Strategies for Integrating Deep Learning Surrogate Models with HPC Simulation Applications

Junqi Yin
Oak Ridge National Laboratory
Oak Ridge, USA
yinj@ornl.gov

Feiyi Wang
Oak Ridge National Laboratory
Oak Ridge, USA
fwang2@ornl.gov

Mallikarjun (Arjun) Shankar
Oak Ridge National Laboratory
Oak Ridge, USA
shankarm@ornl.gov

***Abstract*—The emerging trend of the convergence of high performance computing (HPC), machine learning/deep learning (ML/DL), and big data analytics presents a host of challenges for large-scale computing campaigns that seek best practices to interleave traditional scientific simulation-based workloads with ML/DL models. A portfolio of systematic approaches to incorporate deep learning into modeling and simulation serves a vital need when we support AI for science at a computing facility. In this paper, we evaluate several strategies for deploying deep learning surrogate models in a representative physics application on supercomputers at the Oak Ridge Leadership Computing Facility (OLCF). We discuss a set of recommended deployment architectures and implementation approaches. We analyze and evaluate these alternatives and show their performance and scalability up to 1000 GPUs on two mainstream platforms equipped with different deep learning hardware and software stacks.**

***Index Terms*—AI surrogate model, HPC Simulation, SmartRedis**

I. Introduction

Traditionally, simulation and model-based applications dominate high performance computing (HPC) scientific campaigns. With the growth of machine learning (ML) especially deep learning (DL), data and learning is gradually being recognized as both a peer and a catalyst to accelerate scientific discovery process. There are several use patterns for combining simulations and learning depending on the extent of exchange of results and data between one modality (e.g., simulation) to another (e.g., training or inference). In one of the earliest examples of large-scale deep-learning campaigns, DL was used in a solely *stand-alone* mode of operation, learning from collected data to extract patterns to train classification models[1]. Alternatively, in a *complementary mode* of operation, deep learning may operate on large data produced by a simulation which are easy to label and thus support learning, or from observational data from instruments which inform parameters for simulation campaigns. Such ML/DL training can scale up to the full supercomputers [2, 3]. Finally, in an *interleaved* mode of operation for a within-a-campaign smart simulation, online learning (e.g., anomaly detection for early termination of a large simulation job [4]) or learning-simulation interleaved computing (e.g., steering through a large search space [5]) can accelerate the overall computational campaign. In this case, when we work with large simulations with bottleneck kernels, ML surrogate models can enable larger size and/or longer time scale. In this paper, we focus on integrating such surrogates within HPC simulation campaigns.

At Oak Ridge Leadership Computing Facility (OLCF), we encounter learning applications across the spectrum – from standalone distributed DL training [6] to interleaved AI/ML modeling and simulation [7]. We find that an attractive and natural approach to use resources at extreme scale is incorporating the interleaving approach by deploying AI/ML surrogate models into the simulation campaign. While there is already a Gordon Bell winning application [8] using surrogate modeling in a large-scale molecular dynamics simulation (which uses a DL framework built-in API (TensorFlow C++ API)), a systematic exploration on how best to deploy DL surrogates at scale is lacking. Through a real thermodynamics simulation application for materials science, we conduct an extensive evaluation of surrogate modeling approaches. Our contributions are the following:

- We systematically evaluate different deployment strategies for surrogate modeling in simulations on heterogeneous HPC platforms. We offer recommendations on the best approaches and practices based on the latency and throughput requirements of the applications.
- We demonstrate three types of implementation of integrating DL surrogate models with simulation applications, including framework built-in API, framework inference server, and third-party API.
- We develop a set of surrogate models for a real Monte Carlo application for thermodynamics calculation in materials science, and show its performance and scalability up to 1000 GPUs.
- We compare the performance of surrogate modeling on two platforms and DL stacks, one on Nvidia GPUs and CUDA, and the other on AMD GPUs and ROCm.

The rest of the paper is organized as follows: in Section II,

This manuscript has been co-authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

we briefly review the background on the example application and the developed surrogate models. We then provide details of our deployment architectures and implementation methods in Section III, and present the evaluation results in Section IV. We discuss the generalizability of our observations in Section V, and conclude in Section VI.

II. Background

A. Monte Carlo Simulation

Monte Carlo (MC) Simulation is an important numerical methods-driven computational strategy that has been used widely in quantum physics [9], nuclear physics [10], materials science [11], etc. It typically involves a repetitive sampling of the same targeted function (e.g., energy). Depending on the cost of function evaluation (e.g., *ab intio* or empirical), the state-of-the-art results often demand exponentially expensive computations (e.g., [9]). Therefore, there is a need and an opportunity to derive machine learning based surrogate models [12, 13] to replace the expensive function evaluations.

We focus our study on the MC-based thermodynamic calculation in materials science as the application to evaluate various deployment strategies for surrogate modeling [11]. We consider a real high-entropy alloy system, a MoNbTaW alloy, which has attracted a lot of attention [14] recently. Each type of atom has equal molar concentration on a body-centered cubic (BCC) lattice. The probability of an N atoms configuration $X = \{x_i\}_N$ at temperature T follows the Boltzmann's distribution $\exp(-E(X)/k_BT)$ where E is the total energy of the configuration and k_B is the Boltzmann constant.

