Intelligent Resource Provisioning for Scientific Workflows and HPC

Benjamin T. Shealy
Department of Electrical and Computer Engineering
Clemson University
Clemson, SC, USA
btsheal@clemson.edu

F. Alex Feltus
Department of Genetics and Biochemistry
Clemson University
Clemson, SC, USA
ffeltus@clemson.edu

Melissa C. Smith
Department of Electrical and Computer Engineering
Clemson University
Clemson, SC, USA
smithmc@clemson.edu

2021 IEEE Workshop on Workflows in Support of Large-Scale Science (WORKS) | 978-1-6654-1136-3/21/$31.00 ©2021 IEEE | DOI: 10.1109/WORKS54523.2021.00007

***Abstract*—Scientific workflows and high-performance computing (HPC) systems are critically important to modern scientific research. In order to perform scientific experiments at scale, domain scientists must have knowledge and expertise in software and hardware systems that are highly complex and rapidly evolving. While computational expertise will be essential for domain scientists going forward, any tools or practices that reduce this burden for domain scientists will greatly increase the rate of scientific discoveries. One such example is knowing ahead of time the resource usage patterns of an application for the purpose of resource provisioning. A tool that accurately estimates these resource requirements would benefit HPC users in many ways, by reducing job failures and queue times on traditional HPC systems and reducing costs on cloud computing systems. In this work we present Tesseract, a semi-automated tool that predicts resource usage for any application on any computing platform, from historical data, with minimal input from the user. We employ Tesseract to predict runtime, memory usage, and disk usage for a diverse set of scientific workflows, and in particular we show how these resource estimates can prevent under-provisioning.**

***Index Terms*—high performance computing, machine learning, Nextflow, resource provisioning, scientific workflows**

I. Introduction

In the past two decades, scientific research (as well as many other domains) has been transformed by the collective emergence of three phenomena – machine learning, big data, and high-performance computing (HPC). Recent advances in machine learning were enabled by (1) the increased availability of computational resources, which reside primarily in HPC datacenters, and (2) the massive amount of training data that is generated by devices, ranging from smartphones to DNA sequencers, that become cheaper and more commonplace each year. As a result, the HPC system has itself become a scientific instrument that complements traditional lab equipment, as scientists create increasingly complex workflows to extract insights from large datasets. These computational experiments require a great deal of computational expertise, especially as experiments become large, and domain scientists are struggling to acquire this expertise while doing their own research. Thus the *usability* of HPC systems is a major bottleneck to scientific progress today, and while computational expertise will continue to become a necessary skill for domain scientists, anything that simplifies the process of science experiments at scale will ultimately increase the rate of scientific discovery.

One of the greatest challenges with computational experiments is knowing the amount of resources that are required, such as CPU-hours, GPU-hours, memory, storage, and I/O bandwidth. Understanding the resource usage patterns of an application is critical when using large-scale computing systems. Users must request the resources that they need for an experiment, and there are pitfalls to both under-provisioning and over-provisioning. On shared HPC systems such as university clusters, over-provisioning may increase the time that the job is waiting in the queue, and under-provisioning may cause the job to fail and have to be restarted. On cloud platforms, there are no queue times or walltime limits because resources are highly available, but there are significant financial risks related to incorrectly provisioning other resources such as memory and storage. Thus the lack of knowledge about resource usage patterns is a hindrance on HPC clusters and a major setback on cloud platforms. These challenges are multiplied for scientific workflows with multiple steps, with each step potentially having very different resource requirements. Workflow managers such as Nextflow [6] greatly reduce the burden of executing scientific workflows, but they do not assist users in resource provisioning. An accurate resource prediction tool would confer significant benefits to users simply by addressing the aforementioned problems – reducing job failures, queue times, and costs in the case of cloud platforms.

Resource prediction has received moderate research attention with promising results, but few studies have translated into usable tools for domain scientists. We believe this gap between research and application is due to the complexity and non-uniformity of large-scale computational systems, as well as the understandable lack of computational expertise among domain scientists. To that end, we present Tesseract, a tool that provides intelligent resource prediction for any application, on any computing platform, and can be used by experts and non-experts alike. In this paper, we demonstrate the use of Tesseract with a number of real scientific workflows on a university cluster, and we focus specifically on how Tesseract helps to prevent under-provisioning.

978-1-6654-1136-3/21/$31.00 ©2021 IEEE
DOI 10.1109/WORKS54523.2021.00007

II. Related Work

In this section we describe past research efforts on resource prediction and performance prediction. We define *resource prediction* as the task of estimating how much of some resource will be used by an application before it is run. A closely related task is *performance prediction* or *performance modeling*, which is the task of measuring how efficiently an application uses a resource.

