pickled and compressed for easier use in the rest of the pipeline.

Prior to running experiments we train the ML models over the collected data sets and select the best one based on their F_1 score and accuracy. At the end of the pipeline the models are pickled and exported for use in the scheduler.

B. Job Scheduler Implementation

Using this exported model we modify the Flux [29] framework to implement our scheduler. Flux is a job scheduling software designed for HPC that integrates graph-based resource modeling with traditional batch scheduling. This section details how we implement our algorithm in Flux.

`RUSH` adds a scheduling policy within Flux to implement its algorithm. This is done by adding a new "scheduling policy" class to Flux. We extend the class `queue_policy_fcfs_t`, which in turn extends the general `queue_policy_base_t` class. Our subclass `queue_policy_rush_t` implements the scheduling algorithm detailed in Section IV-B.

The `RUSH` implementation provides a modified function for ordering the queue. It first orders the queue with $\mathcal{R}_1$ as FCFS. When jobs are about to be run a Python script is first executed that runs the ML model with the next job as input.

This Python script then reads the collected counter data, runs the ML models, and provides its prediction to standard output. Our implementation uses this to make a scheduling determination as defined in Algorithm 2.

Jobs are matched to resources using Flux's default algorithm. Information about this mapping is captured implicitly in the system counters. Thus `RUSH` can be utilized with any resource mapping algorithm.

VI. EXPERIMENTAL SETUP

In this section, we describe the experiments and metrics used to evaluate the ML models and the new job scheduler.

A. Scheduling Experiments

To test the effectiveness of our scheduler we designed experiments to mimic typical workloads on an HPC system. We then compare the proposed scheduling policy on this workload with the default FCFS+backfilling scheduler as a control.

In order to create an HPC system-like environment we ran all of the experiments within a fixed set of 512 nodes on Quartz. These nodes lie in the same pod of the fat-tree cluster. The nodes are allocated by the system Slurm scheduler as a single job and we run Flux within this allocation to handle scheduling jobs in the experiments.

To mimic a typical HPC workload we design several experiments using the seven proxy applications listed in Sec III-B. We setup a queue of jobs that takes between 30 and 50 minutes for all of them to run to completion. Each job runs on 16 nodes with 512 processes. At the beginning of the experiment we submit 20% of the jobs to the Flux queue immediately and submit the rest uniformly over 20 minutes. This mimics normal scheduling behavior where knowledge of every job to be scheduled is not known apriori.

Since we ran on a single pod on the fat-tree, we used a noise job that runs on 1/16th of the nodes in the experiment that continuously sends variable amounts of all-to-all traffic on the network. This allowed us to run fewer experiments as we observed variation more frequently with the noise. To account for other system noise we run ten trials of each experiment: five with FCFS+EASY and five with `RUSH`.

We ran several different experiments to test how the scheduling policy performed under different circumstances. Table II highlights the experiments conducted within each 512-node reservation.

We first test the scheduling policy on all of the applications in "ADAA". This experiment runs all seven proxy applications and uses an ML model trained on a dataset containing runs from all seven applications.

To test the generalizability of the scheduler experiment "PDPA" only runs three applications and uses the ML model trained on the other four applications exclusively. We use Laghos, LBANN, and PENNANT as the applications to run and AMG, Kripke, sw4lite, and SWFFT to train the ML model. Experiment "ADPA" runs the same applications, but uses the full dataset for training. This serves as a control for "PDPA".

The final two experiments, "WS" and "SS", test how the policy generalizes to different scales of the jobs. Both run all of the applications and use an ML model trained on all of the data. However, they run each application on 8, 16, and 32 nodes. "WS" uses weak scaling to change the input parameters and "SS" strong scaling.

B. Metrics for Evaluating the ML Models

Before the scheduler is run in these experiments the ML models need to be trained and exported. This section discusses

the metrics used to evaluate the success of the models in predicting variation.