To accelerate the sampling, replica of alloy systems at various Ts are simulated simultaneously via replica exchange MC simulations [15] with the following transition probability between replica m and n,

$$W(\{X_m, T_m\}|\{X_n, T_n\}) = \min\left[1, \exp(-\triangle)\right], \qquad (1)$$

where

$$\triangle = (1/k_BT_n - 1/k_BT_m)(E(X_m) - E(X_n)) \qquad (2)$$

Within each replica, an atom i is exchanged with its neighbor j according to classical Metropolis sampling with the acceptance probability $P_{ij} \propto \min\left[1, \exp(-(E(\{x_i, x_j\}) - E(\{x_j, x_i\}))/k_BT)\right]$.
Therefore, each MC step consists of an atom pair exchange trail for each atom with its neighbors and a replica exchange trail among different Ts. The computational complexity of MC sampling is hence $\sim O(N^2)$.

The cost of a MC step depends on how E is calculated. For fully first-principle based E, the computation campaign [16] is not affordable even with a current supercomputer. We have built machine learning regression models [13] using data from high-fidelity quantum-mechanical calculations. In this study, we train deep learning models of various sizes on the same dataset to show the considerations of deploying surrogate models.

TABLE I
Deep learning surrogate models for the energy evaluation of MoNbTaW alloy. The architecture is given by the number of nodes in each hidden layer (e.g. 128x2-64x2 means 2 hidden layers each with 128 nodes followed by 2 hidden layers with 64 nodes), and it is the same for each of the four element. The R^2 score is the averaged model performance of all elements.

name	architecture	# parameters	R^2 score	
			FP32	mixed
tiny	24-24	5,257	0.991	0.988
small	128x2-64x2 -24x2	55,817	0.991	0.992
mdedium	200x11	440,801	0.994	0.992
large	512x6-256x6 -128x6-64x6-32	2,019,009	0.994	0.993

B. Surrogate Modelling

Following the effective pair interaction formulation [13], the energy E can be expressed as the summation of the pair interaction of each atom with its neighboring atoms within some cut-off radius. Therefore, for an atom specie i, a model can be established from the neighboring atoms (input) to the subsystem energy E_i (target). The advantage of this formulation is that the same model can be applied to the alloy with different number of atoms.

In Table I, the details of our models and its performance are listed. All models follow the multilayer perceptron (MLP) network architecture. For the potential modeling, MLP is commonly used [8, 12] because the input features are usually well-defined physically quantities and the feature extraction operation such as convolution is not needed. In fact, the latest breakthrough in quantum potential modeling brought by DeepMind[17] is also built upon a MLP model. One thing to consider in choosing the size of hidden nodes is that a multiplier of eight is enforced to satisfy the requirement of DL specific acceleration engines (e.g., Tensor core in Nvidia GPUs).

Depending on the number of hidden layers and nodes, our model size ranges from 5K to 2M parameters. Comparing to exploding model sizes in learning, those models are on the small side, but they are of typical sizes[8, 12, 13, 17] in modeling energy potentials, where the input nodes are often limited given the interaction range. In fact, due to the near linear interaction of MoNbTaW alloy, the tiny model of 5K parameters seems to be sufficient, and the exploration of other models is mainly to show the effect of model size on the deployment. On the other hand, potential modeling usually requires a high threshold on the model accuracy because of the accumulative effect (i.e., summation of subsystems). Given our energy function is the sum of the surrogate model for each element, the small improvement in model accuracy has a magnified effect on overall simulation accuracy. Before model deployment, the first step is to optimize the computation defined in the model, which is discussed as follows.

Model Optimization 1) The models are usually generated from a python training code, and may contain training-only

operations (e.g., checkpoint saving) and variables. Hence the first step is to freeze and optimize (e.g., remove unused operations, fuse common operations) the computation graph in the model. For TensorFlow, this can be achieved with framework provided utilities such as `freeze_graph` and `optimize_for_inference`. 2) By default, the training is performed in single precision (FP32), so the inference is also in FP32. To take advantage of extra boost from the accelerator, the model should be trained in mixed precision (e.g., `mixed_precision` policy in TensorFlow). Some care is needed for regression task such as potential modeling. 3) The input data for the inference needs to be transferred to the accelerator, so the proper data type should be chosen to reduce the overhead of data transfer. In our case, because the inputs for the energy model are neighboring atom species, we use `numpy.uint8` in python for training and `tensorflow::DT_UINT8` and `uint8_t` in C++ for inference. Further data packing is possible.

III. Deployment Architectures and Methods

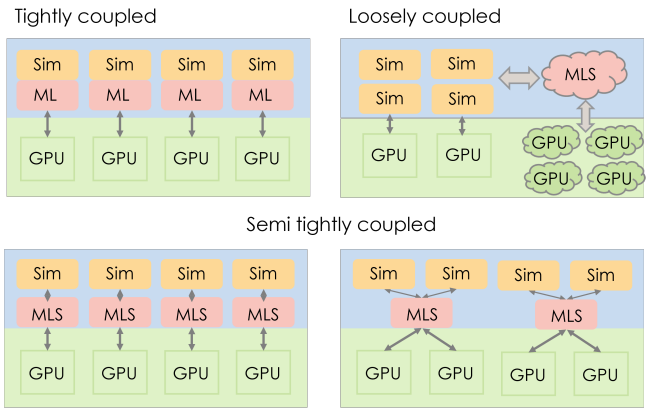


Fig. 1. Illustration of deployment architectures for integrating machine learning (ML) surrogate models with simulation (Sim) applications. Tightly coupled: single executable, one-to-one mapping to accelerator; Semi tightly coupled: separate executable, one-to-one or many-to-many mapping in single machine via ML server (MLS); Loosely coupled: separate executable, many-to-many mapping in different machines.

A. Architecture Overview

Given the trend of increasing number of heterogeneous supercomputers in HPC [18], especially for deep learning oriented systems, we consider the deployment scenarios for the accelerator (e.g., GPU) based machines, which OLCF has deployed at world-leading scales in its recent leadership-class supercomputers.

