Scaling Deep Learning for Cancer with Advanced Workflow Storage Integration

Justin M. Wozniak[1], Philip E. Davis[2], Tong Shu[3], Jonathan Ozik[4], Nicholson Collier[4], Manish Parashar[2], Ian Foster[1], Thomas Brettin[5], and Rick Stevens[6]
[1] Data Science & Learning, Argonne National Laboratory and University of Chicago
[2] Rutgers University
[3] Mathematics & Computer Science, Argonne National Laboratory
[4] Decision and Infrastructure Sciences, Argonne National Laboratory and University of Chicago
[5] Computing, Environment, and Life Sciences, Argonne National Laboratory
[6] Computing, Environment, and Life Sciences, Argonne National Laboratory and University of Chicago

Abstract—**Cancer Deep Learning Environment (CANDLE) benchmarks and workflows will combine the power of exascale computing with neural network-based machine learning to address a range of loosely connected problems in cancer research. This application area poses unique challenges to the exascale computing environment. Here, we identify one challenge in CANDLE workflows, namely, saving neural network model representations to persistent storage. In this paper, we provide background on this problem, describe our solution, the Model Cache, and present performance results from running the system on a test cluster, ANL/LCRC Blues, and the petascale supercomputer NERSC Cori. We also sketch next steps for this promising workflow storage solution.**

I. Introduction

Data management is a critical part of emerging deep learning architectures [19]. Training neural networks demands high-end CPUs, GPUs, and other accelerators, but these workloads start with loading large amount of training data and end with the storage of large representations of the weights in the neural networks. Training data sets can be very large, e.g. thousands of hours of driving video or petabytes of genomics data, making typical training cycles read-intensive. Some workloads, however, such as hyperparameter optimization or population-based training generate, store, and reload many network models over time. This is in addition to checkpointing for fault recovery. In this work, we consider the problem of storing neural network models (NNs).

The CANcer Deep Learning Environment (CANDLE) application suite enables top-tier supercomputing systems be applied to three key problem areas in cancer research, the RAS pathway problem, the drug response problem, and the treatment strategy problem. The initial release of the application framework CANDLE/Supervisor [41] addresses the problem of hyperparameter exploration for NNs. Initial results involving applications from the CANDLE project running on DOE systems at OLCF Titan, ALCF Theta, and NERSC Cori demonstrate both scaling and multi-platform execution.

The ultimate goal of CANDLE is to develop high-quality NNs to address their target cancer problems. Its hyperparameter optimization workflow then generates and evaluates many single-node NN configurations on many nodes, with each node evaluating one configuration after another. The total data generated from evaluating C configurations is thus $C \times M$, where M is the memory of a single node in bytes; if a computer has N nodes, each able to evaluate a configuration in T seconds, then data is generated at a rate of $M \times N \ / \ T$ bytes/second. For example consider the growth in potential write rate over the last generation of DOE computing installations:

- On Cori [1], we have M=16 GB, N=9,000, and T=3,600 for a sustained rate of 40 GB/s.
- On Summit [2], we have M=512 GB, N=4,608, and T=3,600 for a sustained rate of 655 GB/s.

As computers increate in speed and scale, these rates will increase rapidly.

Thus, there is a great deal of data, more than can be expected to be written to persistent storage. Since low-quality NNs do not have to be saved, we can perform online analysis of the model ensemble and reduce the storage access by simply discarding the low-quality NNs. Thus, a high-speed cache is needed.. Our specific solution to this problem is the Model Cache, a system that temporarily stores the NNs until analysis and reduction are performed. This solution integrates the Swift/T workflow system and the DataSpaces cache storage system.

This paper offers the following: **(1)** a description of several machine learning-based workflows relevant to cancer; **(2)** an architecture for caching intermediate neural network data products; and **(3)** performance results from running the system on large-scale systems.

The remainder of this paper is organized as follows. In §II, we describe the aspects of machine learning relevant to this work. In §III, we describe the three CANDLE applications currently being developed. In §IV, we describe the background of the components of the new Model Cache. In §V, we describe the architecture of the CANDLE/Supervisor software system and its integration with the Model Cache. In §VI, we describe the practicalities and portability issues. In §VII, we describe performance results from these systems. In §IX, we describe future work, and we conclude in §X.

II. Use Cases

Machine learning (ML) has the capability to transform many scientific problems. In response to the growing power of ML techniques and the increasing available computing power at large scale computing facilities, the U.S. Department of Energy Exascale Computing Project (ECP) launched the **Cancer Deep Learning Environment (CANDLE)**. CANDLE is developing a suite of software to support scalable ML, and in particular deep learning (DL), that is, ML with multi-layer neural networks, on DOE supercomputing resources. While CANDLE is focused on three cancer pilot projects in the near term, its longer-term goal is to support a wide variety of DL applications across DOE science domains.

A. Deep learning frameworks

Numerous research groups in both industry (Google, Facebook, Microsoft, etc.) and academia (Berkeley, Oxford, Toronto, etc.) are developing DL frameworks. Popular frameworks include Caffe [22], Keras [11], Theano [37], Torch [14], Poseidon [42], Neon [35], TensorFlow [3], CNTK [28], and the Livermore Big Artificial Neural Net (LBANN) [38]. Each of these frameworks differ with respect to the ML tasks they target, their ease of use, data pre-processing, and target problems. Most frameworks were architected for a single node implementation and a few distributed memory multi-node implementations have recently emerged; but these implementations are primarily targeted at smaller core counts and for commodity cluster environments. Moreover, these implementations rely on avoiding communication by storing data on local disks. Implementations targeting high-performance computing systems will need novel techniques to fully exploit these systems' specialized interconnect bandwidth and topologies, and deep memory hierarchies.

