

Portable and Reusable Deep Learning Infrastructure with Containers to Accelerate Cancer Studies

George F. Zaki,¹ Justin M. Wozniak,² Jonathan Ozik,³
Nicholson Collier,³ Thomas Brettin,⁴ and Rick Stevens⁵

¹ Frederick National Laboratory for Cancer Research, Frederick, MD

² Data Science and Learning, Argonne National Laboratory and University of Chicago

³ Decision and Infrastructure Sciences, Argonne National Laboratory and University of Chicago

⁴ Computing, Environment, and Life Sciences, Argonne National Laboratory

⁵ Computing, Environment, and Life Sciences, Argonne National Laboratory and University of Chicago

Abstract—Advanced programming models, domain specific languages, and scripting toolkits have the potential to greatly accelerate the adoption of high performance computing. These complex software systems, however, are often difficult to install and maintain, especially on exotic high-end systems. We consider deep learning workflows used on petascale systems and redeployment on research clusters using containers. Containers are used to deploy the MPI-based infrastructure, but challenges in efficiency, usability, and complexity must be overcome. In this work, we address these challenges through enhancements to a unified workflow system that manages interaction with the container abstraction, the cluster scheduler, and the programming tools. We also report results from running the application on our system, harnessing 298 TFLOPS (single precision).

I. INTRODUCTION

High performance computing (HPC) is being made more accessible through the development of high-level programming tools and environments. These allow a wider range of scientific experts to benefit from continued growth in computing capabilities, including many-core concurrency, accelerators including GPUs, and so on. Additionally, the emergence of scripting frameworks backed by efficient numerical libraries, analysis packages, and communication frameworks has made the rapid development of performant scientific applications a reality. This is in addition to the availability of deep learning (DL) toolkits programmed through very high level Python APIs and deployed on powerful GPUs.

These advances are due in part to the adoption of hierarchical programming, where end users develop applications in Python or R, but the performance-critical sections of code are developed in C, C++, Fortran, and accelerator programming systems. This naturally allows for separation of concerns, where the applications, scripting APIs, and numerics are programmed by different groups of experts, resulting in a more complex computational ecosystem and community. This approach has been pioneered by hierarchical “glue code” systems like Tcl [13] and Java JNI, although with more modest adoption in scientific computing.

Getting started with these complex programming environments is much more challenging than, say, monolithic use of `cc` or `f77`. Thus, we introduce an additional separation of concerns between the scientific user and the programming

model and middleware layer. The latter contains the programming tools and scripting environment, along with the numeric or analytic frameworks required for the users. In this work, we address this with containers. Typically, containers are used to deploy whole applications, but we address a more complex problem of delivering a programming environment along with scheduler interaction and the ability to manage data and deep learning models outside the container. We treat this problem as a separate concern to be managed.

Deep learning at scale has the capability to greatly enhance cancer research. In this work, we used the CANcer Deep Learning Environment (CANDLE) application suite as a reference set of small but representative application workflows that address three key topics in cancer studies. These CANDLE applications have been deployed on petascale machines as individual benchmarks and as scalable workflows using the infrastructure provided by the workflow manager CANDLE/Supervisor [17]. This system provides a very high level programming model but is deployable on the largest available computing resources. In this work, we investigate deploying this infrastructure on modest cluster resources, thus delivering it closer to scientific users. Additionally, the containers may be deployed again on emerging supercomputers that support containers, such as OLCF Summit [5].

Contributions. This paper offers the following: **1)** A description of cancer research workflows that pervasively use deep learning; **2)** An approach to deliver these workflows on a commodity cluster and allow for user extensions and programming; and **3)** performance behavior results from running representative workflows.

The remainder of this paper is organized as follows. In §II, we describe our cancer workflows in more detail, the available parallelism in these workflows and the use of HPC resources. In §III, we describe the programming model and architecture used to integrate these workflows on these systems. In §IV, we provide a performance evaluation of the complete system. In §V, we describe future work, and in §VI, we offer concluding comments.

II. HYPERPARAMETER SEARCH IN DEEP LEARNING

In this section, we provide background on the cancer application suite we are using and the hyperparameter optimization workflows we are running for it.

A. CANDLE application benchmarks

The CANDLE application benchmarks are a suite of Python and Keras-based applications that are designed to both 1) promote the application of deep learning to cancer problems today and 2) help prepare cancer applications for the exascale era. The three “Pilot” areas of CANDLE applications are:

- 1) Analysis of molecular dynamics simulation outputs in the RAS pathway;
- 2) Drug response prediction based on the patient genetic sequence; and
- 3) Determination of optimal cancer treatment strategies via clinical text document analysis.