In terms of the architecture for deploying the surrogate models in simulation, there are three primary types of deployment which are illustrated in Fig. 1:

1) *tightly coupled*, where a single executable is responsible for both simulation (Sim) and machine learning (ML) inference. This often leads to a one-to-one mapping between the process and the accelerator.
2) *loosely coupled*, where Sim and ML server (MLS) executables are running at different communication regions (e.g., different machines). Many accelerators managed by the MLS forms a many-to-many mapping with the Sim processes.
3) *semi tightly coupled*, where Sim and MLS are running separately but on the same machine. Depending on the number of accelerators in a node and performance considerations, it can be either one-to-one, many-to-one, or many-to-many mapping.

We examine how we can map the workflow of MC sampling to these configurations. For most MC applications, the proposal generation (i.e., generation of new configurations of alloy in our case) is usually cheap. Since there is often a sequential dependency (e.g., in Markov Chain MC), the simulation is more suited on CPU with expensive kernels offloaded to the accelerator. For our MC application, the simulation is performed on CPUs and the surrogate model inference is executed on GPUs, and we consider all three types of deployment configurations.

Based on the architecture illustrated in Fig. 1, we discuss various implementation methods, including using DL framework built-in APIs, framework inference serving APIs, and third-party (SmartRedis[19]) serving APIs. The pseudocodes are in TensorFlow/SmartRedis syntax.

B. Framework built-in APIto Include DL Components

Although most developers interact with DL frameworks via a Python interface, the popular frameworks such as TensorFlow and PyTorch do also offer C++ API support. As illustrated in Impl 1, the first step is to use `tensorflow::LoadSavedModel` to load the pre-trained models, assuming it is saved in the `SavedModel` format. Then, the model prediction can be invoked by providing the `Tensor` input data, and the name of input and output nodes, which can be obtained with `saved_model_cli` utility. The key computation kernel for energy evaluation is hence summed over the model inferences for each element in the alloy.

There are several advantages of using a framework's built-in API: 1) it is easy port to different hardware (e.g., from Nvidia GPU to AMD GPU) given the framework's C++ API is readily available, 2) it usually provides the best inference latency because it is compiled into the application code itself, and 3) it is easy to maintain and deploy given the single application executable. On the other hand, such deployments lack deeper flexibility in resource management, and can result in poor utilization and throughput.

C. Inference Server

In addition to the C++ API, DL frameworks also provide inference serving, which is usually utilized in commercial production settings. In HPC, we can take advantage of the inference server to better manage the resources, especially for heterogeneous systems. To use the TensorFlow Serving, the first step is to launch the `tensorflow_model_server` with specified `model_config_file`, which contains the

Implementation 1 Use TensorFlow C++ API

```
1: function LoadModel(string ModelPath)
2:     tensorflow::SessionOptions SessOpt
3:     tensorflow::RunOptions RunOpt
4:     tensorflow::SavedModelBundle Model                ▷ Get model path, and setup session and run options
5:     tensorflow::LoadSavedModel(SessOpt, RunOpt,
           ModelPath, {"serve"}, &Model)                 ▷ load pre-trained model
6:     return Model
7: end function

8: function Predict(tensorflow::SavedModelBundle Model, tensorflow::Tensor Inputs)
9:     tensorflow::Tensor Inputs                         ▷ input tensor
10:    vector<tensorflow::Tensor> Outputs
11:    string InputNode
12:    string OutputNode                                 ▷ input and output nodes
13:    Model.GetSession().Run(data, {OutputNode}, {}, &Outputs)
14:    return Outputs
15: end function

16: function Energy
17:    for i ← 0, Nelements do
18:        Model_i ← LOADMODEL(Path_i)
19:    end for

20:    for i ← 0, Nelements do                           ▷ Add energy by element species
21:        tensorflow::TensorShape DataShape
22:        tensorflow::Tensor Inputs(tensorflow::DT_UINT8, DataShape)
23:        for j ← 0, Nneighbors do
24:            Inputs ← Atom[i][j]
25:        end for
26:        Outputs = PREDICT(Model_i,Inputs)
27:        Energy ← Outputs
28:    end for
29:    return Energy
30: end function
```

Implementation 2 Use TensorFlow Serving

```
1: function Energy
2:     Host ← server_ip : port
3:     for i ← 0, Nelements do
4:         Stub_i ← tensorflow::serving::PredictionService::NewStub(
   grpc::CreateChannel(Host, grpc::InsecureChannelCredentials()))
5:     end for

6:     for i ← 0, Nelements do                           ▷ Add energy by element species
7:         tensorflow::serving::PredictRequest predictRequest
8:         tensorflow::serving::PredictResponse Outputs
9:         grpc::ClientContext context
10:        predictRequest.mutable_model_spec().set_name(Model_i)
11:        predictRequest.mutable_model_spec().set_signature_name("serving_default")
12:        for j ← 0, Nneighbors do
13:            predictRequest.mutable_inputs() ← Atom[i][j]
14:        end for
15:        Stub_i.Predict(&context, predictRequest, &Outputs)
16:        Energy ← Outputs
17:    end for
18:    return Energy
19: end function
```

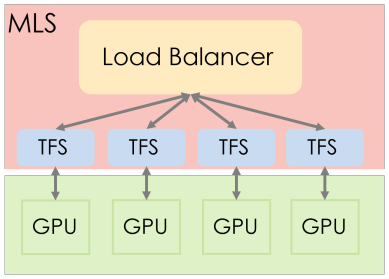


Fig. 2. The inside view of machine learning server (MLS) with load balancer and TensorFlow Serving (TFS).

model name, file path, and platform (e.g., TensorFlow or ONNX). Therefore, in the client code, as shown in Implementation 2, there is no need to load the model and the `PredictionService` API can be directly invoked once a connection (preferably with gRPC, which is generally faster than HTTP) is established with the server. The compilation of the client code, however, is potentially more cumbersome because it is coupled with the server code. Unlike the TensorFlow C++ API where a shared library and header files are provided, the TensorFlow Serving has tighter dependencies on the versions of Protocol Buffers and TensorFlow, hence requires to be first "converted" with Protocal Buffers compiler `protoc` into C++ files and then compiled together with the client code.