A. Analytical Modeling

The earliest approaches to performance prediction developed analytical models that captured runtime or memory complexity of the application as well as the behavior of the underlying hardware. Many studies [3], [7], [9], [16], [17] developed detailed models of particular applications and architectures in an attempt to capture all of the intricacies of these systems. Analytical modeling can yield very accurate predictions, but a new model must be developed for every new application or architecture, each of which requires deep expert knowledge. As a result, analytical modeling is not suitable for non-expert users such as domain scientists who want to predict the resource usage of the applications they use.

B. Empirical Modeling

Many studies have explored the use of machine learning models to predict resource usage. Ipek *et al.* [15], one of the earliest studies of its kind, trained a multilayer perceptron (MLP) to predict the runtime of SMG2000 with 5-7% test error. In particular, the authors found that allocating an extra core to handle OS-related tasks and modifying the training process to minimize percentage error instead of absolute error mitigated the effect of noise and greatly reduced the test error.

Matsunaga *et al.* [18] trained a number of machine learning algorithms to predict the runtime, memory usage, and disk usage of two bioinformatics applications, BLAST and RAxML. They evaluate several classical algorithms as well as a custom algorithm called PQR2, a type of regression tree that can use any regression model at each leaf node. In addition to application-specific input features, the authors use simple benchmarks to measure CPU speed, memory speed, and disk I/O speed. They visualize the impact of these system characteristics on the overall trend in runtime for their selected applications. Rodrigues *et al.* [21] used similar methods to develop an online memory usage prediction tool on an IBM POWER8 cluster. Interestingly, these authors included several textual features, such as user ID, working directory, and the executed command, as categorical inputs, however they did not specify how much these features contributed to prediction accuracy. Additionally, these authors used a database of jobs executed on their cluster over a period of time, rather than focusing on a few specific applications.

Da Silva *et al.* [5] profiled several Pegasus workflows with the Kickstart profiling tool, explored the relationship between input parameters and the profiling results, and developed a model that uses density-based clustering and regression trees to identify correlations between input parameters in order to predict runtime, memory usage, and disk usage. Additionally, the authors integrated their prediction model into an online prediction tool which continuously estimates the resource usage of tasks in a workflow, updating its predictions with the real usage patterns of tasks as they finish. They provide a clear methodology towards automatic resource prediction; we aim to improve upon their work by focusing on ease-of-use and preventing under-provisioning.

Empirical approaches to resource prediction are powerful because they do not require any knowledge about the internal details of the application or workflow. Generally speaking, only those features that are readily available and easy to obtain, such as input parameters and input data characteristics, are used. Runtime predictions can be improved further by including system metrics based on simple benchmarks (as shown by Matsunaga *et al.*). The main drawback of empirical models is that training data is expensive to acquire. Machine learning models typically need at least $\mathcal{O}(100)$ samples, which means that the application-under-test must be run many times in order to train a sufficiently accurate model. The high cost of acquiring training data can be mitigated by using historical job information, as in the study by Rodrigues *et al.*, since those jobs would have been run anyway. Thus empirical modeling is a promising approach to resource prediction that is portable and requires minimal computational expertise.

III. Tesseract: Intelligent Resource Prediction

In this section, we describe the implementation of Tesseract and how it is used by a domain scientist.

A. Implementation

Tesseract draws inspiration from a number of previous studies, including Ipek *et al.* [15], Matsunaga *et al.* [18], and Da Silva *et al.* [5]. The overall approach of Tesseract is to collect the performance data from past runs of a workflow and then train a regression model to predict the resource usage of future runs. This approach is open-ended enough that it may be able to capture the many sources of variation that contribute to application performance on a heterogeneous computing system.

Tesseract can predict resource usage for any application or workflow that is implemented as a Nextflow pipeline. Nextflow [6] is a workflow manager that provides broad support for many key elements of scientific workflows; it is language-agnostic, portable across many execution environments, highly scalable, and it supports many other useful features such as caching, containerization, and pipeline sharing for reproducibility. Tesseract depends on Nextflow to collect the input features and resource metrics that comprise the training data. Any standalone application can be easily wrapped into a single-step Nextflow pipeline, and any workflow can be refactored into a Nextflow pipeline, although difficulty may vary. Tesseract itself is a collection of Python scripts that use a number of standard libraries for machine learning, including Numpy, Pandas, scikit-learn [19], Tensorflow [1],

and Keras [4], as well as matplotlib [14] and seaborn [27] for visualizations.

To further describe the implementation of Tesseract, we address a number of design questions that must be considered.