Performance variation is rare and, thus, the dataset is imbalanced. There are significantly more samples with little or no variation than there are samples with variation. This means testing accuracy is not a useful performance metric. A model that always predicts that no variation will occur would still yield an accuracy greater than 90%, but not provide any meaningful information to the scheduler. Due to this limitation, we use precision and recall related metrics to evaluate the success of our models. In particular, we use the F-measure (F_1 score) to compare and find the best performing model.

$$F_1 = \frac{\text{tp}}{\text{tp} + \frac{1}{2}(\text{fp} + \text{fn})}$$

where tp is the number of true positive predictions, fp the false positives, and fn the false negatives. F_1 score is a standard measure for how well models predict imbalanced labels.

When comparing different models, we use the average F_1 score from cross-validation. The F_1 score was calculated for the binary classification problem of variation vs no variation.

C. Metrics for Evaluating the Job Scheduler

We record different metrics that help us evaluate our new job scheduler across multiple different axes of improvement. Schedulers can provide improvement in several different areas, each of interest to different parties in the supercomputing eco-system. Providing reliability and high resource utilization is important to system administrators, while end-users may be more concerned with wait queue time and ease-of-use. Additionally, the efficiency of the scheduler is typically crucial to everyone.

Scheduler efficiency can be measured in terms of the *makespan*, which is defined as the duration from the submission of the first job to the end of the last job. The makespan describes the amount of time it takes a scheduling policy to complete a workload on a certain system.

However, some policies with better makespans may see adverse performance in other areas. So we also record the *mean time in queue* as well as the *mean job variation per application*. The mean time in queue will show how delaying jobs impacts the average time spent waiting in the queue. The job variation will indicate to what degree `RUSH` successfully mitigates run time variation.

VII. Results

We now present our results from training the machine learning models and the job scheduling experiments.

A. Prediction Accuracy of ML Models

Figure 3 presents the performance of different ML models we experimented with based on their F_1 scores. We see that given the system data in Table I and longitudinal run time data (see Section III-B), the ML pipeline described in Section IV-A is able to accurately predict run time variation.

The high F_1 scores show that the models can predict true labels or instances of variation well. While all of them perform well in this regard, the AdaBoost classifier outperforms the others. The results in the rest of the paper use Adaboost as the classifier.

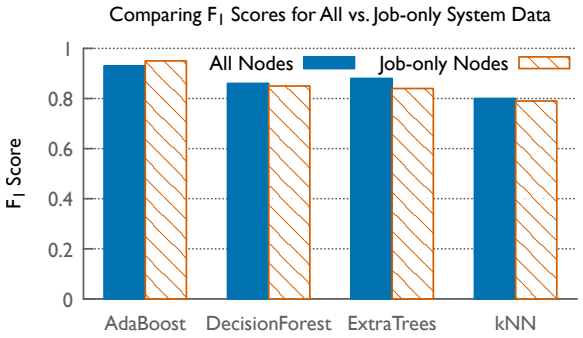

Fig. 3. F_1 scores for different ML models. We see that the AdaBoost model has the highest F_1 scores. Additionally, we see that the models have comparable performance even without access to full system data.

The models are also insensitive to data exclusivity. We have two choices when aggregating data from the system. We can either aggregate over all the nodes on the system or only over those nodes that are allocated to the jobs in the dataset. When system data from only the job's nodes are used, we see comparable performance to training over all the nodes in the system. This is an important performance component as it allows the scheduler to only collect subsets of system data at a time (from the nodes a job is going to be scheduled on) when making scheduling decisions. This is a significant reduction in data processing that allows the scheduler to aggregate counters more frequently and efficiently.

B. Reduction in Application Performance Variability

The model accuracy results above confirm that we can use the trained model to advise the job scheduler regarding whether an incoming job will experience variation or not. Next, we discuss results from the experiments described in Section VI-A and how `RUSH` helps in reducing performance variation.