B. Deep learning background

The artificial neural networks used in CANDLE comprise a set of layers, each with one or more nodes, and connected via links that send data from one node to another. Each node in each layer applies some function to its input values to determine the data that it should forward to the node(s) to which it is connected in the next layer. Using notation defined by Nielsen [24] and illustrated in Figure 1, the jth node in layer l has bias b^l_j and output (aka activation) a^l_j, and defines a weight w^l_{jk} for the connection from the kth node in layer $l-1$. Then:

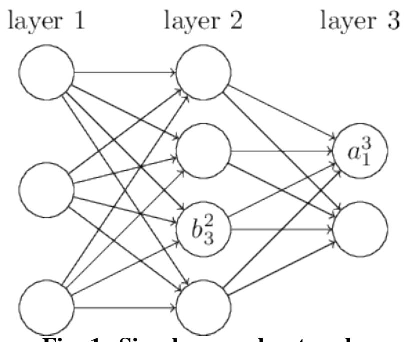


Fig. 1: Simple neural network.

$$a^l_j = K\left(\sum_k w^l_{jk} a^{l-1}_k + b^l_j\right)$$

where the activation function K is some predefined function, such as the hyperbolic tangent or sigmoid function, that allows for a smooth but nonlinear mapping from input to output.

We provide this background information to explain the nature of the computational challenges faced in CANDLE. As shown in Figure 2, we can think of the process by which a DL model is developed and applied as a pipeline involving distinct model selection, model training, and inferencing steps.

Model selection, also known as hyperparameter optimization, is concerned with selecting the NN architecture that is to be used for an application: that is, the DL method to be used and the set of layers, nodes, and links on which that method will be applied. This process is largely ad hoc and artisanal, combining expert knowledge and judgment (e.g., to select a DL method and place constraints on the NN architecture) with large-scale computation to explore architectural variants—the latter phase being the hyperparameter optimization process.

Model selection requires a mechanism for evaluating different DNN architectures. Typically this involves first a *model training* run, in which the DNN is trained on a given training data set and validated repeatedly against a test data set. The training run is performed over a series of *epochs*, in which the weight of each node in a particular DNN architecture is adjusted to minimize the error in the output values for the complete set of training data, and then an evaluation step where a second hold-out set of data are used to evaluate the trained model, yielding a score.

A common approach to training is to use an algorithm called gradient descent to find the set of weights over all nodes that collectively minimize the error in the validation. This training process is complex, with many algorithms and algorithm variants proposed that variously improve the quality of the minimum value (e.g., by avoiding local minima in the optimization) and/or reduce computational requirements. The training process can be correspondingly computationally expensive.

Finally, **inferencing**, also known as scoring, is concerned with applying a trained DNN to a particular input (e.g., new cancer patient data) to obtain a new output (e.g., proposed treatment solution). The data and computation involved in this phase are typically much smaller than that involved in the selection or training phases. This provides the value ($F(p)$) that is optimized by the overall workflow.

In the work reported here, we are concerned primarily with the hyperparameter optimization problem to apply automated optimization techniques to enhance the role of human expertise when designing NNs.

C. Hyperparameter optimization

The hyperparameter optimization problem is to find, among the space of all possible DNN architectures, the combination of DL method and associated layers, nodes, and links that yields the best, or at least satisfactory, performance on some supplied test data. The number of possible architectures can be extremely large: easily $O(10^{21})$ in the current CANDLE workflows, as described below.

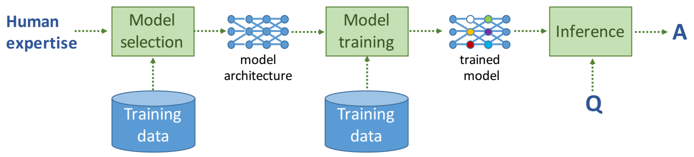


Fig. 2: A schematic view of a typical DL model development and application process. Feedback loops may apply, as when new data or poor inferencing results leads to repeated model training or even repeated model selection.

The architecture of an NN is heavily parameterized. The design parameters broadly include the number of layers, neurons per layer, activation function, and so on. The quality of the network is essentially its accuracy; a loss function F is determined such that its value is a measure of the error in the trained network behavior when applied to a validation set. The hyperparameter optimization problem is to minimize $F(p)$ across all parameter sets p in the parameter space P. However, P is large and F is expensive. P is the cross product of all valid network settings, some of which may be categorical, some integer, and some continuous. Evaluating F involves training the network on a training data set and applying it to a validation set, a task that can take minutes, hours, or longer. In the end we must collect, for each parameter set (p), the associated NN model representation (i.e., the result of the training process) and the resulting F, as well as other data for profiling or validation.

With such large state spaces, it is rarely feasible to perform an exhaustive search. Grid search methods, which define a few possible values for each parameter and then evaluate every feasible combination of those parameters, rarely perform well as they cover too little of the space; random search methods often perform better but are still not adequate. Simple gradient descent or multiple gradient descent can reduce the number of configurations in the search space that must be evaluated by one or two orders of magnitude, but that still leaves an intractable problem,

Various frameworks and libraries implement more sophisticated methods for exploring larger hyperparameter spaces. For example, HyperTune [27] uses Bayesian optimization to refine network hyperparameters. Another alternative is the popular Python library, SciKit-Learn [26], a multipurpose ML library for Python (easily integrated with Keras) that can be used for hyperparameter search. HyperOpt [7], a hyperparameter search framework that is designed to perform searches using distributed hardware, has a SciKit-Learn variant [6]. Genetic/evolutionary algorithms may also be applied. An implementation of genetic algorithms for hyperparameter search is the NeuroEvolution of Augmenting Topologies (NEAT) algorithm [34]. Another system for evolutionary algorithms for hyperparameter tuning is Optunity [12], which works with the Distributed Evolutionary Algorithms in Python (DEAP) [18] framework that implements different, generalized evolutionary algorithms.