Which resources do we need to predict? While Tesseract can target any resource metric that is captured by Nextflow, we are particularly interested in runtime, memory usage, and disk usage, as these resources are the most pertinent to the problem of resource provisioning. Runtime refers to the duration of a job. Memory usage refers to the maximum amount of memory used at one time throughout the duration of a job. Disk usage refers to the total amount of output data written to disk storage by a job. All of these metrics are automatically collected by Nextflow. The number of CPUs is treated as an input rather than an output because it is almost always provided as an input. Applications are generally designed to work within the CPU restraints they are given, whereas memory and disk usage are usually determined by the problem size and other input parameters. If the number of CPUs are restricted, the application can still do the same work but it will simply take longer; on the other hand, if an application requires more memory or disk space than is available, generally the application has no other option than to terminate. In this way we can derive metrics such as CPU-hours from runtime as long as we have the number of CPUs as an input. The same reasoning can be applied to GPUs, FPGAs, and any other such accelerators, which is pertinent to this work since we consider several GPU-enabled applications. It should also be noted that Tesseract predicts the resource usage of individual tasks rather than entire workflow runs, as resource provisioning typically occurs at the level of tasks.

What features can we use as inputs? Previous studies have addressed this question in many different ways. To help guide our exploration, we have devised a set of categories inspired by Guo [12]. We say that a computational experiment has three possible sources of inputs:

- **Code**: the software and its dependencies; compile-time and run-time input parameters
- **Data**: any data that is provided as input to the software; file size, dimensions
- **Environment**: the hardware and operating system on which the software is run; hardware attributes, kernel settings, environment variables, benchmarking results

These three categories form a basis for understanding how to select the right input features for a prediction model. For example, one could include specific command-line parameters (code), input data size (data), and benchmarking metrics for the underlying hardware (environment). Features can be added and removed in each category to match the needs of the particular application. Because input features can come from so many different sources, Tesseract requires the user to manually define the input features for each application.

How detailed does the input data need to be? Some studies have only used input parameters, input data characteristics, and simple benchmarking metrics; others have used much more fine-grained information such as hardware counters and other low-level profiling traces. Based on the results of these studies and our own preliminary results, we have found that the first case provides enough data to achieve sufficiently low prediction error. Since the input features must be specified by the user, the process should be as simple as possible. Including profiling information would require additional profiling tools, which may be unfamiliar to users, or code instrumentation, which is even more prohibitive if the user is not familiar with the source code. On the other hand, input parameters and input data characteristics can be obtained from the job script with few modifications. Benchmarking metrics are somewhat more complicated and are not explored in this work, although we intend to incorporate such metrics in the future.

How accurate do the predictions need to be? Some studies define 20% relative error or less as acceptable [2]. Relative error can be assessed using mean absolute percentage error (MAPE):

$$\text{MAPE } (\%) = 100 \times \sum_{i=1}^{n} \left| \frac{y_{true,i} - y_{pred,i}}{y_{true,i}} \right|$$

While we find this threshold reasonable, any sort of prediction error by itself does not adequately describe the effectiveness of a resource prediction model because the costs of underestimation and overestimation are different. An under-provisioned job will fail and have to be re-run, whereas an over-provisioned job may have some negative effects (i.e. longer queue time, wasted resources) depending on the situation, but will still complete. A good resource prediction model should avoid under-provisioning entirely, even at the expense of some over-provisioning, so long as it is not extreme [1]. We address this issue by using regression models that provide a confidence interval around each point prediction, and using the upper bound as the resource request. We evaluate these intervals using the coverage probability (CP), which is the percentage of intervals that contain the true target value. Given a sufficiently large number of predictions, a 95% confidence interval should provide a coverage probability of at least 95%. This metric will essentially measure a model's ability to avoid both under-provisioning and extreme over-provisioning. For aesthetic consistency with prediction error, we use coverage error, which we define as $CE\ (\%) = 100 - CP$. Achieving at least 95% coverage is equivalent to achieving 5% coverage error or less.

How many training samples are needed? The amount of training data is a serious consideration since training data is expensive to acquire, which is a primary drawback of data-driven resource prediction. Tesseract should be able to achieve an acceptably low prediction error with as few training samples as possible to minimize the cost of acquiring training data. More importantly, however, Tesseract should be able to learn from the runs that users have already performed as part of their normal work. That way, training prediction models will not require any more runs than would have been performed

[1]Under-provisioning is not so much an issue for applications that use checkpointing, however many applications, including our entire test suite, do not use checkpointing.

anyway. For the purpose of this study, we generated our own runs based on actual use cases among our peers, in order to ensure that Tesseract will work on real performance data when used in production.