Each Pilot area has a small number of Python applications. The training and validation data is available for download as specified in each application. The applications use a common set of Python-based utilities, and implement a common interface for use by the higher-level workflows described below.

To support the use cases described above, we developed the CANDLE/Supervisor architecture [17] diagrammed in Figure 1. The overall goal is to solve the hyperparameter optimization problem to minimize $F(p)$, where F is the performance of the neural network parameterized by $p \in P$, where P is the space of valid parameters.

The optimization is controlled by an **Algorithm** ① selected by the user. The Algorithm can be selected from those previously integrated into CANDLE, or new ones can be added. These can be nearly any conceivable model exploration (ME) algorithm that can be integrated with the **EMEWS** ③ software framework. EMEWS [14] enables the user to plug in ME algorithms into a workflow for arbitrary model exploration; optimization is a key use case. This is implemented in a reusable way by connecting the parameter generating ME algorithm and output registration methods to interprocess communication mechanisms that allow these values to be exchanged with Swift/T. EMEWS currently provides this high-level queue-like interface in two implementations: EQ/Py and EQ/R (EMEWS Queues for Python and R). Thus, the ME algorithm can be expressed in Python or R. The Algorithm is run on a thread on one of the processors in the system. It is controlled by a Swift/T script ② provided by EMEWS, that obtains parameter tuples to sample and distributes them for evaluation.

The Swift/T [6], [16] workflow system is used to manage the overall workflow. It integrates with the various HPC schedulers to bring up an allocation. A Swift/T run deploys one or more load balancers and many worker processes distributed across compute nodes in a configurable manner. Normally, Swift/T evaluates a workflow script and distributes the resulting work units for execution across the nodes of a computer system over MPI. Swift/T can launch jobs in a variety of ways, including

in-memory Python functions in a bundled Python interpreter, shell commands, or even MPI-based parallel tasks. However, in this use case, workflow control is delegated to the Algorithm via the EMEWS framework, which provides the Swift/T script.

During an optimization iteration, the Algorithm produces a list of parameter tuples ④ that are encoded as arguments to a Python-based **Wrapper** script ⑤. These wrapper scripts are the interfaces to the various CANDLE Pilot applications. The parameters are encoded in JavaScript Object Notation (JSON) format which can be easily converted by the Python **Wrapper** script into a Python dictionary, from which a CANDLE Pilot application can retrieve the parameter values. These scripts are run concurrently across the available nodes of the Swift/T allocation, typically one per node. Thus, the deep learning software (**DL**), the underlying learning engine (e.g., TensorFlow), has access to all the resources on the node. The **Pilots** are Python programs that implement the application-level logic of the cancer problem in question. They use the **Keras** interface to interact with the DL and are coded to enable the hyperparameters to be inferred from a suitable default model file, or to be overwritten from the command line. It is this construction that allows the parameter tuples to be easily ingested by the respective Pilots, and use a standardized interface developed as part of the project.

The result of a Wrapper execution is a performance measure on the parameter tuple p , typically the validation loss. Other metrics could be used, including training time or some combination thereof. These are fed back to the Algorithm by EMEWS to produce additional parameters to sample. The results are also written to a Solr-based **Metadata Store** ⑦, which contains information about the Wrapper execution. The Metadata Store accesses are triggered by Keras callback functions, which allow Wrapper code to be invoked by Keras at regular intervals. Thus, a progress history is available for each learning trial run, as well as for the overall optimization workflow. Good models can also be selected and written to a **Model Store**.

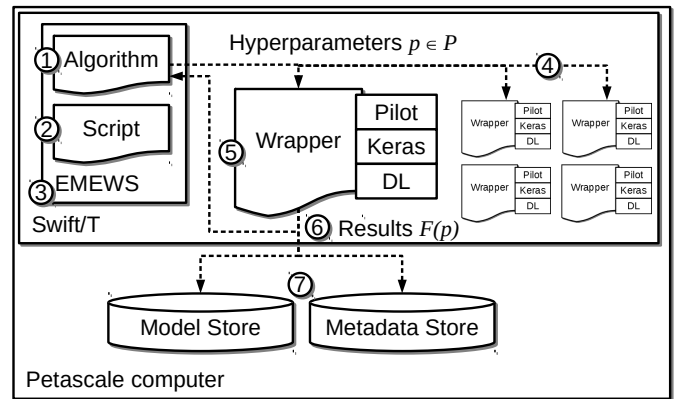


Fig. 1: CANDLE/Supervisor original architecture.