In practice, to achieve a better load balance especially in the cases of many-to-many mapping (See Fig. 1), we utilize `HAProxy` to monitor servers, each binding to an accelerator, as shown in Fig. 2, which has been shown [20] to perform well for a Python-based deployment.

The advantages of using TensorFlow Serving include: 1) it provides more flexibility in managing the resources since it decoupled the client from the server processes, 2) it can achieve a better inference throughput, and 3) it can support multiple applications simultaneously. On the other hand, considering the delicate dependencies and compilation, the maintenance cost is relatively high.

D. Third-Party API

Besides framework's utilities, there are third-party libraries such as Nvidia Triton, RedisAI, etc, but they are typically not designed with consideration for the integration with HPC simulation applications. SmartRedis, built upon RedisAI, addresses this particular need with support for C++ and Fortran. In Implementation 3, we show the steps to use SmartRedis in the client code. Similar to TensorFlow Serving, the service needs to be launched with `redis-server`, specifying the port and RedisAI module. The Redis server manages a distributed in-memory database, and RedisAI enables the locality-aware inference operation within the database. Therefore, in the client code, once the model is registered with `set_model_from_file`, there are 3 explicit steps to complete the inference, i.e., transferring the inputs to the server with `put_tensor`, invoking the prediction with `run_model`, and obtaining the results with `unpack_tensor`.

The advantages of SmartRedis includes: 1) it simplifies the integration with HPC applications. 2) it enables the online analytics with the distributed in-memory database. 3) it supports both DL and ML frameworks (e.g., SciKit-Learn). On the other hand, it may not offer the best performance in terms of latency and throughput.

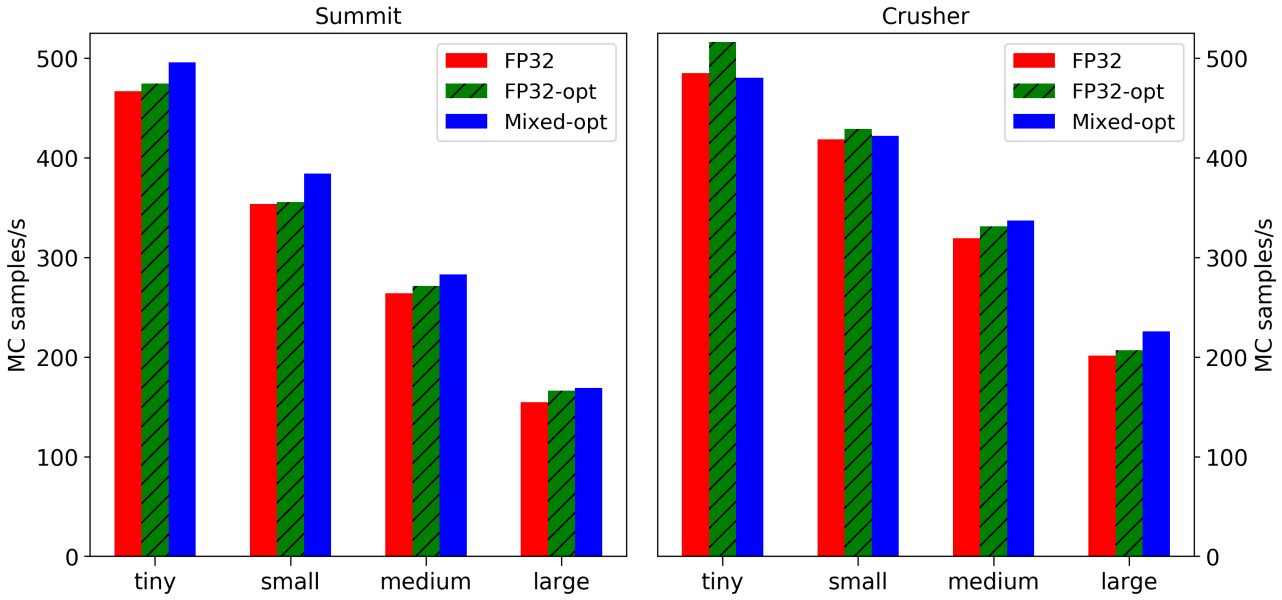


Fig. 3. *Tightly-coupled deployment with TensorFlow built-in API.* The inference throughput for surrogate models (tiny, small, medium, and large) in a Monte Carlo simulation on both Summit (per V100) and Crusher (per GCD of MI250X). Each model is in FP32, FP32 with graph optimization (FP32-opt), and optimized mixed precision (Mixed-opt).