Figures 5 and 4 show the number of runs that experienced variation in the first three experiments in Table II. Averaged across the five repetitions of the ADAA experiment, FCFS+EASY has between 1.5 and 3.5 runs on average per application with significant variation (see Figure 5). Using `RUSH`, this is reduced to between 0 and 1.5. The most variation prone applications, Laghos and LBANN, have almost no occurrences of significant variation when the `RUSH` scheduler is used. This shows the ability of the scheduler to reduce variation when its ML model has apriori knowledge of all the applications being run.

Experiments ADPA and PDPA show that the variation improvement also holds when the ML model has been trained on a subset of the running applications. This is shown in Figure 4 where we see similar improvement when the ML model is trained over the full dataset (left) versus a partial dataset (right). Compared to ADAA, ADPA and PDPA show slightly higher

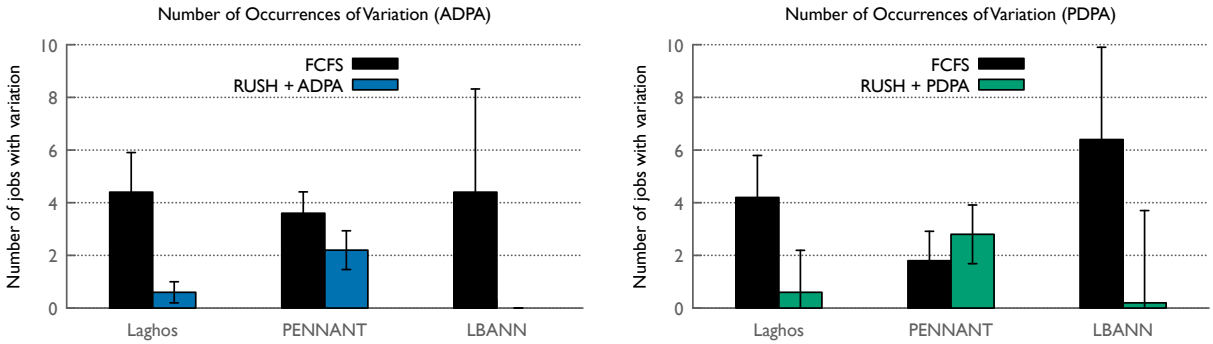

Fig. 4. There is only a slight increase in the number of applications experiencing variation when using the ML model trained on data from all of the applications (left plot, ADPA) and separate applications (right plot, PDPA.)

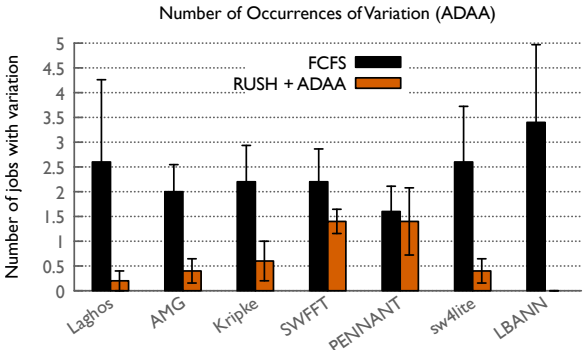

Fig. 5. The number of runs that experience variation significantly reduces under the proposed scheduler (RUSH) for the ADAA experiment when compared to FCFS+EASY.

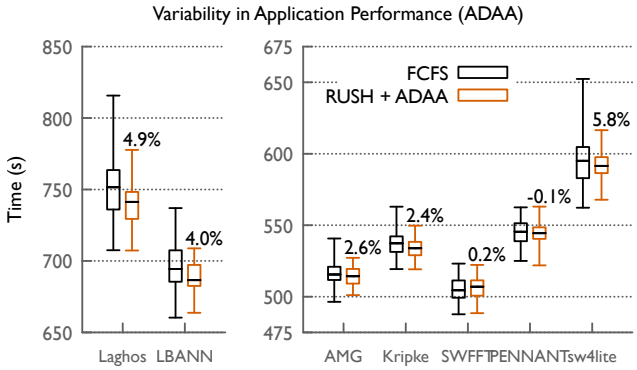

Fig. 6. Distribution of execution times for each application in the ADAA experiment. RUSH reduces the maximum run time and the range of run times.

amounts of variation in both FCFS+EASY and RUSH. This is due to the fact that Laghos and LBANN are the applications with the most variation and now more instances of them are running together than before. Pennant ends up having more runs with variation on average in PDPA than ADPA in the RUSH experiment. However, the increase is small in comparison to the decreases in LBANN and Laghos.