B. Model-based Optimization

Model-based optimization (MBO) approaches are used for tackling expensive black-box model optimization by approximating the model’s objective function through a surrogate regression model and then optimizing over the surrogate model’s response surface. In this work we used the mlrMBO [8] R package to implement MBO algorithms and integrated it into our MBO workflow using the EMEWS Queues for R (EQ/R) capabilities of EMEWS [14]. The mlrMBO package is designed for optimization problems with mixed continuous, categorical and conditional parameters. It follows a Bayesian optimization [7] approach which proceeds as follows. In the initialization phase, n_s configurations are sampled at random, evaluated, and a surrogate model M is fitted with the input-output pairs. In the iterative phase, at each iteration, n_b promising input configurations are sampled using the model M . These configurations are obtained using an infill criterion that guides the optimization and tries to trade-off exploitation and exploration. The infill criterion selects configurations that either have a good expected objective value (exploitation) or high potential to improve the quality of the model M (exploration). The algorithm terminates when a user-defined maximum number of evaluations and/or wall-clock time is exhausted. Crucial to the effectiveness of mlrMBO is the choice of the algorithm used to fit M and the infill criterion. Given the mixed integer parameters in the hyperparameter search, we used random forest [9] because it can handle such parameters directly, without the need to encode the categorical parameters as numeric. For the infill criterion, we used the qLCB [11], which proposes multiple points with varying degrees of exploration and exploitation.

C. Asynchronous Search

The standard CANDLE/Supervisor workflow pattern involves sending batches of parameter combinations to be evaluated in synchronized stages. The asynchronous search (AS) workflow modifies this pattern and, instead, allows an optimizer to continuously update as new results are learned and to generate new hyperparameter combinations based on the new information. This improves computational resource utilization since there are no global synchronization barriers. The AS workflow contains specific optimizer and model agnostic framework code and is implemented using the Scikit-Optimize Python package [4], EMEWS Queues for Python (EQ/Py), and the MPI for Python (mpi4py) package [2]. The base AS workflow also uses a random forest classifier to create a response surface for optimizing the model hyperparameters. Thus, similar to the mlrMBO workflow, it implements an MBO algorithm, but the ME algorithm is in Python and the surrogate model is updated asynchronously. EQ/Py provides the interface for receiving parameters from the AS optimizer, while mpi4py is used to communicate results from the model runs directly back to the optimizer.

D. Parameter Space Gridding

A parameter space can be sampled using *a priori* selected parameter combinations. Unlike MBO and asynchronous search, this approach does not require adaptive algorithms to guide the parameter space sampling. This is the simplest, though most costly, approach for hyperparameter optimization. While we include this approach as a comparative base-case, we note that there are two primary drawbacks to utilizing an user-specified set of discrete hyperparameters for reducing loss: 1) it requires the user to make assumptions concerning topological efficiencies and efficacies and 2) it is limited to a small, finite set of models (i.e., it is forcing a complex algorithm into constrained bounds). The *a priori* sampling workflows are implemented using the unrolled parameter file EMEWS capabilities.

III. ARCHITECTURE FOR WORKFLOW PORTABILITY

To run the workflows described above on the Biowulf cluster at National Institute of Health, we had to re-architect the workflow to use a container-oriented paradigm. This gave us the opportunity to re-architect our HPC-oriented scripts into a package that is more usable on modern commodity clusters that offer support for containers.

Biowulf is a heterogeneous cluster with over 95,000 cores with Intel hyperthreaded processors (e.g., Xeon E5 and E7 family) and Nvidia GPU accelerators (e.g., K80, P100, and V100). Based on the application, Biowulf nodes have different number of CPUs, and RAM (e.g., 64GB-3TB). All nodes in the cluster have access to an NFS storage.

The cluster can be used for embarrassingly parallel application as well as HPC applications where parallel file I/O and communication loads are significant compared to the application runtime.

In deep learning applications, the hyperparameter tuning using MBO requires relatively little communication between the hyperparameter server and the workers. However, for a given value of hyperparameters, data and model parallelism requires extensive communication between workers. Biowulf has heterogeneous network infrastructure (e.g., 10G, and 56G FDR Infiniband) so it supports these two requirements.

A single SLURM scheduler provides access to the cluster via a queue. All nodes are accessible via this scheduler, but only limited number of GPUs nodes may be allocated at a time for a given user. Currently, this number is 48 P100 GPUs. In our experimental runs, we ran on at most 32 nodes. At 9.3 TFLOPS single precision per GPU [3], this gave us access to 298 TFLOPS.

Since the CANDLE workflows are designed to run in a plain Linux-based HPC programming environment, they needed to be extensively modified. The main distinction is that some of the scripts run outside the container, and some run inside the container. This made for some challenging rearchitecture.