Implementation 3 Use SmartRedis/RedisAI

```
1:  function ENERGY
2:      SSDB ← redis_host_ip : port
3:      SmartRedis::Client* clients[Nelements]
4:      for i ← 0, Nelements do
5:          Client_i.set_model_from_file(model_i, path_i, "TF", "GPU", batch_size,
        mini_batch_size, "hea", {"x"}, {"Identity"})
6:      end for

7:      for i ← 0, Nelements do                    ▷ Add energy by element species
8:          uint8 Data[]
9:          data_key ← "data_i"
10:         outputs_key ← "energy_i"
11:         for j ← 0, Nneighbors do
12:             Data ← Atom[i][j]
13:         end for
14:         Client_i.put_tensor(data_key, Data, NT[t], Nneighbors*(NE-1), SmartRe-
        dis::TensorType::flt, SmartRedis::MemoryLayout::contiguous))
15:         Client_i.run_model(model_i, {data_key}, {outputs_key})
16:         Client_i.unpack_tensor(outputs_key, Outputs.data(), NT[t], SmartRe-
        dis::TensorType::flt, SmartRedis::MemoryLayout::contiguous)
17:         Energy ← Outputs
18:     end for
19:     return Energy
20: end function
```

IV. Evaluation

A. Experiment Setup

Hardware To evaluate the performance of various surrogate-model deployments, we execute the integrated example MC application on both the Summit supercomputer [21] and a early-access testbed system for Frontier called Crusher [22]. Each Summit node is equipped with 6 Nvidia Volta GPUs (V100) and 2 IBM Power9 (P9) CPUs. Pairs of 3 V100s are fully connected with NVLink fabrics of 50 GB/s bandwidth, and nodes are then connected via EDR InfiniBand with a capability of 25 GB/s. Crusher is a development system with an architecture identical to Frontier's but at a smaller scale (192 nodes). Each Crusher node is equipped with 4 AMD Instinct MI250x GPUs (dual Graphics Compute Die (GCD)) and 1 3rd Gen EPYC CPU. All 4 MI250x are connected with each other using 50 GB/s Infinity Fabric, and nodes are connected via Slingshot-11 with 100 GB/s.

Software To compare different deployment strategies, all the evaluations are performed for the same TensorFlow models, although other model formats are also supported by TensorFlow Serving and SmartRedis. TensorFlow V2.4.1 and SmartRedis v0.2.0 along with CUDA v11.0.3 and cuDNN v8.0.4 are utilized on Summit, and TensorFlow v2.5.0 with ROCm v4.5.0 are utilized on Crusher.

The evaluation metrics include the *inference throughput* (aggregated MC samples per unit time, MC samples/s) and *latency* (time taken for an MC update), as well as the *scaling efficiency* in terms of the surrogate model inference and *time-to-solution* for the overall application. The following studies cover all three types of implementation approaches (i.e., Implementation 1,2, and 3) under four deployment architectures (See Fig. 1).

B. Results

Since the communication is among simulation processes, and ML clients and inference servers, we focus our attention on the single node performance for one-to-one deployment of both tightly and semi-tightly coupled design. Then we evaluate the cross-node performance for loosely coupled architecture, and finally scale up the application to over 1000 GPUs.

Tightly-coupled deployment In this scenario, the simulation process manage the inference directly and each process binds to a GPU (one-to-one). Following the Implementation 1, in Fig. 3, the per device throughputs (in MC samples/s)

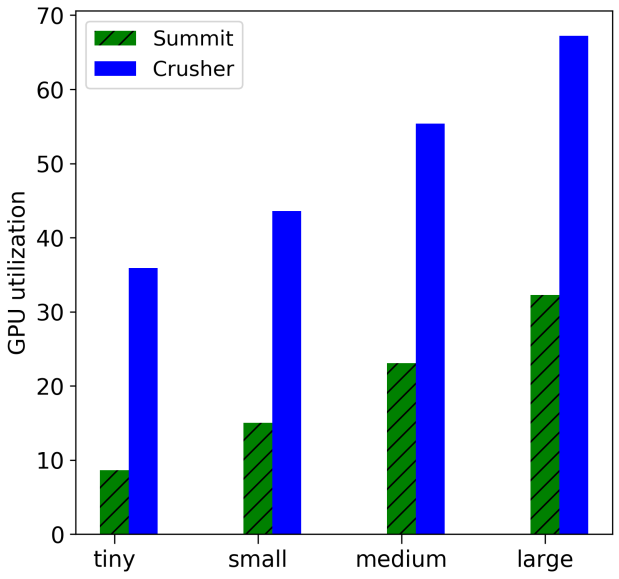


Fig. 4. *Tightly-coupled deployment with TensorFlow built-in API.* The GPU utilization for surrogate models (tiny, small, medium, and large) in a Monte Carlo simulation on both Summit and Crusher.

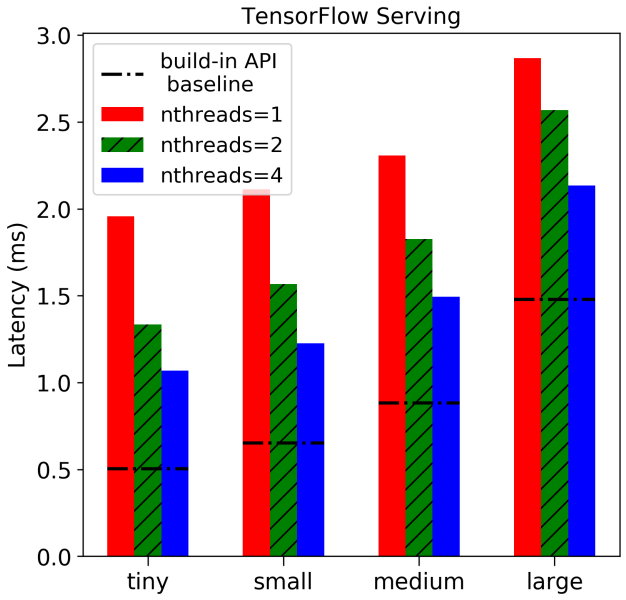


Fig. 5. *Semi tightly-coupled deployment (one-on-one mapping) with TensorFlow Serving.* The latency for surrogate models (tiny, small, medium, and large) in a Monte Carlo simulation on Summit. Models (mixed precision) are concurrently executed with 1, 2, and 4 OpenMP threads.

are plotted for both Summit and Crusher. With model optimization, including freezing and optimizing the computational graph and using mixed precision, about 10% improvement are obtained for these types of MLP models. As the model size increases 3 orders of magnitude from thousands (tiny) to millions (large), the throughput decreases to about 1/3. For every model size, Crusher (per GCD of MI250X) outperforms Summit (per V100) by up to 15%, hence the overall speedup of MI250X is about 2.3X, which is reflected in the GPU utilization as well, as shown in Fig. 4. Note that the thermodynamic quantities evaluated with the same model on both platform, and the outputs are identical.