To summarize, Tesseract is implemented based on the following design principles and goals:

- Predict runtime, memory usage, and disk usage (for output files) of individual workflow tasks;
- Use input features that are fast and easy to acquire, no profiling or code instrumentation;
- Use input parameters, input data characteristics, basic system metrics (in the future);
- Achieve less than 20% relative error on point predictions;
- Achieve less than 5% coverage error when prediction error is high;
- Re-use performance data from historical runs.

B. Usage

Here we describe how a domain scientist would use Tesseract with their scientific workflow. The entire usage cycle can be summarized as follows:

1) User annotates a scientific workflow with trace directives that define the input features for each process in the workflow
2) User runs the workflow many times as part of their normal work
3) Tesseract combines input features from execution logs with the execution traces to produce a performance dataset for each workflow process
4) Tesseract trains a prediction model for each resource metric and each workflow process
5) User employs prediction models to estimate resource usage of future workflow runs

Given a Nextflow pipeline, the user must annotate each process script in the pipeline with "trace directives". A trace directive defines an input feature to be included in the training data, and it is a core idiom used by Tesseract. A trace directive is a print statement that prints the `#TRACE` prefix followed by a key-value pair, which denotes the name and value of an input feature. The key will become a column in the performance dataset, and the value will be saved for each task. Trace directives are evaluated and printed to the execution log during task execution. Since the user may not yet know which inputs will be most important for prediction, it is better to be inclusive rather than exclusive at this stage. Input features can always be discarded later on, but if an input feature is not included as a trace directive, it will be difficult or impossible to recover it later on without rerunning the workflow. In general, trace directives consist of input parameters and simple input data characteristics such as file size, number of lines, or number of rows and columns.

Once a pipeline has been annotated, it must be run many times in order to generate sufficient performance data. These runs should ideally be a part of the user's normal work, or they can be generated for the specific purpose of acquiring training data. In either case, the runs should be representative of real experiments and should span a range of input conditions. Nextflow will produce an execution trace for each task which contains the desired resource metrics. After enough experiments have been run, Tesseract aggregates the input features and resource metrics from all runs and produces a performance dataset for each process in the workflow.

Tesseract creates a prediction model for each resource metric, for each process in the workflow. We refer to a particular workflow / process / resource metric as a "prediction target". A workflow with multiple steps and multiple resource metrics will produce many prediction targets. However, we have found that some prediction targets do not actually require a prediction model because a simpler heuristic can be used. For example, many processes take only a few minutes to execute, or use only a few megabytes of memory, or produce only a few kilobytes of output data (i.e. log files). For these targets, the maximum target value rounded up is sufficient for the purpose of requesting resources. Therefore, Tesseract only trains a model for prediction targets with standard deviation greater than 0.1 hr (in the case of runtime) or 0.1 GB (in the case of memory and disk usage), otherwise it uses the maximum target value rounded up. Tesseract can use any prediction model that implements the scikit-learn Estimator API, but also provides models that can be used out-of-the-box. These models are described in Section IV.

In the final step, the user employs Tesseract to provide resource estimates for future runs. Currently, this step is done by querying Tesseract and manually updating the corresponding resource settings in the Nextflow pipeline. In the future, we would like to integrate Tesseract with Nextflow such that resource estimates are automatically queried and applied when a task is launched.

IV. Experimental Design

In this section, we describe the experiments that were performed to evaluate Tesseract, including the applications and the computing environment that were used.

A. Application Suite

We selected five scientific workflows that are used frequently by fellow researchers at our institution. Using these workflows for development and evaluation will ensure that Tesseract is easy to use for domain scientists, such as our peers, who do not have deep expertise in performance engineering. We briefly describe these workflows here:

- **GEMmaker** is a bioinformatics pipeline for constructing gene expression matrices (GEMs) from Illumina RNA-seq data [13].
- **Gene Oracle** is a pipeline for identifying biologically significant biomarkers, or "candidate genes," from a high-dimensional gene expression matrix (GEM) [25].
- **HemeLB** is a high-performance Lattice Boltzmann code for sparse complex geometries, typically used to study vascular flow [10], [11], [23].

- **KINC** is a bioinformatics pipeline for constructing gene co-expression networks (GCNs) [22].
- **TSPG** is a deep learning pipeline for identifying global gene expression transitions between biological conditions [24].

Every workflow listed above except for HemeLB was already implemented as a Nextflow pipeline. HemeLB is a conventional MPI application, which we have wrapped in a single-step Nextflow pipeline for the purpose of this study, in order to demonstrate the use of Tesseract with traditional MPI applications. Additionally, every workflow except for GEMmaker contains GPU-enabled steps.