A. Benefits of containers

Containers are a Linux kernel feature to ease system administration. They allow for, effectively, one or more entire

operating system (OS) instances to be isolated from the main Linux OS. The root filesystem is packaged in an image file, however, the original filesystem can be mounted in the container. Other resources in the container are isolated from each other. At a very high conceptual level, containers are like virtual machines (VMs), but are implemented quite differently, as an OS feature that multiplexes itself. Containers typically start up faster than VMs and have less overhead at run time [10].

Containers have multiple general benefits. First, containers minimize or eliminate software installation and configuration complexities for end users. Containers allow for users to easily switch between software versions, such as Python 2 and Python 3 packages, without the risk of incompatibilities. This allows users to try new, untested environments without losing the productivity in their older, well-tested environments.

For CANDLE users, these benefits are very important. CANDLE workflows are a shared project between users interested in basic cancer research, deep learning research, and exascale computing. Each subgroup has differing interests in the value of experimental changes to different parts of the software stack. For example, users of CANDLE at the National Cancer Institute and Frederick National Laboratory need a more production-ready system to address more realistic problems in the health sciences, and are less interested in exotic changes needed to run on the latest petascale system. So containers allow us to easily and reliably package the CANDLE workflows for this use case.

Giving the mentioned advantages, using container technologies had increased popularity in the scientific community. However, unlike many web applications that benefit from enterprise micro-service virtualization, security and scalability are first class requirements in HPC platforms. In that context, we have used Singularity [12] a container solution developed for scientific application. Singularity offers the mobility, reproducibility, and user freedom of containers while it is able to support existing HPC resources.

B. Challenges of containers

While container technologies allow enhanced productivity in dynamic environments and reduce dependencies on system administrators, they need careful attention to achieve that goal. We highlight a few of these challenges.

First, container construction and maintenance requires careful attention to software versions and compatibilities. Containers allow this problem to be delegated to the software managers on a team, with minimal interference to system administrators and application-level users. As a problematic example, the container could be based on an operating system distribution using the tag *latest*. While such practice might help in using the most up to date version, it can break dependencies down the road.

On the other hand, in the container definition file or recipe, if the version of a given package is specified, then all dependent packages have to be compatible. For example, the recipe cannot contain the versions of some python packages while

omitting the others. When the version is not specified, the python package manager might install the latest version of a module which can break compatibility.

Second, to run MPI applications using Singularity, the MPI implementation inside the container must match the implementation on the host operating system. This step might require users to rebuild the container from the recipe instead of using the already built image. Our approach to mitigate this overhead is described in the evaluation section.

Third, the portability benefit of using containers can also come at the expense of limited customization and optimization of software packages for a given processing architecture. For example, one cluster might need optimized TensorFlow installation for Intel Xeon Phi accelerators, while another cluster needs optimized CUDA installation for their Volta GPUs. Providing one container with all possible optimized versions is labor intensive. However, such optimized installation has to be provided for the software packages that consume most of the time of the workflow, and for the base operating system used in the container only.

C. The container architecture for CANDLE workflows

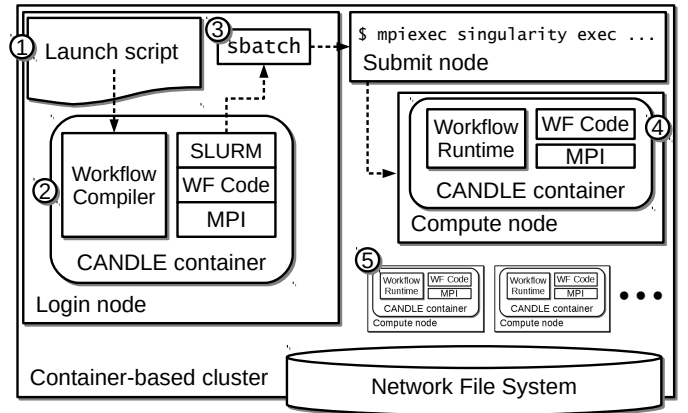


Fig. 2: Container usage for CANDLE workflows.

Our container-based workflow startup infrastructure is shown in Figure 2. We bundled all the requisite software, including the CANDLE Benchmarks, Supervisor scripts, Python, R, Keras, and Swift/T into a “CANDLE container.” The container allows the user to launch it interactively to make any necessary edits. The system is then run by the user invoking a launch script (1) that specifies the workflow expressed in Swift and the settings for the run, including the number of nodes requested, queue to use, and so on. This invokes a container in which Swift/T is installed (2). The component shown as “Workflow Compiler” uses Swift/T to translate the Swift workflow into runnable format, shown as the “WF Code”, as well as generate the SLURM submit script and MPI runtime settings. This process takes about 1 second. Then, the user must use `sbatch` to launch the generated workflow scripts (3).