Semi tightly coupled deployment (one-to-one mapping) In this scenario, the inference is managed by a separated service (Implementation 2 and 3), and the simulation process sends the inference request via the client API. Comparing to the single-process tightly coupled approach, the client-server setup can incur additional overhead even when each client has a dedicated server. To improve the latency, we take advantage of client-server concurrency by adding OpenMP threading so all atom species (4 for MoNbTaW alloy) inferences can be executed simultaneously. Note that this is not specific to our MC application because the batch dimension can be generally divided into mini batches to increase the concurrency. As shown in Fig. 5, adding more clients for TensorFlow Serving significantly improve the overall latency for the energy function evaluation, but comparing to the tightly coupled case, it takes up to twice of time even though the difference decreases as the model size increases. For SmartRedis (see Fig. 6), the latency is generally better than TensorFlow Serving, and can achieve about 85% of the best performance. In Fig. 7, we further break down the

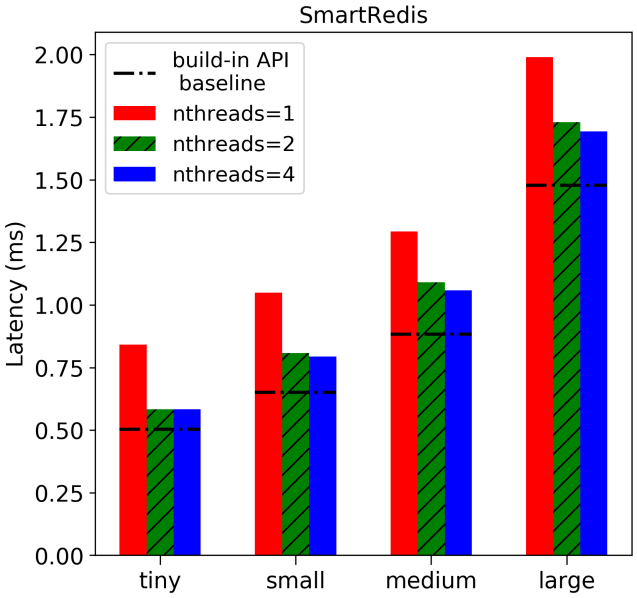


Fig. 6. *Semi tightly-coupled deployment (one-to-one mapping) with SmartRedis.* The latency for surrogate models (tiny, small, medium, and large) in a Monte Carlo simulation on Summit. Models (mixed precision) are concurrently executed with 1, 2, and 4 OpenMP threads.

timing for each step of an inference invocation, i.e., sending input data (`put_tensor`), executing model (`run_model`), and retrieving the result (`unpack_tensor`). As model size increases, the time for data transfer remains constant because the input and output data size is fixed, and the percentage on the model inference increases. For models with hundreds of millions or more parameters, the cost of same node data

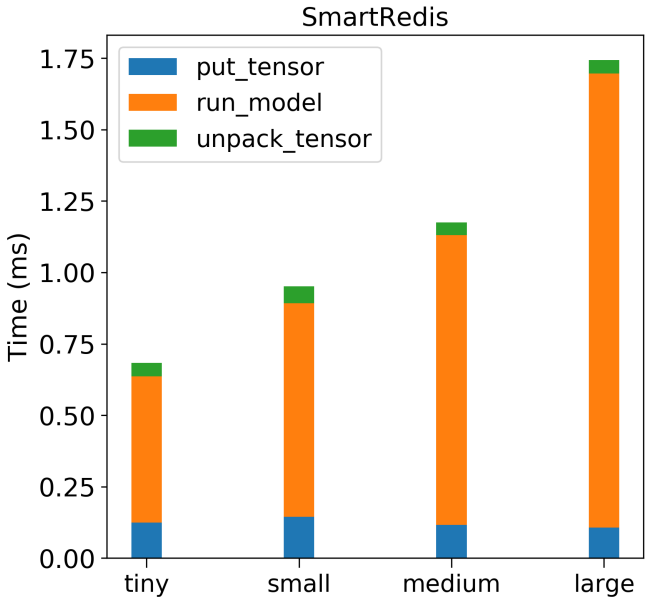


Fig. 7. *Semi tightly-coupled deployment (one-to-one mapping) with SmartRedis.* The timing breakdown of a inference for surrogate models (tiny, small, medium, and large) in a Monte Carlo simulation on Summit.

transfer can be negligible.