We annotated the individual processes in each workflow with trace directives in order to obtain input features, as described in Section III. We generated performance datasets for each workflow by running the workflow many times on a range of parameters and input data, based on the typical usage of these workflows by our peers. For more information on these workflows, including the input features that were defined for the individual workflow processes, we refer the reader to the publications listed above and their corresponding GitHub repositories.

B. Computing Environment

The Palmetto cluster at Clemson University is a condominium cluster with over 20 hardware phases, ranging from older CPU-only nodes to newer nodes equipped with GPU nodes and a high-speed interconnect. The GPU nodes include Tesla K20, K40, P100, and V100 GPUs. All of the performance data for this study was collected on the Palmetto cluster.

C. Resource Prediction

We used Tesseract to predict runtime, memory usage, and disk usage for each workflow in our test suite. This setup produced over 120 prediction targets, 16 of which were selected by Tesseract for model training based on the criteria described in Section III. The selected targets span all five workflows and all three resource types. Nearly all of the excluded targets had resource usage below 1 hr or 1 GB. For each selected prediction target, we trained a neural network and a random forest. The neural network was configured with hidden layer sizes $(128, 128, 128)$, ReLU activation, MAE loss, and the Adam optimizer. Additionally, the neural network uses L2 regularization and dropout in order to facilitate the use of confidence intervals. The neural network was implemented in Keras using the TensorFlow backend. We used the random forest regressor in scikit-learn with 100 decision trees and MAE criterion. Preliminary experiments revealed that these two models consistently outperformed other regression models and neural network architectures. We evaluated each model by performing 5-fold cross validation. Since each fold is used once as the test set, we combined the predictions of each fold in order to compare predicted and actual values for the entire dataset. We evaluated these predictions using MAPE, as described in Section III.

Additionally, we extended the models described above to provide a 95% confidence interval around each point prediction. For the neural network, we enable dropout during inference and take the mean and variance of multiple repeated predictions, as demonstrated by Gal *et al.* [8]. For the random forest, we use the jackknife [26] to obtain a a bias-adjusted variance estimate based on the predictions of the individual decision trees in the random forest. The `forestci` package [20] integrates this approach seamlessly with the random forest regressor in scikit-learn. For both models, the 95% confidence interval is equivalent to the point prediction (mean) +/- two standard deviations. We evaluated the confidence intervals using coverage error, as described in Section III. The coverage error effectively measures the percentage of jobs that would fail when using the upper bound of the confidence interval, rather than the point prediction, for resource provisioning.

V. Results and Discussion

The results of our resource prediction experiments are summarized in Figure 1, which shows the prediction error and coverage error achieved by each model for each prediction target. An effective model should achieve less than 20% MAPE, or, failing that, achieve less than 5% coverage error. While many of the models did not achieve low prediction error, every model except for three achieved sufficiently low coverage error, and for every prediction target there is at least one model with low coverage error. In other words, even when prediction error is high, such as in cases where the training data is sparse or noisy, the prediction model can still protect against under-provisioning by providing sufficiently large confidence intervals. These results demonstrate that Tesseract can predict resource usage across a diverse set of workflows, using only basic input features, for the purpose of provisioning resources for jobs. The neural network and random forest are evenly matched overall, with each model providing better performance over the other in different cases. In practice, the user could use one or the other across the board, or use both and have Tesseract choose the better model in each case.

Additionally, Figure 2 provides more detailed results for three prediction targets, one for each of runtime, memory usage, and disk usage. These scatter plots are merely a visual sample in lieu of displaying detailed results for all 16 prediction targets. In general, these plots are useful because they provide context for the error metrics at the level of individual predictions. For example, in the case of KINC / similarity_mpi runtime, the scatter plots show that most of the predictions are very close to the true target value, even though both of these models had more than 100% MAPE. This discrepancy is caused by the fact that some of the runs for this process had unusually long runtimes compared to similar runs; in other words, the training data contained a few anomalies. In the future, we intend to detect these kinds anomalies automatically by checking whether the actual resource usage is outside the confidence interval of the predicted usage, Tesseract could then alert the user to review the job for errors.