Since fewer runs suffer from variation using RUSH, we also expect the run times of each application to be more predictable. Figures 6, 7, and 8 present the run time distributions in each experiment. The run time distribution includes all of the runs of each experiment in Section VI-A split by application. Figure 6 compares the run times of the proxy applications between FCFS+EASY and RUSH scheduling policies for the ADAA experiment. We observe that the maximum and mean run times reduce for the most sensitive applications, Laghos, LBANN, and sw4lite. The scheduling policy is able to successfully reduce variation in most instances. This is shown by the smaller ranges in run times and number of runs closer to the mean.

From these results we also see an improvement in the maximum run time. This is likely the most important improvement from the perspective of an end-user as now they have a tighter upper limit on their application's running time.

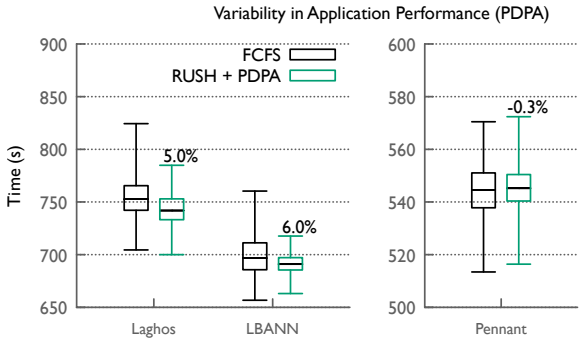

Fig. 7. Distribution of execution times for each application in the PDPA experiment. The scheduler still performs well for applications where its ML model has never seen their data.

As before, Laghos, LBANN, and sw4lite, all experience the largest improvement in terms of maximum run time.

In Figure 7 we see that RUSH performs just as well when it has partial data versus full data. PDPA has similar improvements in maximum run time when compared with ADAA. In our experiments we find that ADPA, the control for PDPA, shows similar results to ADAA for LBANN, PENNANT,

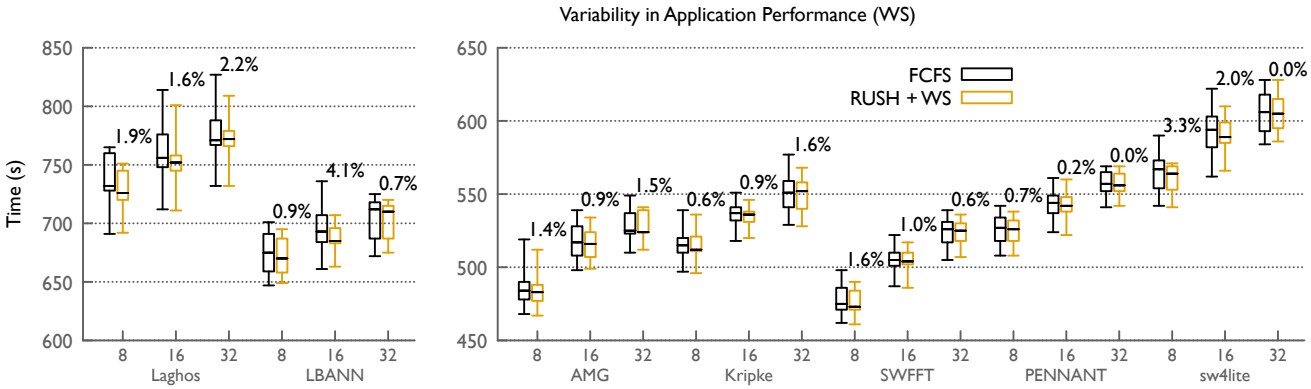

Fig. 8. Distribution of execution times for each application in the Weak Scaling (WS) experiment.

and Laghos. We can conclude that having access to historical runs for an application prior to scheduling is not necessary to reduce its maximum run time. The generalizability of RUSH is important, since this data is typically not readily available.