Once the submitted job starts, it uses `mpiexec` to start the CANDLE container on each allocated compute node. Within this container, the Swift/T “Workflow Runtime,” an MPI program, is started. It is able to connect to other processes that are part of this MPI job (5). The main CANDLE application workflow is then able to start.

IV. EVALUATION

A. Building the CANDLE container

The singularity definition file for CANDLE includes the recipe for all the required packages to run the parameter optimization workflows and the machine learning models [1]. In singularity, the MPI implementation in the container image must match the implementation on the host cluster. On Biowulf, multiple MPI implementations are installed. At runtime, the installation that matches the CANDLE container is loaded.

Building the CANDLE container for other clusters will require making sure the correct MPI implementation is used in the container recipe. In practice, the system admins for the HPC cluster provide a template for a simple MPI application running in the container environment for that cluster. This template is then used to configure the MPI section in the CANDLE recipe.

Per singularity requirement, the CANDLE container is built on a virtual machine (VM) where we have root access, then the container single file is copied to the Biowulf cluster for end users. Productivity of using containers is achieved by running the workflow on the VM as well as on the cluster without modification. The current CANDLE container size is about 1.9GB. As mentioned in the introduction, the actual deep learning models are shipped separately.

B. Using the CANDLE container

Once the singularity image is copied to the production system (e.g., Biowulf) end users have to focus only on defining the configuration of their experiments without dealing with software installation. To set up a deep learning experiment using the CANDLE container, health science researchers should define the following:

- The deep learning model. The script that describes the model should accept the value of the model hyperparameters as input arguments and print the score of the model after training.
- The definition of the hyperparameter search space. Depending on the optimization algorithm, the hyperparameters space can be enumerated for grid search, or the range of every parameter can be described using the API provided by the optimization package (e.g., `mlrMBO`, `Scikit-Optimize`).
- The compute job. Users have to define the number and type of processing nodes, the job time, memory per node, and any special queue for the schedule, etc.

Once the job is defined, it is submitted to the cluster using a script that wraps the steps described in Section III.

C. The U-Net application

To evaluate the usability and scalability of the workflow, we developed a benchmark that shows how to use the container to train deep learning problems in addition to the examples mentioned in Section II. In that context, we trained a popular image segmentation DL network U-Net [15]. U-Net has an encoder/decoder architecture where its input is a 2-D image and its output is a 2-D array of the same size of the input. Every pixel in the output array indicates a classification as foreground or background for the corresponding pixel in the input.

U-Net is expressed in Keras using the TensorFlow backend. The hyperparameters for the network consist of the number of layers in the encoder/decoder stacks, the number of convolutional filters, the size of the convolution filters, the dropout value, and the activation functions.

The three hyperparameter optimization algorithms: random grid search, `mlrMBO`, and asynchronous search were evaluated. For random grid search, continuous and integer parameters (e.g., dropout, number of layers) were sampled to discrete values and the evaluation took place using a random order of the candidate configurations.

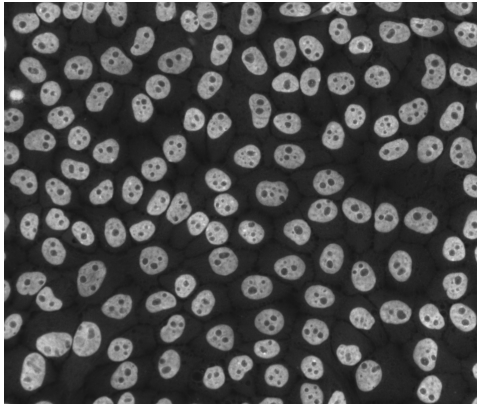
The U-Net application is used in a workflow to perform image segmentation for the nuclei in images generated from fluorescent microscopy. This is a necessary first step for many image processing pipelines for cancer cell analysis. The algorithm is required to perform with varying distribution of pixel intensities, signal to noise ratios, different cell types, and most challenging overlapping cell where there no clear background signal that separates two different cells. Examples for input and output for the segmentation workflow is shown in Figure 3.