The system that we used in this work is mlrMBO [9], a R-based black-box optimization package that uses a surrogate regression model to guide the function evaluations required to approximate a given objective function. It supports mixed continuous, categorical, and conditional parameters. mlrMBO follows a Bayesian optimization [8] approach, as follows. In the initialization phase, n_s configurations are sampled at random, evaluated, and a surrogate model M is fitted to the input-output pairs. In the iterative phase, the model M is used to identify n_b promising input configurations at each iterations, which are then evaluated with the function (in our case, by performing model training and evaluation). These configurations are selected using an infill criterion that seeks to trade off exploitation and exploration by choosing configurations that either have a good expected objective value (exploitation) or high potential to improve the quality of 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 [10] because it can handle such parameters directly, without the need to encode the categorical parameters as numeric. For the infill criterion, we used qLCB [20], which proposes multiple points with varying degrees of exploration and exploitation.

We use the Extreme-scale Model Exploration With Swift (EMEWS) [25] framework to drive many concurrent model evaluations. This framework uses the Argonne-developed Swift/T [40], [4] language to distribute the model exploration workload efficiently across a multi-node system. We have previously used it to explore spaces of size $O(10^{91})$ [13].

III. Workflows

We next describe the three pilot cancer problems that motivate the design of our systems solution. While each challenge is at a different scale (i.e., molecular, cellular, population) and has a specific scientific team collaborating on the data acquisition, data analysis, model formulation, and simulation runs, they also share several common threads. Common sets of cancer types appear at all three scales, all have to address significant data management and data analysis problems, and all need to integrate simulation, data analysis, and DL. CANDLE is investigating three promising pilot applications of DL technology to cancer research [41]:

P:RAS – The RAS pathway problem. The RAS/RAF pathway is a series of chemical events that is implicated in 30% of cancers. The goal of this pilot is to understand the molecular basis of key protein interactions in this pathway. We expect this capability to accelerate the identification and development of effective therapeutics targeting cancers driven by RAS mutations, including the three deadliest cancers occurring today: pancreatic, lung, and colon. This approach starts with a set of outputs from large-scale molecular dynamics simulations, to which unsupervised learning is applied to uncover features that can be used to describe the state space of protein movement and binding, producing a higher-level surrogate model. This higher-level model can then be used to explore (far more efficiently) the possible dynamics of RAS interactions, delivering many millions of hypothetical trajectories that are then scored according to likelihood. By investigating (through direct numerical simulation) the most likely of these trajectories, we close the loop—testing our hypothesis and then learning from the results. Any new information is used to refine the definitions of likelihood and affect future hypothesis. We expect this use of ML plus molecular dynamics to develop and test protein binding hypotheses to enhance our understanding of RAS signaling pathways dramatically.

P:DRUG – The drug response problem. The goal of this pilot is to develop predictive models for drug response that can be used to optimize pre-clinical drug screening and drive precision medicine-based treatments for cancer patients. integrate information about drug molecular structures, drug interactions, drug combinations and drug molecular targets with information about the patient's genetics, including their baseline genotype as well as the specific genetics and other molecular and cellular properties of their tumor, including gene mutations, gene expression patterns, proteome, transcriptome including small and non-coding RNAs, metabolomics, prior treatments, co-morbidities and environmental exposure. Our current working data contains drug and drug-like molecular screening data from over 300,000 compounds that have been tested on at least 60 cell lines, giving us $O(10^7)$ training cases.

P:TREAT – The treatment strategy problem. The goal of this pilot is to automate the analysis and extraction of information from millions of cancer patient records to determine optimal cancer treatment strategies across a range of patient lifestyles, environmental exposures, cancer types and healthcare systems.

Traditional natural language processing (NLP) algorithms have been developed to automate this process, using carefully crafted keyword-based rules for information extraction. However, the tremendous variation in clinical expression and the large size of the controlled medical vocabularies (containing more than 100,000 medical terms and expressions describing diseases, conditions, symptoms, and medical semantics that are typically present in unstructured clinical text) mean that hand-engineered rule extraction is neither scalable nor effective for large-scale clinical deployment. DL has the potential to address these challenges and capture both semantic and syntactic information in clinical text without requiring explicit knowledge of clinical language.

IV. Frameworks

In this section, we describe the two systems we brought together to construct the Model Cache.

A. *Swift/T*

Swift/T [40] is the workflow system used by CANDLE to control hyperparameter search and other ensembles such as uncertainty quantification. This system revolutionized high-performance workflows by supporting fully in-memory workflows that combine in-memory libraries and data into a composite application capable of representing a complete computational experiment in a lightweight script. A key part of this effort involved scaling the workflow enactment system into a fully parallel runtime [39] as well as developing compiler optimizations for distributed dataflow processing [4]. The system can thus maintain trillions of tasks, executing billions per second across petascale machines.

The Swift/T architecture is based on a handful of premises:

1) The workflow system should run inside one big allocation on the machine, and not rely on the login nodes or other site-specific features.
2) The workflow system should be developed by programming against standard APIs such as MPI, and not rely on non-standard systems features.
3) The workflow system should be able to invoke user code through library interfaces or scripting language interfaces (e.g., Python, Tcl) without launching external processes.
4) Workflow progress should be possible without accessing the filesystem (operating on in situ data).

These design premises made Swift/T a natural fit for deep learning workflows on HPC systems.

B. *DataSpaces*

DataSpaces [16] is a scalable data-staging substrate that supports advanced coordination and interaction services. DataSpaces provides the abstractions and mechanisms to support flexible and dynamic inter-application coupling and interactions at runtime, and supports asynchronous data insertion and retrieval to/from the staging area. It provides simple put/get semantics for applications to exchange data through a shared space abstraction. The DataSpaces runtime enables direct memory-to-memory communication between the interacting nodes using Remote Direct Memory Access (RDMA).