Figures 8 and 9 present the experiments where the applications are run under weak and strong scaling respectively. In the WS experiment, RUSH reduces the spread of run times and the maximum run time more in the 8 and 16 node count runs. This is likely due to more communication in the 32 node runs and bias in the ML model from only training on 16 node runs.

Figure 9 shows the percent improvement in maximum run time when the applications are strong scaled. We see that the scheduler still provides improvement even as the amount of work per node decreases. For each application the maximum run time is reduced and sw4lite and LBANN show the greatest improvements. In experiments WS and SS, there were no applications with increase in the maximum run time. The run time distributions either stayed the same or, more often, reduced in range. This displays the ability of RUSH to extend to other node counts even under different types of scaling.

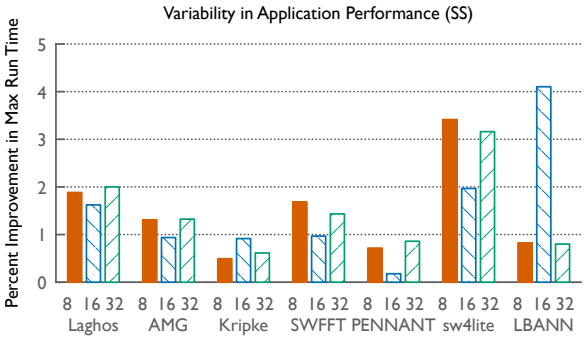

Fig. 9. Percentage improvement in the maximum run time for each application in the Strong Scaling (SS) experiment when comparing RUSH with the baseline.

C. Scheduler Evaluation

In addition to mitigating variation for individual users, we also want to ensure that the scheduler does not impact system throughput negatively. We start with comparing the makespan for the two scheduling policies in Figure 10. For each experiment, the makespan is improved by between 18 and 66 seconds. The variation in each application has been reduced without burdening the makespan significantly and in some cases improving it. By reducing the expected run time of some of the applications, RUSH reduces the duration of some of its jobs. In cases where a significant amount of variation is prevented, the scheduler will have a lower makespan.

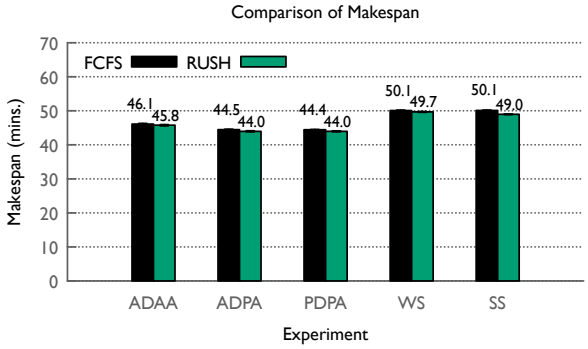

Fig. 10. Scheduler makespans. For each experiment this figure displays FCFS+EASY and RUSH's makespans averaged over their five trials. RUSH outperforms FCFS+EASY in each experiment.

Figure 11 shows the differences in wait times for each application for the ADAA experiment. This plot only includes wait times for the 80% of applications that were not placed in the queue at the start of the experiment. In the case of RUSH, the wait times are spread out and show both favorable and worse performance compared to the FCFS+EASY scheduler. The average wait time went up for variation intensive applications such as Laghos, sw4lite, and LBANN. This is due to them being pushed back in the queue more often than others. Both Kripke and AMG got through the queue faster on average in the RUSH scheduler. Though the wait times vary, they are always within a minute. This less than a percent increase in wait time is insignificant, especially compared to the reduction in variation that can be obtained at its cost.