D. Performance results

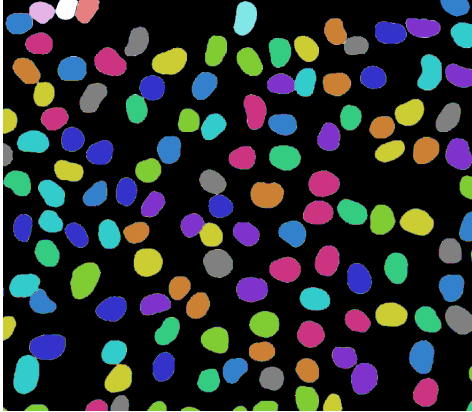
Figures 4-6 show Gantt charts for exploring the hyperparameter space on eight Biowulf’s P100 GPU nodes. The vertical line in the Gantt chart corresponds to every worker. If two configurations are assigned to the same worker, the evaluations are given contrasting colors for display purpose. To test the workflow, every hyperparameter evaluation was capped to run for 2 epochs, the total job runtime was set for 2 hours, and the number of configurations was capped at 1000.

Asynchronous search and random grid search have an overall good utilization of GPUs, while `mlrMBO` imposes barrier synchronization after every iteration which reduces the throughput and utilization. However, it is also important to consider the quality of the generated model hyperparameters, which is out of the scope of this paper.

Figure 7 shows scaling of the workflow using the `mlrMBO` optimization from 1 to 32 nodes. Using 1 or 2 nodes, the evaluation did not get past the initialization. By increasing the number of nodes, more `mlrMBO` iterations are evaluated. This however takes place at the expense of dropped throughput per node as the `mlrMBO` barrier synchronization contributes to relatively large idle time. In the scaling results shown in figure 7, the average idle times (i.e., the percentage of time



(a) Microscopy image of nuclei



(b) Ground truth for nuclei semantic segmentation

Fig. 3: Example of U-Net input and output

Workers:	1	2	4	8	16	32
Idle time:	0.01%	0.2%	4%	6%	27%	44%

TABLE I: Idle times for increasing worker node counts.

the workers are not processing any evaluation) are shown in Table I.

At 32 nodes, a great deal of time is wasted on workers that complete early and must wait for the synchronization before the new hyperparameters are ready to run. This utilization gap can be addressed by running more samples per iteration, allowing the Swift/T load balancer to add work to idle workers. It can also be addressed by the asynchronous approach in §II-C which shows good per node throughput.

E. Native versus containerized performance

To evaluate the performance of the CANDLE container, we installed the packages needed to run the basic EMEWS workflow on Biowulf. We set an experiment to run the grid search hyperparameter evaluation with a dummy model that imports the Keras deep learning package and exit. Running the MPI job using 2 workers for 5 minutes produced 891 dummy evaluations using native installation and 869 dummy evaluations using the CANDLE container. This result shows near perfect scaling and less than a second overhead to run one evaluation. This overhead is minimal compared to the time a

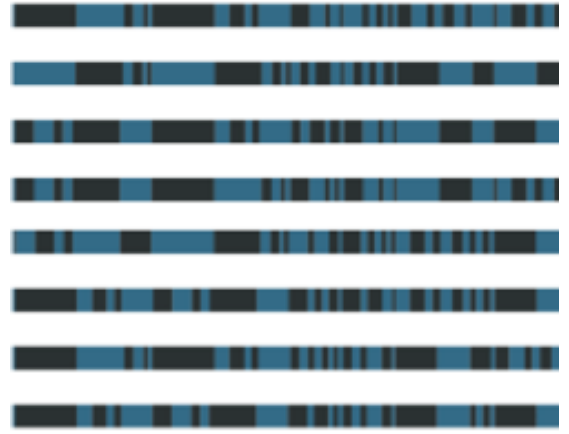


Fig. 4: Random grid search Gantt chart using 8 GPUS on 8 nodes.

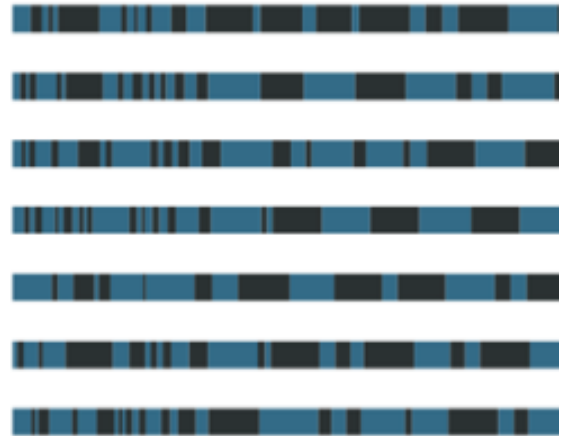


Fig. 5: Asynchronous search Gantt chart using 8 GPUS, 8 nodes.

typical deep learning training task takes (tens of minutes to hours).