The goal of the DataSpaces abstraction is to enable the live data of interest, which is extracted from a running application, to be efficiently indexed, and asynchronously accessed and processed by other components in the application workflow. The data of interest can be dynamically specified as tuples. Tuples are essentially key-value pairs, and the keys are defined using an application-specific address space.

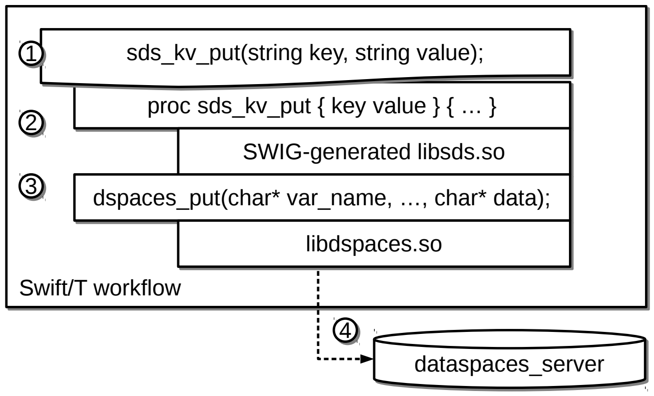
Applications use DataSpaces to avoid the latency of parallel file-systems caused by concurrent accesses by multiple independent tasks and also to accelerate data transfers, with less variability compared to using persistent storage systems such as files or databases.

C. *Integration: The SDS Module*

The Swift-DataSpaces (SDS) module is a generic Swift/T extension library that allows workflows to share data using the DataSpaces tuple space, in accordance with the Swift/T design premises (§IV-A). The module is based on the following typical plan for Swift/T extensions:

1) Swift/T can call directly into C functions, such as the DataSpaces API, if they have a Python or Tcl wrapper.
2) We can easily use SWIG to generate Tcl wrappers for C libraries.
3) We then can write Tcl programs that use DataSpaces.
4) We can produce concise Swift/T functions that use the Tcl functions.
5) Then we can write highly concurrent Swift/T workflows that directly access DataSpaces (without forking external tools); that is, DataSpaces is part of the workflow system.

This SWIG-based approach is a popular way to access compiled code from higher-level languages such Python, and is used by many scientific projects such as PETSc, Trilinos, and NAMD. Swift/T can directly access Tcl functions, so we can create a Tcl interface and produce Swift/T wrappers for these functions with minimal coding effort.



1
sds_kv_put(string key, string value);
2
proc sds_kv_put { key value } { … }
SWIG-generated libsds.so
3
dspaces_put(char* var_name, …, char* data);
libdspaces.so
Swift/T workflow
4
dataspaces_server

Fig. 3: Callstack used when Swift/T calls DataSpaces functions.

As shown in Figure 3, the Swift/T programmer simply programs against a simple string/string key/value API ①. Behind the scenes, this is translated into a Tcl function ②, which invokes the C-based DataSpaces API ③. The data is then moved over the network to the DataSpaces server ④.

Our prototype API simply provides features to move strings and whole (binary) files into and out of DataSpaces. This is a simplified API for users like CANDLE that only need these features at this point. In the near future, we plan to add support for the richness of the DataSpaces API, including multidimensional arrays and variably-typed, slice-based data access.

V. Architecture

Emerging multi-petaflop supercomputers are powerful platforms for ensembles of neural networks that can address many problems in cancer research, but it is difficult to assemble and manage large studies on these machine, which have tens of thousands of compute nodes. Typical workflow approaches would face challenges due to system scale, system complexity, management of complex workflow patterns, integration with disparate software packages, and data acquisition. CANDLE/-Supervisor addresses the problem of hyperparameter optimization for cancer-based problems, and solves the common workflow challenges outlined above.

A. Hyperparameter search framework

To support the search patterns described in §III, we developed the CANDLE/Supervisor architecture diagrammed in Figure 4. 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 [25] enables the user to plug in ME algorithms into a workflow for arbitrary model exploration; optimization is a key use case. The ME algorithm can be expressed in Python or R. 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). 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 [40], [4] workflow system is used to manage the overall workflow. It integrates with the various HPC schedulers (§VI) 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. In this use case, however, 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 **DL** has access to all the resources on the node. The DL is the underlying learning engine; we have tested with Theano and TensorFlow. 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**.

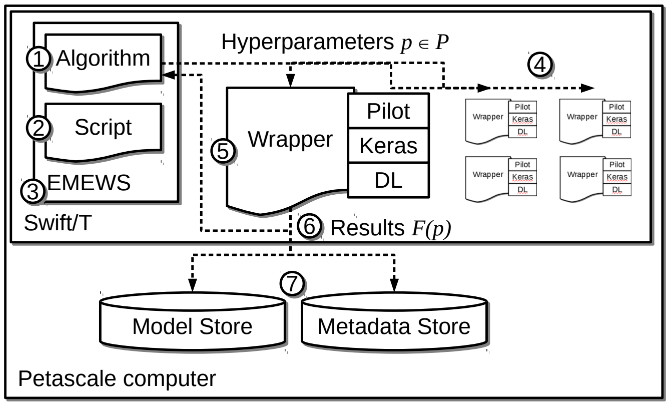


Fig. 4: CANDLE/Supervisor original architecture.

B. Key contribution: The Model Cache

Conceptually, CANDLE Model Store contains all trained models produced by the system, which in practice means that it is the size of system memory, multiplied by the number of iterations carried out, which in our current applications are performed at the rate of roughly one per hour. So the Model Store poses an I/O challenge similar to that of regularly checkpointing a traditional exascale application. However, we observe that most models are not worth saving, and this can be determined at run time by comparing the relative quality of the models in the Model Cache and deciding which to actually write to persistent storage and which to simply delete. Thus, we just need to save the models in a cache long enough for the workflow system to compare them to each other.