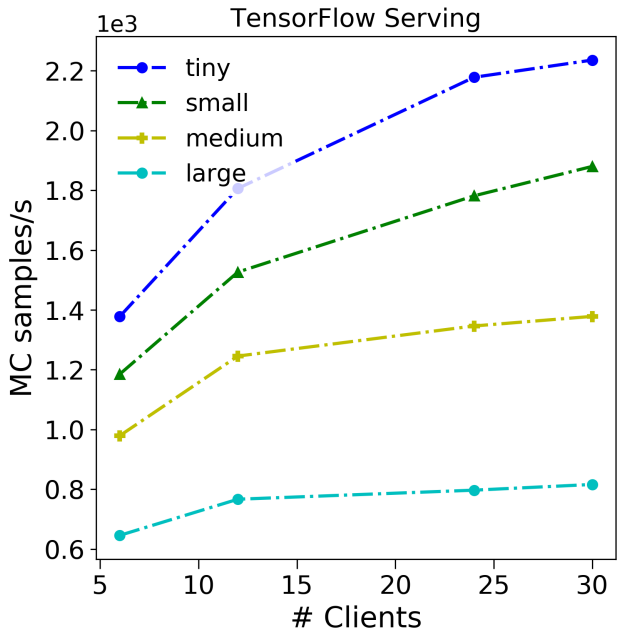


Fig. 8. *Semi tightly-coupled deployment (many-to-many mapping) with TensorFlow Serving.* The inference throughput for surrogate models (tiny, small, medium, and large) in a Monte Carlo simulation on a single node of Summit. The inference services are running on all 6 GPUs with HAProxy for load balancing. The simulation is running with 6, 12, 24, and 30 ranks (clients).

Semi tightly coupled deployment (many-on-many mapping) In this scenario, resources are managed in physically closed (e.g., same node) two separated groups: one for simulation processes and the other for inference services. The clients send inference requests to a load balancer (`HAProxy` in our setup), which then evenly dispatches the workload to available GPUs. This is useful to increase the overall application throughput. In Fig. 8, we plot the throughput for simulating up to 30 MC sampling processes. For larger models (e.g., `large`), the aggregated throughput quickly plateaus, while for smaller models, the throughput keeps increasing, indicating server resources not yet saturating.

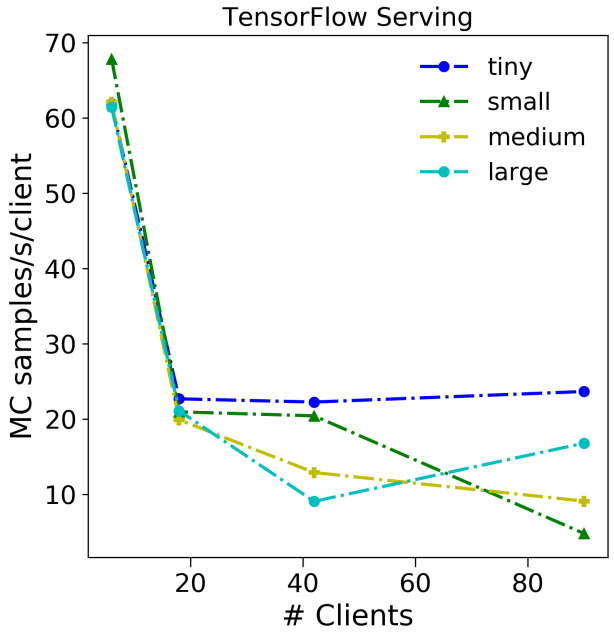


Fig. 9. *Loosely-coupled deployment with TensorFlow Serving.* The inference throughput for surrogate models (tiny, small, medium, and large) in a Monte Carlo simulation on Summit up to 16 nodes (6 simulation clients per nodes). The inference services are running on one node with HAProxy for load balancing. The simulation is running with the rest of nodes.

Loosely coupled deployment In this scenario, a dedicated resource (one node) is reserved for the inference services, while the simulation is executing on separated nodes, hence the inference requests need to be send across nodes. This is suitable for a HPC platform with different partitions (e.g., CPU nodes and GPU nodes). In Fig. 9, we plot the per client throughput up to 96 simulation clients (16 nodes on Summit). There is a steep penalty for transferring input data and output predictions across nodes, but the throughput seems to be stabilised after 18 clients with some network fluctuations.

Scaling For a thermodynamics simulation of fixed MC sampling steps, the application with surrogate models is bound by the inference latency. Therefore, we employ the tightly coupled deployment to show the overall application performance. In Fig. 10 and 11, we plot the aggregated throughput and time-to-solution up to 1000 GPUs on Summit and Crusher, respectively. As expected, the inference throughput scales linearly. The weak scaling of the time-to-solution is about 90% on Crusher and 70% on Summit with large fluctuations, which is likely due to network traffic on the production system.