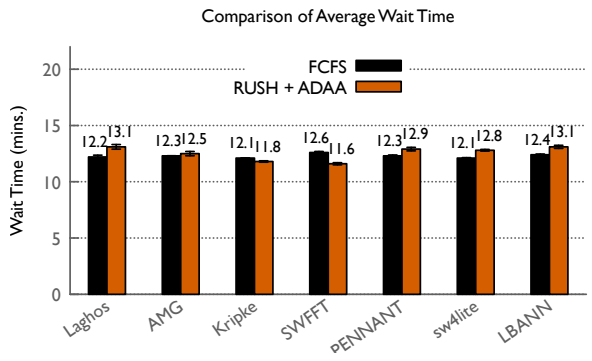

Fig. 11. Average wait time per application in experiment ADAA. RUSH has a larger range of wait times and is often higher.

VIII. Conclusion

Performance variability arising from resource sharing between concurrently running jobs on HPC systems can lead to inefficiencies for jobs and the system as a whole. In this work, we have shown that historical run information and system monitoring data representing the current system state can be used to predict the variation a job may incur if scheduled right away. Exploiting this result, we have developed a resource utilization aware scheduling algorithm called RUSH that uses machine learning models to predict the future performance of incoming jobs. We demonstrate that RUSH can be used to modify the default job schedule to mitigate variation and even improve overall system utilization. In practice, an implementation of this policy could improve utilization as well as allow users to run code with more predictable run times.

In the future, we will continue investigating how this apriori knowledge of performance variation can be integrated with schedulers and other system mechanisms to improve resource utilization. We will also further explore latent performance metrics and their use in HPC-focused scheduling.

Acknowledgment

This material is based upon work supported in part by the National Science Foundation under Grant No. 2047120. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-825754).

References

[1] I. J. Costello and A. Bhatele, "Analytics of longitudinal system monitoring data for performance prediction," 2020.

[2] A. Bhatele, J. J. Thiagarajan, T. Groves, R. Anirudh, S. A. Smith, B. Cook, and D. K. Lowenthal, "The case of performance variability on dragonfly-based systems," in *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '20. IEEE Computer Society, May 2020.

[3] "Slurm workload manager," 2020. [Online]. Available: https://slurm.schedmd.com/documentation.html

[4] "Ibm spectrum lsf session scheduler," 2021. [Online]. Available: https://www.ibm.com/docs/en/spectrum-lsf/10.1.0?topic=lsf-session-scheduler

[5] D. Klusáček and V. Chlumskỳ, "Evaluating the impact of soft walltimes on job scheduling performance," in *Workshop on Job Scheduling Strategies for Parallel Processing*, 2018, pp. 15–38.

[6] A. Agelastos, B. Allan, J. Brandt, P. Cassella, J. Enos, J. Fullop, A. Gentile, S. Monk, N. Naksinehaboon, J. Ogden, M. Rajan, M. Showerman, J. Stevenson, N. Taerat, and T. Tucker, "The lightweight distributed metric service: A scalable infrastructure for continuous monitoring of large scale computing systems and applications," in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 154–165.

[7] O. Aaziz, J. Cook, and M. Tanash, "Modeling expected application runtime for characterizing and assessing job performance," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, 2018, pp. 543–551.

[8] F. Petrini, D. J. Kerbyson, and S. Pakin, "The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q," in *Proceedings of the 2003 ACM/IEEE conference on Supercomputing (SC'03)*, 2003.

[9] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs, "There goes the neighborhood: performance degradation due to nearby jobs," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. IEEE Computer Society, Nov. 2013. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503247

[10] B. Li, S. Chunduri, K. Harms, Y. Fan, and Z. Lan, "The effect of system utilization on application performance variability," in *Proceedings of the 9th International Workshop on Runtime and Operating Systems for Supercomputers*, ser. ROSS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 11–18. [Online]. Available: https://doi.org/10.1145/3322789.3328743

[11] A. Patke, S. Jha, H. Qiu, J. M. Brandt, A. C. Gentile, J. Greenseid, Z. Kalbarczyk, and R. K. Iyer, "Application-aware congestion mitigation forhigh-performance computing systems," *CoRR*, vol. abs/2012.07755, 2020. [Online]. Available: https://arxiv.org/abs/2012.07755

[12] M. Luo, D. K. Panda, K. Z. Ibrahim, and C. Iancu, "Congestion avoidance on manycore high performance computing systems," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 121–132. [Online]. Available: https://doi.org/10.1145/2304576.2304594

[13] M. R. Wyatt, S. Herbein, K. Shoga, T. Gamblin, and M. Taufer, "Canario: Sounding the alarm on io-related performance degradation," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 73–83.