To support this mode of operation, we designed the Model Cache, shown in Figure 5. This extends the CANDLE Supervisor architecture with multiple components, including local storage on each node and the DataSpaces servers. Using DataSpaces storage allows for low-latency acces to model data, since it is independent of the parallel I/O subsystem. In a typical use case, the workflow stores model configurations and associated in the Model Cache, where a separate process can, depending on the application, compute summary statistics, eliminate uninteresting configurations, and/or otherwise compress configurations before outputting them.

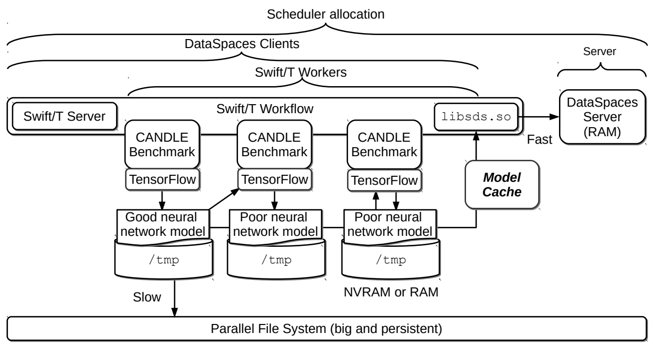


Fig. 5: CANDLE/Supervisor with Model Cache.

VI. Computing systems

We ran our benchmark workflows described previously (§III) on LCRC Blues and NERSC Cori. These systems vary greatly in their hardware and software systems, as indicated by the following summary (which does not include differences in compiler versions, software module management, and storage system policies or capabilities):

- **LCRC *Blues* at Argonne National Laboratory**
  - 306 nodes with
    - 16-core Intel Xeon E5-2670 @ 2.6GHz
    - 16 GB RAM
  - Scheduler: PBS
- **NERSC *Cori* at Lawrence Berkeley National Laboratory**
  - 2,388 nodes with
    - Intel Xeon Haswell CPUs
    - 128 GB RAM
  - 9,688 nodes with
    - Intel Xeon Phi
    - 16 GB MCDRAM, 96 GB DDR
  - Scheduler: SLURM

We use Swift/T to abstract the scheduler and compute layout settings. The launch parameters for Swift/T allow the user to specify the scheduler type, processor count, workers per node, and other common settings in a uniform way across systems.

We use our Wrapper script abstraction (§V) to abstract the Python configuration and DL library settings. The wrapper script is invoked in one of two ways, either by a short piece of Python code, the text of which is embedded in the Swift/T script and executed directly by the Swift/T runtime embedded Python interpreter, or by a bash script that is executed via a Swift/T `app` function [40]. App functions are Swift/T language functions that are implemented as command-line programs, in this case a shell script that calls the Python interpreter passing it the wrapper script as an argument. In each case, the Swift/T script receives the hyperparameters from the model exploration algorithm and passes them to the wrapper script either via a string template in the embedded Python code or as a command line argument to the bash script.

VII. Performance results

In this section, we measure the performance of the Swift-DataSpaces integration for synthetic workloads. In each case, we measure the performance of a Swift/T workflow using DataSpaces for data sharing (termed 'Bench'), against a control case of using the parallel file system (PFS) for the same workload (termed 'Control').

A. Small put/get rate

In the first test, we measure the performance of the two systems on small key/value pairs (8 bytes). The two workflow scripts are shown here:

```
foreach i in [0:N-1] {
  sds_kv_put("key"+i, "value1") =>
  sds_kv_get("key"+i, 100);
}
```

Bench 1

```
foreach i in [0:N-1] {
  file f<"key"+i> = write("value");
  string s = read(f);
}
```

Control 1

The Bench case uses our DataSpaces module (§IV-C) to do N paired put()/get() operations. Each put()/get() pair is ordered by the Swift/T ordering operator => but the N loop iterations execute concurrently anywhere in the system, ordered by the Swift/T load balancer.

The Control case uses the Swift/T builtin functions write()/read() to write the string "value" into file key*i*. The write()/read() pair is ordered by the data dependency on f.

We set N to 10 times the number of workers in the system.

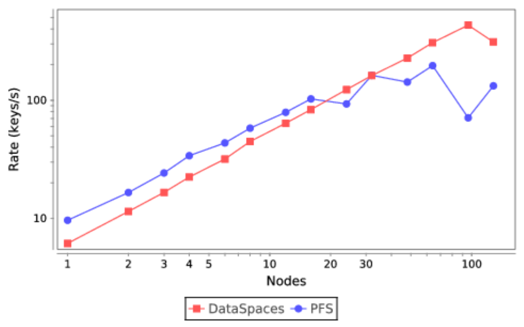
Results are shown in Figure 6. We simply timed the whole workflow and obtained a keys/second rate for each node count. Note that two additional nodes are required for each run: a Swift/T server and a DataSpaces server.

B. Large put/get rate

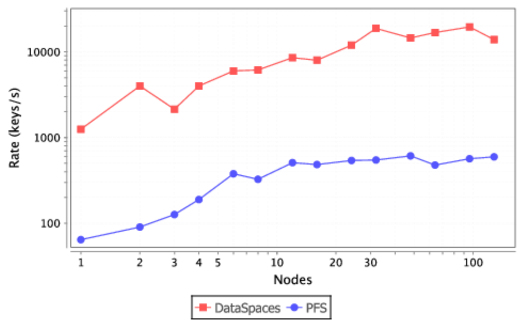
In this case, we measure the ability of the system to manage larger values of size 1 MB. An input file of this size was created in advance, then read by the workflow system and distributed as the value string, and used in workflows like Bench 1 / Control 1 above. Results are shown in Figure 7.