V. FUTURE WORK

The next step for this work is to deploy this approach on supercomputers that support containers like Summit. If successful, we will monitor the adoption of containers in other supercomputing centers. If containers become ubiquitous, we will be left with the question of whether to continue support for workflows without containers, and what the maintainability, usability, and portability of such workflows may be.

VI. CONCLUSION

Containers can greatly enhance the ability of users to access complex software applications and workflows. In this paper, we used a container to simplify the deployment of a complex cancer-based application suite called CANDLE for scientific users. This system combines multiple scripting languages, an MPI-based workflow system, large Python and R libraries

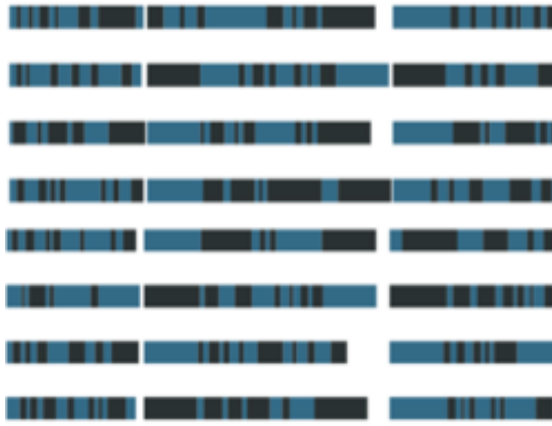


Fig. 6: mlrMBO Gantt chart using 8 GPUS, 8 nodes.

containing TensorFlow, optimizers, and so on, and Python-based application benchmarks. The workflow was originally designed to run on petascale DOE supercomputers, but we used containers to redeploy the system on a smaller cluster commonly used by health scientists.

In this paper, we described the motivation for the use of containers by this application workflow. We described the workflow itself and the benefits gained by the incorporation of a container-based solution. We showed the behavior of the complete system running real cancer workflows on the Biowulf cluster, delivering 298 TFLOPS (single precision) to the deep learning modules.

We developed a container-based middleware that allows a workflow developed for petascale systems to be deployed on a terascale cluster, without loss of performance or usability. The programming model used is the Swift/T workflow language, a scalable, MPI-based dataflow language. This system allows cancer researchers to benefit from a heterogeneous cluster, running a workflow with multiple layers of concurrency. In short, this paper demonstrates that containers are a viable approach to broaden the use of advanced programming models developed for extreme scale systems.

ACKNOWLEDGMENTS

This material is based upon work supported by the U.S. Department of Energy, Office of Science, under contract number DE-AC02-06CH11357. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

This work utilized the computational resources of the NIH HPC Biowulf cluster. (<http://hpc.nih.gov>)

REFERENCES

- [1] Candle-Distribution. <https://github.com/ECP-CANDLE/Distribution>.
- [2] MPI for Python. <https://mpi4py.readthedocs.io>.
- [3] Nvidia Tesla P100 data sheet. <https://www.nvidia.com/en-us/data-center/tesla-p100>.
- [4] Scikit-Optimize. <https://scikit-optimize.github.io>.
- [5] Summit. <https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit>.
- [6] Timothy G. Armstrong, Justin M. Wozniak, Michael Wilde, and Ian T. Foster. Compiler techniques for massively scalable implicit task parallelism. In *Proc. SC*, 2014.
- [7] James S. Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems*, pages 2546–2554, 2011.
- [8] Bernd Bischl, Jakob Richter, Jakob Bossek, Daniel Horn, Janek Thomas, and Michel Lang. mlrMBO: A modular framework for model-based optimization of expensive black-box functions. *arXiv preprint arXiv:1703.03373*, 2017.
- [9] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.
- [10] W. Felber, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and Linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 171–172, March 2015.
- [11] Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. Parallel algorithm configuration. *Learning and Intelligent Optimization*, pages 55–70, 2012.
- [12] Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. Singularity: Scientific containers for mobility of compute. *PLOS ONE*, 12(5):1–20, 05 2017.
- [13] John Ousterhout. Scripting: Higher-level programming for the 21st century. *IEEE Computer*, March 1998.
- [14] Jonathan Ozik, Nicholson Collier, Justin M. Wozniak, and Carmine Spagnuolo. From desktop to large-scale model exploration with Swift/T. In *Proc. Winter Simulation Conference*, 2016.
- [15] O. Ronneberger, P. Fischer, and T. Brox. U-Net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, volume 9351 of *LNCS*, pages 234–241. Springer, 2015. (available on arXiv:1505.04597 [cs.CV]).
- [16] Justin M. Wozniak, Timothy G. Armstrong, Michael Wilde, Daniel S. Katz, Ewing Lusk, and Ian T. Foster. Swift/T: Scalable data flow programming for distributed-memory task-parallel applications. In *Proc. CCGrid*, 2013.
- [17] Justin M. Wozniak, Rajeev Jain, Prasanna Balaprakash, Jonathan Ozik, Nicholson Collier, John Bauer, Fangfang Xia, Thomas Brettin, Rick Stevens, Jamaludin Mohd-Yusof, Cristina Garcia Cardona, Brian Van Essen, and Matthew Baughman. CANDLE/Supervisor: A workflow framework for machine learning applied to cancer research. In *Proc. Computational Approaches for Cancer @ SC*, 2017.