[14] M. Naghshnejad and M. Singhal, "A hybrid scheduling platform: a runtime prediction reliability aware scheduling platform to improve hpc scheduling performance," *The Journal of Supercomputing*, vol. 76, no. 1, pp. 122–149, Jan 2020. [Online]. Available: https://doi.org/10.1007/s11227-019-03004-3

[15] D. Carastan-Santos and R. Y. de Camargo, "Obtaining dynamic scheduling policies with simulation and machine learning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3126908.3126955

[16] M. Tanash, B. Dunn, D. Andresen, W. Hsu, H. Yang, and A. 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)*, ser. PEARC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3332186.3333041

[17] E. Gaussier, D. Glesser, V. Reis, and D. Trystram, "Improving backfilling by using machine learning to predict running times," in *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–10.

[18] D. Zhang, D. Dai, Y. He, F. S. Bao, and B. Xie, "Rlscheduler: An automated hpc batch job scheduler using reinforcement learning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '20. IEEE Press, 2020.

[19] A. Kunen, T. Bailey, and P. Brown, "KRIPKE-a massively parallel transport mini-app," *Lawrence Livermore National Laboratory (LLNL), Livermore, CA, Tech. Rep*, 2015.

[20] V. E. Henson and U. M. Yang, "Boomeramg: A parallel algebraic multigrid solver and preconditioner," *Applied Numerical Mathematics*, vol. 41, no. 1, pp. 155–177, 2002, developments and Trends in Iterative Methods for Large Systems of Equations - in memorium Rudiger Weiss. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0168927401001155

[21] V. A. Dobrev, T. V. Kolev, and R. N. Rieben, "High-order curvilinear finite element methods for lagrangian hydrodynamics," *SIAM Journal on Scientific Computing*, vol. 34, no. 5, pp. B606–B641, 2012. [Online]. Available: https://doi.org/10.1137/120864672

[22] A. P. et al, "Swfft," https://git.cels.anl.gov/hacc/SWFFT, 2017.

[23] C. R. Ferenbaugh, "Pennant," https://github.com/lanl/PENNANT, 2016.

[24] "sw4lite," https://github.com/geodynamics/sw4lite, 2017.

[25] B. V. Essen, H. Kim, R. A. Pearce, K. Boakye, and B. Chen, "LBANN: livermore big artificial neural network HPC toolkit," in *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments, MLHPC 2015, Austin, Texas, USA, November 15, 2015*. ACM, 2015, pp. 5:1–5:6. [Online]. Available: https://doi.org/10.1145/2834892.2834897

[26] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "Hpctoolkit: Tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.

[27] A. Bhatele, S. Brink, and T. Gamblin, "Hatchet: Pruning the overgrowth in parallel profiles," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19, Nov. 2019, lLNL-CONF-772402. [Online]. Available: http://doi.acm.org/10.1145/3295500.3356219

[28] B. Schwaller, B. Aksar, O. R. Aaziz, E. Ates, J. M. Brandt, A. Coskun, M. Egele, and V. J. Leung, "A machine learning approach to understanding hpc application performance variation." 10 2019. [Online]. Available: https://www.osti.gov/biblio/1642784

[29] D. H. Ahn, N. Bass, A. Chu, J. Garlick, M. Grondona, S. Herbein, J. Koning, T. Patki, T. R. W. Scogland, B. Springmeyer, and M. Taufer, "Flux: Overcoming scheduling challenges for exascale workflows," in *2018 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*, 2018, pp. 10–19.