C. File transfer rate

In this case, a workflow more like the CANDLE use cases is performed. A filename is generated N times (M=10). For Bench 3, the file is stored in the local RAM-based /tmp, but in the Control 3 it is written to the shared file system. A make_data() task is then executed that writes 1 MB to that file. For Bench 3, the file is then copied to DataSpaces, for Control 3 it is simply left in the filesystem.



(a) Blues



(b) Cori

Fig. 6: Data access rate for simple put/get example with 1 KB messages.

D. Application to CANDLE

Bench 3 and Control 3 mimic the case shown in §V-B, in which files are produced in node-local storage, then copied to DataSpaces. This is the performance critical I/O step. Some small number of models may be reused and restarted elsewhere, or written to persistent storage. Most will be simply deleted. Results are shown in Figure 8.

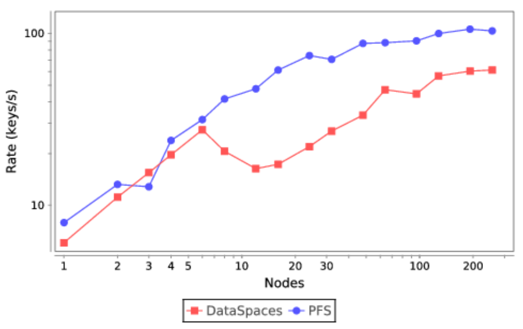
E. Interpretation

Our performance data is currently somewhat noisy due to the impact of other users in the system. We are continuing to run these tests to improve our understanding of the potential impact of this system.

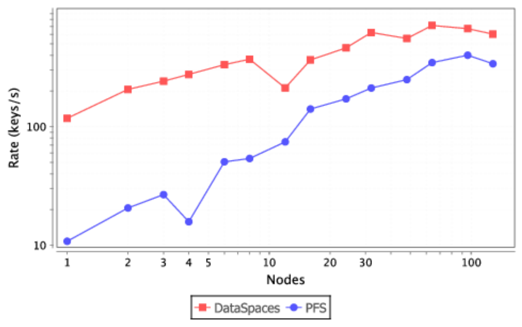
VIII. Related Work

A. Data Movement and I/O Systems

The capacity gap between computing components and data movement and storage systems has been increasing exponentially. In order to achieve a resilient service that moves data while hiding errors and latencies, Thain et al. developed a simple data movement system, Kangaroo, which makes opportunistic use of disks and networks to keep applications running [36]. Bent et al. designed the Batch-Aware Distributed File System (BAD-FS) to orchestrate large, I/O-intensive batch workloads on remote computing clusters distributed across the wide area [5]. BAD-FS consists of two novel components: a storage layer that exposes control of traditionally fixed policies such as caching, consistency, and replication, and a scheduler that exploits this control as necessary for different workloads. By extracting control from the storage layer and



(a) Blues



(b) Cori

Fig. 7: Data access rate for simple put/get example with 1 MB messages.

```
foreach j in [0:M-1] {
  foreach r in [0:turbine_workers()-1] {
    name = make_filename(j,r); //  in /tmp
    file f<name> = make_data(SIZE) =>
    sds_kvf_put(name, name);
  }
}
```

Bench 3

```
foreach j in [0:M-1] {
  foreach r in [0:turbine_workers()-1] {
    name = make_filename(j,r); //  in /PFS
    file f<name> = make_data(SIZE);
  }
}
```

Control 3

placing it within an external scheduler, BAD-FS manages both storage and computation in a coordinated way while gracefully dealing with cache consistency, fault-tolerance, and space management issues in a workload-specific manner. Zhang et al. developed a scalable many-task computing data management system that aggregates computing nodes' local storage for more efficient data movement strategies, and co-designed the data management system with the data-aware scheduler to enable dataflow pattern identification and automatic optimization [43]. In [17], Dorier et al. presented a methodology for the rapid development of custom data services, which are tailored to the needs of specific applications on specific hardware, and designed in close collaboration with users. This methodology promotes the design of reusable building blocks that can be composed together efficiently in runtime based on high-performance threading, tasking, and remote procedure calls.

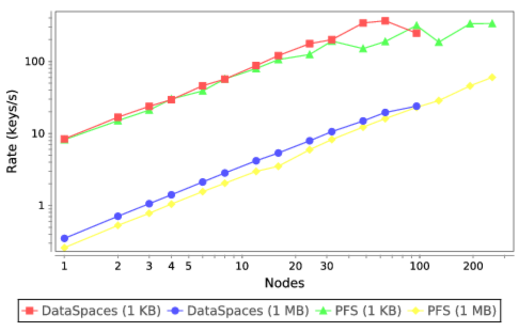


Fig. 8: File transfer rate (Blues).

B. *Workflow Engines*

There exist various workflow engines in different computing environments. The Pegasus Workflow Management System, first developed in 2001, maps abstract workflow descriptions onto distributed computing infrastructures, and separates the workflow description from the description of the execution environment, i.e. keeping the workflow description resource-independent. In its system design, the scalability is considered to meet user demands of complex and large-scale workflows (with millions of tasks). Pegasus workflows are based on the directed acyclic graph (DAC) representation of scientific computation composed of many tasks with dependencies among them [15].