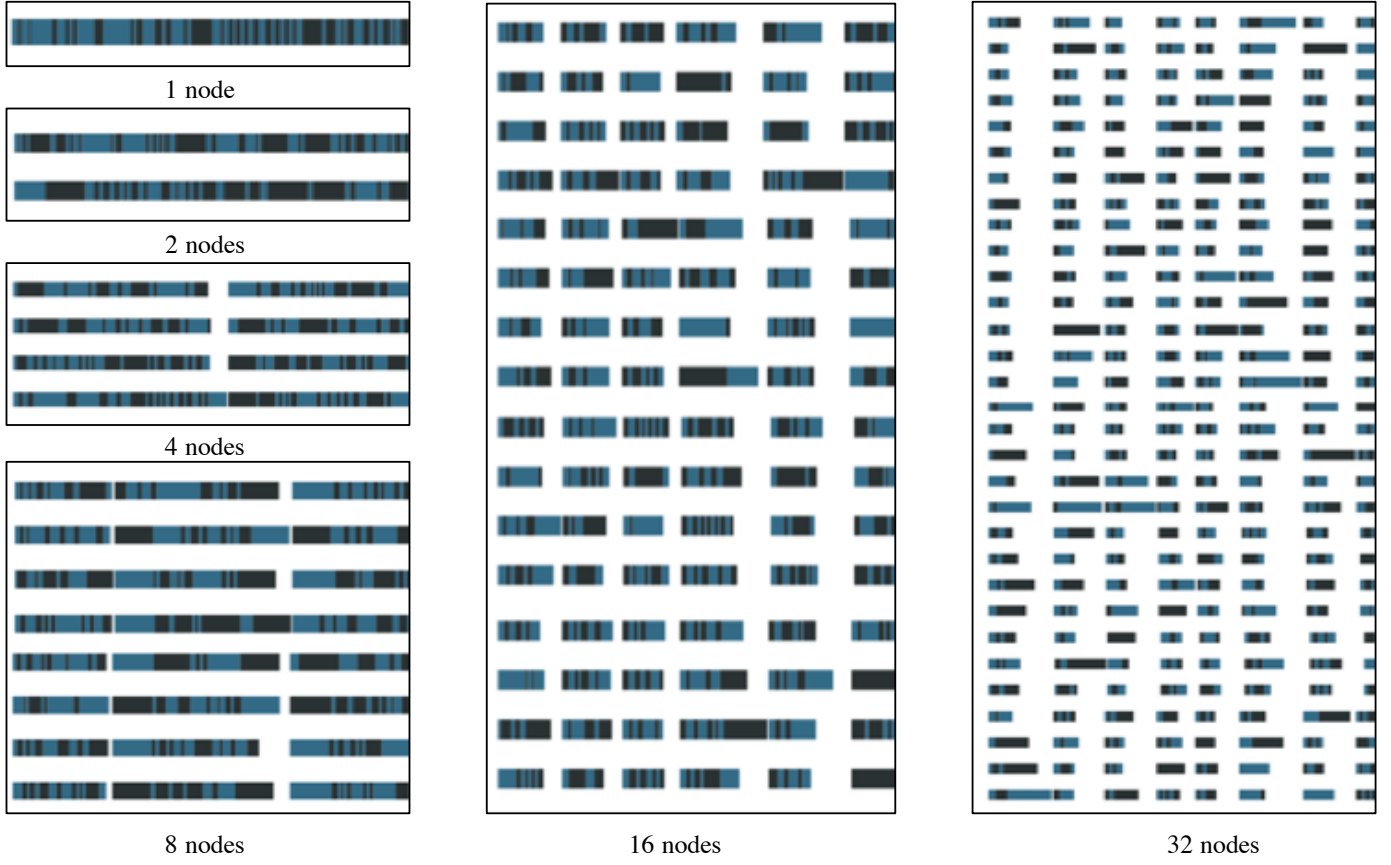


Fig. 7: Scaling mlrMBO optimization from 1 to 32 nodes, 1 GPU per node.