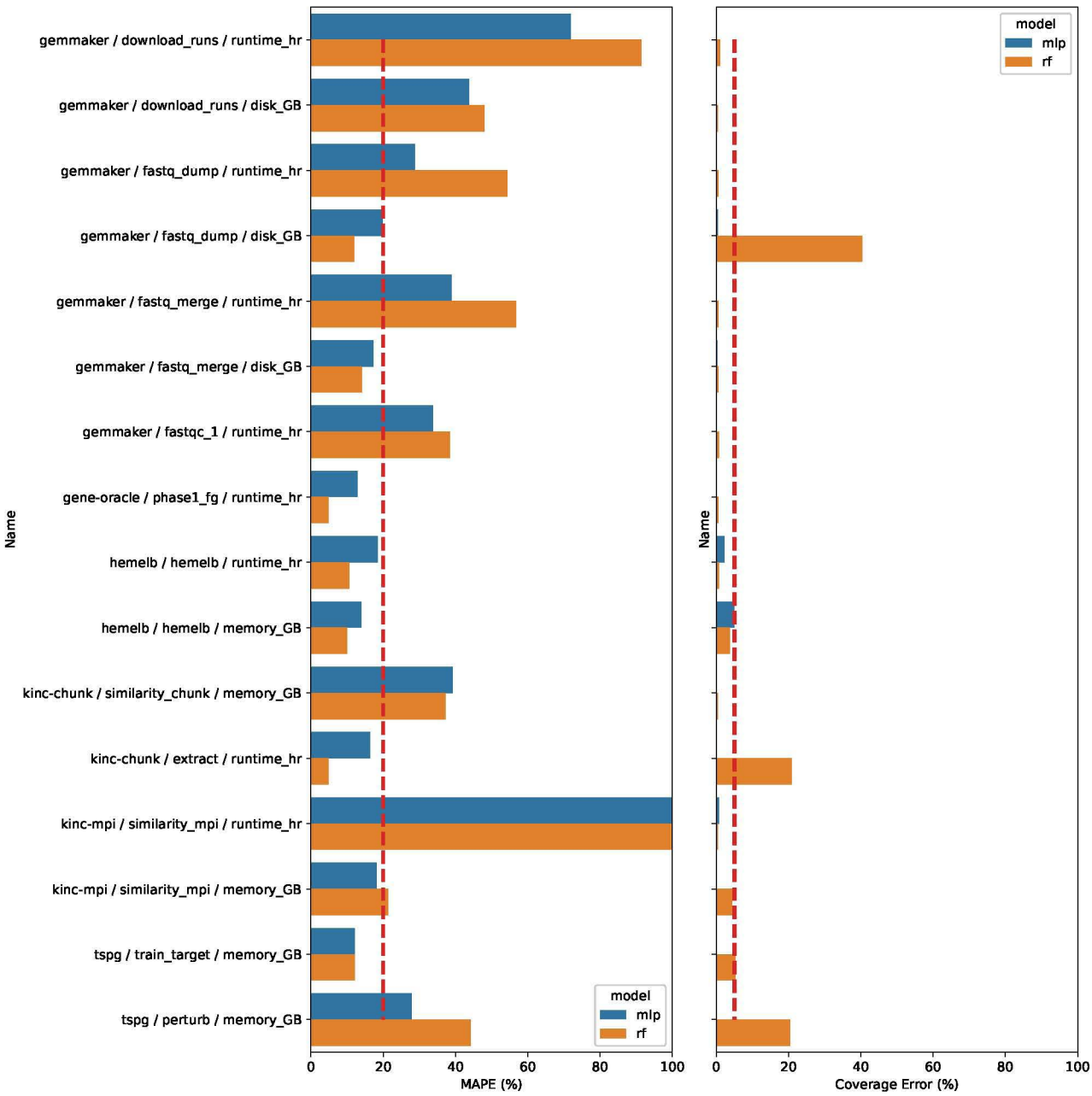
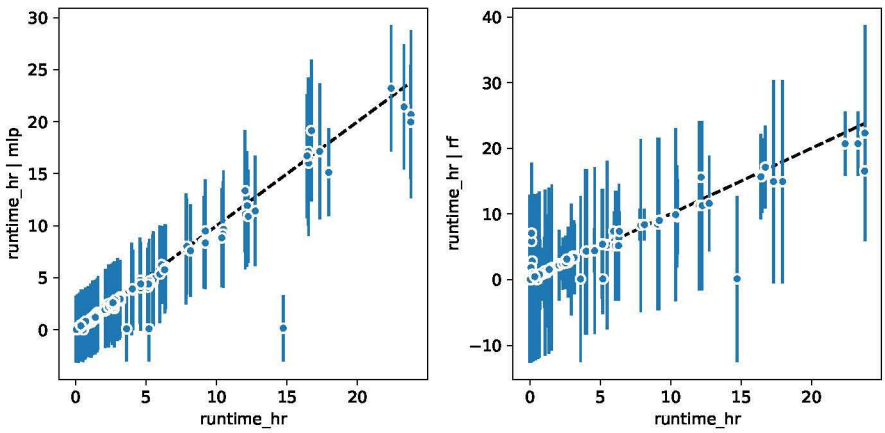
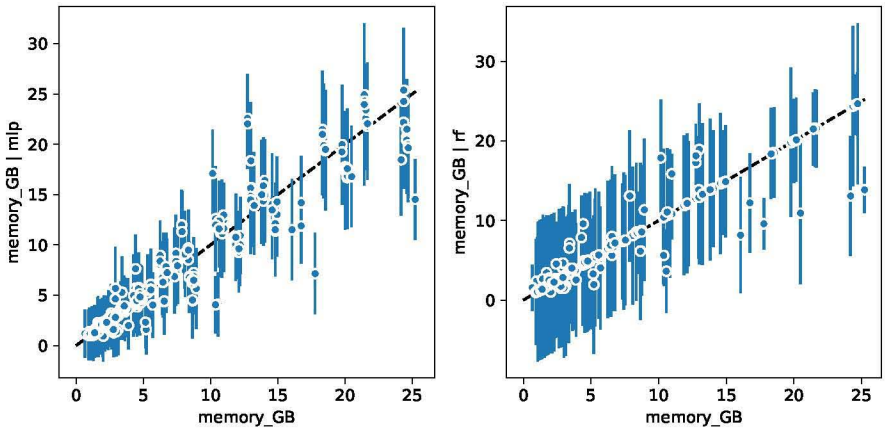