C. *Data-intensive Workflows*

Next-generation e-science is producing colossal amounts of data, and these scientific applications typically feature data-intensive workflows comprised of moldable parallel computing jobs, such as MapReduce, with intricate inter-job dependencies. The granularity of task partitioning in each moldable job of such big data workflows has a significant impact on workflow completion time, energy consumption, and financial cost if executed in clouds. Shu et al. conducted an in-depth investigation into the properties of moldable jobs and provided an experiment-based validation of the performance model where the total workload of a moldable job increases along with the degree of parallelism [29]. In public cloud environments, which provide a cost-effective computing platform for big data workflows where moldable parallel computing models were widely applied to meet stringent performance requirements, Shu et al. constructed a big-data workflow mapping model based on the moldable parallel computing performance model, and formulated a workflow mapping problem to minimize workflow makespan under a budget constraint [32]. To solve this strongly NP-complete problem, they designed i) a fully polynomial-time approximation scheme for a special case with a pipeline-structured workflow executed on virtual machines of a single class, and ii) a heuristic for a generalized problem with an arbitrary DAC-structured workflow executed on virtual machines of multiple classes. Considering that large-scale workflows for big data analytics have become a main consumer of energy in data centers, Shu et al. also delved into the problem of static workflow mapping to minimize the dynamic energy consumption of a workflow request under a deadline constraint in Hadoop clusters, which is strongly NP-hard. In [30] and [33], a fully polynomial-time approximation scheme was designed for a special case with a pipeline-structured workflow on a homogeneous cluster and a heuristic was designed for the generalized problem with an arbitrary DAC-structured workflow on a heterogeneous cluster. This problem was further extended to a dynamic version with deadline-constrained MapReduce workflows to minimize dynamic energy consumption in Hadoop clusters. Shu et al. proposed a semi-dynamic online scheduling algorithm based on adaptive task partitioning to reduce dynamic energy consumption while meeting performance requirements from a global perspective, and also developed corresponding system modules for algorithm implementation in the Hadoop ecosystem [31].

In [23], Luttgau et al. presented an approach to augment the I/O efficiency of the individual tasks of workflows by combining workflow description frameworks with system I/O telemetry data, and introduced a conceptual architecture and a prototype implementation for HPC data center deployments.

IX. Future work

We have described an implementation of the basic features of a scalable workflow framework for machine learning applied to problems in cancer research. There remain many additional features yet to investigate and develop.

Jaderberg et al. [21] introduced population-based training (PBT) as a novel and performant hyperparameter optimization approach. The core idea of PBT is that a "population" of models are trained concurrently within a fixed computational budget and, unlike iterative approaches such as those employed with the mlrMBO library, there are no sequentially generated new models to train. Each of the training models can initiate an (asynchronous) update of its hyperparameters and model parameters (i.e., weights) if it observes that its predictive performance is significantly lagging behind the rest of the running models. By allowing a model to update its weights along with its hyperparameters, model training is "hot" restart, decreasing the time needed for training. In order to implement PBT, each running model's weights, along with hyperparameters and performance, need to be periodically and frequently stored in locations accessible by each of the other running models. Figure 9 shows the design we will utilize to store and retrieve this data. The hyperparameters and performance metrics can be stored in the Metadata Store (currently a Solr database). However, the much larger model weights will be stored using the Model Cache, with identifying keys stored in the Metadata Store. Each running model will thus be able to query the Metadata Store for performance metrics and retrieve hyperparameters and model weights through the combined data storage stack without significantly impacting system I/O.

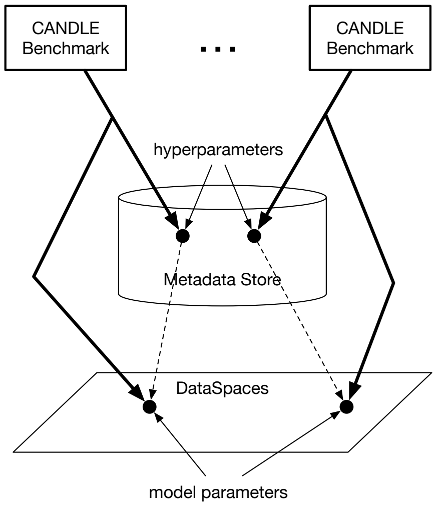


Fig. 9: CANDLE/Supervisor hyperparameter and model storage with the Metadata Store and DataSpaces.

X. Conclusion

Applying machine learning to cancer research is a promising approach in many aspects, including the benchmark problems used here, the RAS pathway, drug response, and treatment strategies. In this paper, we described one aspect of the data management problem for these workflows, efficiently managing neural network weights. Efficiently collecting the useful outputs from this process is an online data analysis and reduction problem suitable for advanced storage and caching techniques. We offered our solution by presenting the Model Cache, which tightly integrates Swift/T and DataSpaces. Thus it integrates with CANDLE/Supervisor, the framework for rapidly testing hyperparameter optimization techniques for machine learning models for CANDLE. The Model Cache saves NN outputs temporarily, while the rest of the workflow determines which models should be retained for the user and which should be discarded. The preliminary performance results show that our solution can outperform the existing filesystems, and is capable of even better performance at larger scales.

Cancer research is an important topic with significant societal impact. The Model Cache and associated infrastructure will allow research teams to leverage the most powerful high-performance computer systems in this problem space.

Acknowledgments

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation's exascale computing imperative. We gratefully acknowledge the computing resources provided on Blues, a high-performance computing cluster operated by the Laboratory Computing Resource Center at Argonne National Laboratory.

References

[1] Cori Configuration. http://www.nersc.gov/users/computational-systems/cori/configuration.
[2] Summit. https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit.

[3] Martín Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[4] Timothy G. Armstrong, Justin M. Wozniak, Michael Wilde, and Ian T. Foster. Compiler techniques for massively scalable implicit task parallelism. In *Proc. SC*, 2014.

[5] John Bent, Douglas Thain, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Miron Livny. Explicit control a batch-aware distributed file system. In *Proc. of NSDI*, page 27, San Francisco, CA, USA, Mar 2004.

[6] James Bergstra et al. Hyperopt: a Python library for model selection and hyperparameter optimization. *Computational Science & Discovery*, 8(1):014008, 2015.

[7] James Bergstra, Daniel Yamins, and David D. Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *Proc. of the 30th International Conference on Machine Learning*, 2013.

[8] 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.

[9] Bernd Bischl et al. mlrMBO: A modular framework for model-based optimization of expensive black-box functions. *arXiv preprint arXiv:1703.03373*, 2017.

[10] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[11] François Chollet et al. Keras. https://github.com/fchollet/keras, 2015.

[12] Marc Claesen et al. Easy hyperparameter search using Optunity. *CoRR*, abs/1412.1114, 2014.

[13] Chase Cockrell et al. High performance machine learning and evolutionary computing to develop personalized therapeutics. In *University of Chicago MindBytes Posters*, 2017.

[14] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A Matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011.

[15] Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip J. Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira da Silva, Miron Livny, and Kent Wenger. Pegasus, a workflow management system for science automation. *Future Generation Computer Systems*, 46:17–35, 2015.

[16] C. Docan, M. Parashar, and S. Klasky. DataSpaces: an interaction and coordination framework for coupled simulation workflows. *Cluster Computing*, 15, 2012.

[17] Matthieu Dorier, Philip Carns, Kevin Harms, Robert Latham, Robert Ross, Shane Snyder, and Justin Wozniak. Methodology for the rapid development of scalable hpc data services. In *Proc. of Workshop in conjunction with ACM/IEEE Supercomputing Conference*, Dallas, TX, USA, Nov 2018.

[18] Félix-Antoine Fortin et al. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, July 2012.

[19] Mike Houston. Production deep learning and scale. In *Proc. Machine Learning in HPC Environments*, 2017.

[20] Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. Parallel algorithm configuration. *Learning and Intelligent Optimization*, pages 55–70, 2012.

[21] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu. Population based training of neural networks. *arXiv:1711.09846 [cs]*, November 2017. arXiv: 1711.09846.

[22] Yangqing Jia et al. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[23] Jakob Luttgau, Shane Snyder, Philip Carns, Justin M. Wozniak, Julian Kunkel, and Thomas Ludwig. Toward understanding i/o behavior in hpc workflows. In *Proc. of Workshop in conjunction with ACM/IEEE Supercomputing Conference*, Dallas, TX, USA, Nov 2018.

[24] Michael A Nielsen. *Neural networks and deep learning*. Determination Press, 2015. http://neuralnetworksanddeeplearning.com.

[25] 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.

[26] 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.

[27] Greg Kochanski Puneith Kaul, Daniel Golovin. Hyperparameter tuning in cloud machine learning engine using Bayesian optimization, 2017. https://cloud.google.com/blog/big-data/2017/08/hyperparameter-tuning-in-cloud-machine-learning-engine-using-bayesian-optimization.

[28] Frank Seide and Amit Agarwal. CNTK: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 2135–2135, New York, NY, USA, 2016. ACM.

[29] Tong Shu. *Performance Optimization and Energy Efficiency of Big-data Computing Workflows*. Dissertation, New Jersey Institute of Technology, Newark, NJ, USA, Aug 2017. http://archives.njit.edu/vol01/etd/2010s/2017/njit-etd2017-096/njit-etd2017-096.pdf.

[30] Tong Shu and Chase Q. Wu. Energy-efficient mapping of big data workflows under deadline constraints. In *Proc. of Workshop on Workflows in Support of Large-Scale Science in conjunction with ACM/IEEE Supercomputing Conference*, pages 34–43, Salt Lake City, UT, USA, Nov 2016. http://ceur-ws.org/Vol-1800/paper5.pdf.

[31] Tong Shu and Chase Q. Wu. Energy-efficient dynamic scheduling of deadline-constrained MapReduce workflows. In *Proc. of IEEE eScience*, pages 393–402, Auckland, New Zealand, Oct 2017.

[32] Tong Shu and Chase Q. Wu. Performance optimization of Hadoop workflows in public clouds through adaptive task partitioning. In *Proc. of IEEE INFOCOM*, pages 2349–2357, Atlanta, GA, USA, May 2017.

[33] Tong Shu and Chase Q. Wu. Energy-efficient mapping of large-scale workflows under deadline constraints in big data computing systems. *Elsevier FGCS*, in press. https://www.sciencedirect.com/science/article/pii/S0167739X17300468.

[34] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

[35] Nervana Systems. Neon. https://github.com/NervanaSystems/neon, 2017. Accessed: 2017-09-14.

[36] Douglas Thain, Jim Basney, Se-Chang Son, and Miron Livny. The kangaroo approach to data movement on the grid. In *Proc. of HPDC*, pages 325–333, San Francisco, CA, USA, Aug 2001.

[37] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[38] Brian Van Essen et al. LBANN: Livermore Big Artificial Neural Network HPC Toolkit. In *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*, MLHPC '15, pages 5:1–5:6, New York, NY, USA, 2015. ACM.

[39] Justin M. Wozniak, Timothy G. Armstrong, Ketan Maheshwari, Ewing L. Lusk, Daniel S. Katz, Michael Wilde, and Ian T. Foster. Turbine: A distributed-memory dataflow engine for high performance many-task applications. *Fundamenta Informaticae*, 28, 2013.

[40] 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.

[41] 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.

[42] Hao Zhang et al. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. *CoRR*, abs/1706.03292, 2017.

[43] Zhao Zhang, Daniel S. Katz, Justin M. Wozniak, Allan Espinosa, and Ian Foster. Design and analysis of data management in scalable parallel scripting. In *Proc. of ACM/IEEE Supercomputing Conference*, Salt Lake City, UT, USA, Nov 2012.