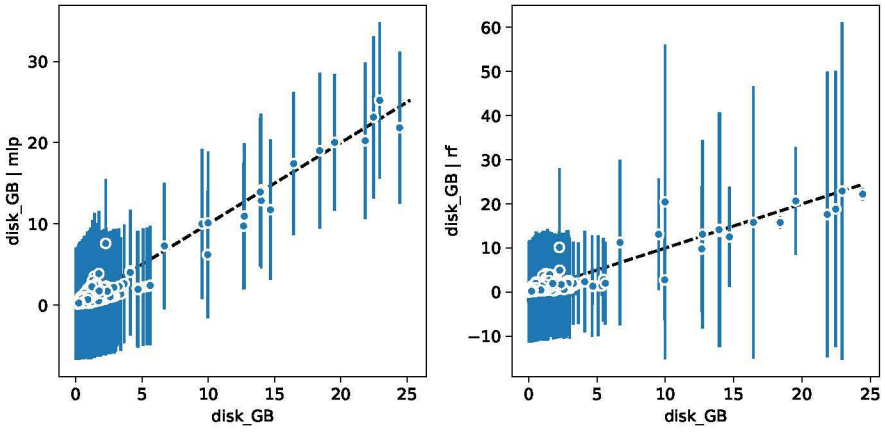
Fig. 1: Summary of results for the 16 selected prediction targets. The left-hand panel shows mean absolute percentage error (MAPE), with 20% MAPE denoted by the dashed red line. The right-hand panel shows coverage error, with 5% coverage error denoted by the dashed red line.



(a) KINC / similarity_mpi runtime



(b) HemeLB memory usage



(c) GEMmaker / download_runs disk usage

Fig. 2: Expected vs predicted target values for three prediction targets, one example for each of runtime, memory usage, and disk usage. In each case, the black dashed line denotes equality, each blue point and vertical bar is a point prediction with corresponding 95% confidence interval.

While we performed these experiments on a single computing platform, Tesseract can be used on any platform that is supported by Nextflow. In the future we would like to evaluate Tesseract on other platforms, including cloud platforms, as well as use system metrics as input features in order to predict runtime across multiple platforms with a single model.

VI. Conclusions

We presented Tesseract, a tool for predicting resource usage of scientific applications and workflows, using a methodology that is generic across applications and platforms and requires little computational expertise. We laid out the history of resource prediction, underscoring the need for tools that are generic and easy to use. We demonstrated how Tesseract addresses this need by describing how the tool is used and by evaluating the tool on a diverse set of scientific workflows. In particular, we show how Tesseract prevents underprovisioning by providing confidence intervals. The source code for Tesseract, including the scripts that were used to generate performance data for this study, is available at https://github.com/bentsherman/tesseract under the MIT license.

Acknowledgment

This work was supported by National Science Foundation Award #1659300 "CC*Data: National Cyberinfrastructure for Scientific Data Analysis at Scale (SciDAS)". Additionally, Clemson University is acknowledged for generous allotment of compute time on the Palmetto cluster.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[2] Laura Carrington, Allan Snavely, and Nicole Wolter. A performance prediction framework for scientific applications. *Future Generation Computer Systems*, 22(3):336–346, 2006.

[3] Siddhartha Chatterjee, Erin Parker, Philip J Hanlon, and Alvin R Lebeck. Exact analysis of the cache behavior of nested loops. *ACM SIGPLAN Notices*, 36(5):286–297, 2001.

[4] François Chollet et al. Keras. https://keras.io, 2015.

[5] Rafael Ferreira Da Silva, Gideon Juve, Mats Rynge, Ewa Deelman, and Miron Livny. Online task resource consumption prediction for scientific workflows. *Parallel Processing Letters*, 25(03):1541003, 2015.

[6] Paolo Di Tommaso, Maria Chatzou, Evan W Floden, Pablo Prieto Barja, Emilio Palumbo, and Cedric Notredame. Nextflow enables reproducible computational workflows. *Nature biotechnology*, 35(4):316–319, 2017.

[7] Ian T Foster, Brian Toonen, and Patrick H Worley. Performance of massively parallel computers for spectral atmospheric models. *Journal of Atmospheric and Oceanic Technology*, 13(5):1031–1045, 1996.

[8] Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*, pages 1050–1059. PMLR, 2016.

[9] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(4):703–746, 1999.

[10] Derek Groen, David Abou Chacra, Rupert W. Nash, Jirí Jaros, Miguel O. Bernabeu, and Peter V. Coveney. Weighted decomposition in high-performance lattice-boltzmann simulations: are some lattice sites more equal than others?, 2014.

[11] Derek Groen, James Hetherington, Hywel B Carver, Rupert W Nash, Miguel O Bernabeu, and Peter V Coveney. Analysing and modelling the performance of the hemelb lattice-boltzmann simulation environment. *Journal of Computational Science*, 4(5):412–422, 2013.

[12] Philip Guo. Cde: A tool for creating portable experimental software packages. *Computing in Science & Engineering*, 14(4):32–35, 2012.

[13] John Hadish, Tyler Biggs, Ben Shealy, Connor Wytko, Sai Prudhvi Oruganti, F. Alex Feltus, and Stephen Ficklin. Systemsgenetics/gemmaker: Release v1.1, January 2020.

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

[15] Engin Ipek, Bronis R De Supinski, Martin Schulz, and Sally A McKee. An approach to performance prediction for parallel applications. In *European Conference on Parallel Processing*, pages 196–205. Springer, 2005.

[16] Tejas S Karkhanis and James E Smith. A first-order superscalar processor model. In *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.*, pages 338–349. IEEE, 2004.

[17] Darren J Kerbyson, Henry J Alme, Adolfy Hoisie, Fabrizio Petrini, Harvey J Wasserman, and Mike Gittings. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, pages 37–37, 2001.

[18] Andréa Matsunaga and José AB Fortes. On the use of machine learning to predict the time and resources consumed by applications. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 495–504. IEEE Computer Society, 2010.

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

[20] Kivan Polimis, Ariel Rokem, and Bryna Hazelton. Confidence intervals for random forests in python. *Journal of Open Source Software*, 2(1), 2017.

[21] Eduardo R Rodrigues, Renato LF Cunha, Marco AS Netto, and Michael Spriggs. Helping hpc users specify job memory requirements via machine learning. In *2016 Third International Workshop on HPC User Support Tools (HUST)*, pages 6–13. IEEE, 2016.

[22] Benjamin T Shealy, Josh JR Burns, Melissa C Smith, F Alex Feltus, and Stephen P Ficklin. Gpu implementation of pairwise gaussian mixture models for multi-modal gene co-expression networks. *IEEE Access*, 7:160845–160857, 2019.

[23] Benjamin T Shealy, Mehrdad Yousefi, Ashwin T Srinath, Melissa C Smith, and Ulf D Schiller. Gpu acceleration of the hemelb code for lattice boltzmann simulations in sparse complex geometries. *IEEE Access*, 9:61224–61236, 2021.

[24] Colin Targonski, M Reed Bender, Benjamin T Shealy, Benafsh Husain, Bill Paseman, Melissa C Smith, and F Alex Feltus. Cellular state transformations using deep learning for precision medicine applications. *Patterns*, page 100087, 2020.

[25] Colin A Targonski, Courtney A Shearer, Benjamin T Shealy, Melissa C Smith, and F Alex Feltus. Uncovering biomarker genes with enriched classification potential from hallmark gene sets. *Scientific reports*, 9(1):1–10, 2019.

[26] Stefan Wager, Trevor Hastie, and Bradley Efron. Confidence intervals for random forests: The jackknife and the infinitesimal jackknife. *The Journal of Machine Learning Research*, 15(1):1625–1651, 2014.

[27] Michael Waskom and the seaborn development team. mwaskom/seaborn, September 2020.