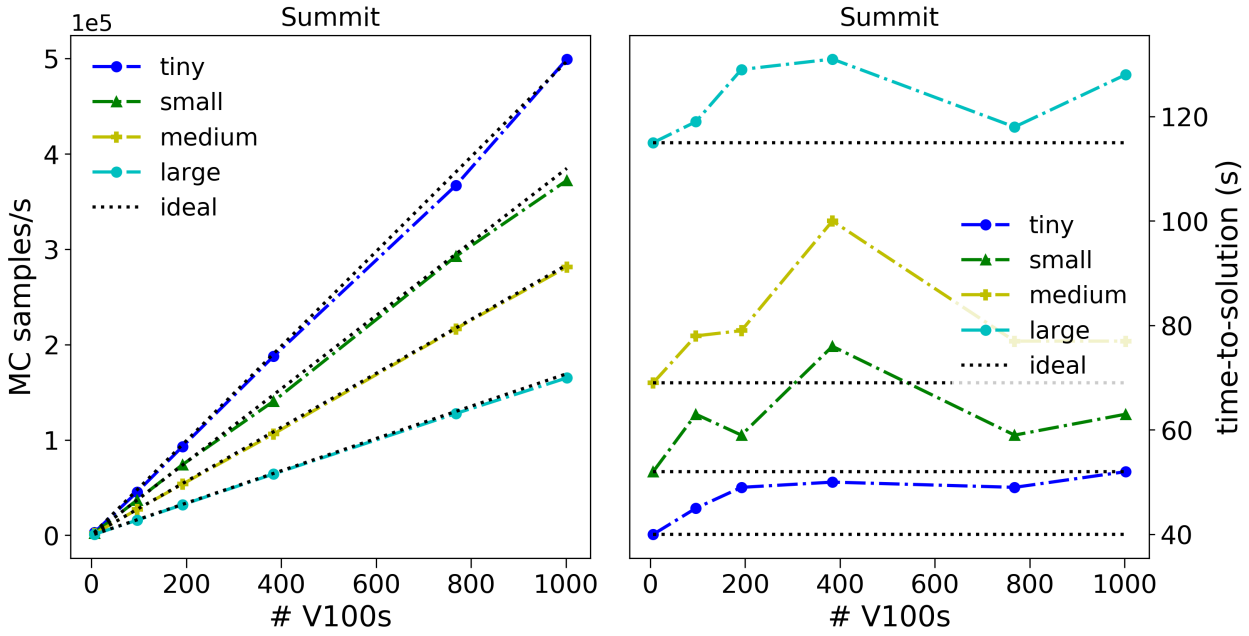


Fig. 10. *Tightly-coupled deployment with TensorFlow built-in API.* The scaling of both inference throughput and time-to-solution for surrogate models (tiny, small, medium, and large) in a Monte Carlo simulation on Summit up to 1000 V100s.

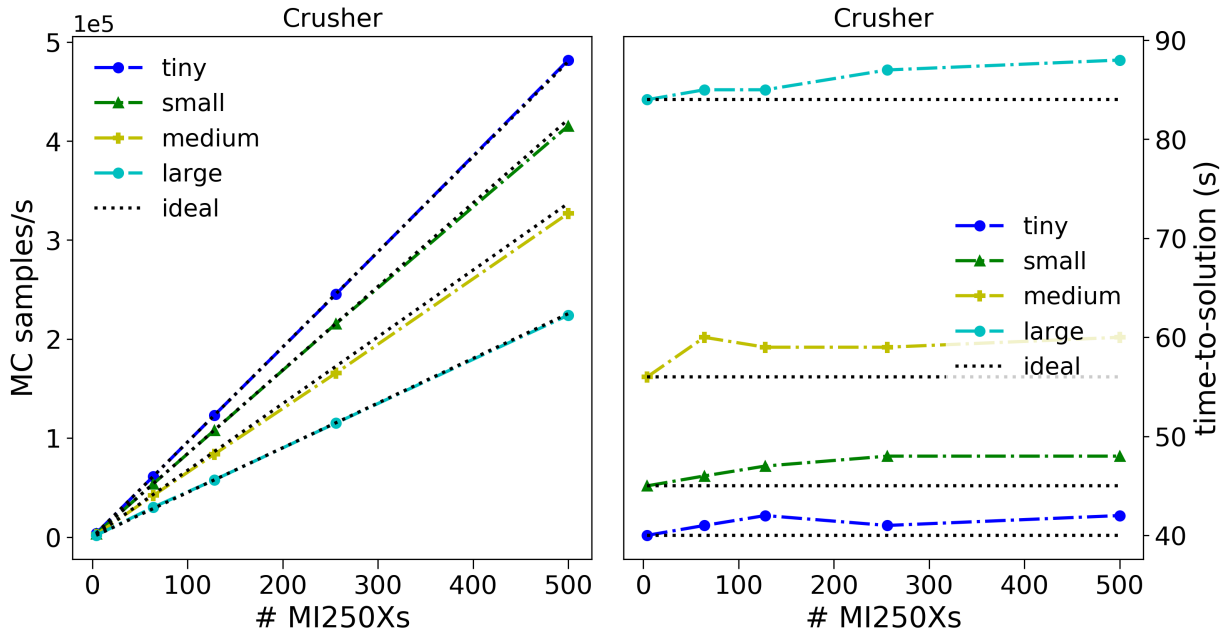


Fig. 11. *Tightly-coupled deployment with TensorFlow built-in API.* The scaling of both inference throughput and time-to-solution for surrogate models (tiny, small, medium, and large) in a Monte Carlo simulation on Crusher up to 500 MI250Xs (1000 effective GPUs).

V. Discussion

Although our evaluation is based on a Monte Carlo application, the observations we draw are generally applicable to campaigns that are set up as simulation phases exchanging data with an ML/DL module for inference. Table II summarizes the key points we detail below.

Usability Most developers perform DL modeling in Python, and it is not straightforward to deploy the pre-trained models in the C++/Fortran simulation applications. The DL framework built-in C++ API is relatively easy to use, and provides good portability because of good support from hardware vendors (e.g., TensorFlow-ROCm on AMD GPUs) given DL's popularity. The exact same model trained in *Python* using *NVidia GPUs* can be directly utilized on a *C++* application on *AMD*

TABLE II

OVERVIEW OF THE CHARACTERISTICS OF DIFFERENT DEPLOYMENT METHODS FOR SURROGATE MODELING IN SIMULATIONS. THE LEVELS OF USABILITY AND PERFORMANCE ARE INDICATED BY FILLED CIRCLES, WITH THE FULLY FILLED ● BEING THE BEST.

deployment method	usability		performance		
	portability	flexibility	concurrency	latency	throughput
TensorFlow C++ API	●	◑	◑	●	◑
TensorFlow Serving	◑	◕	●	◑	◕
SmartRedis/RedisAI	◑	●	●	◕	●

GPUs without any changes. The inference server, framework or third-party, on the other hand, may require additional porting to support on different hardware and/or software versions. In terms of flexibility, the inference server does support multiple deployment scenario such as many-to-many resource mapping and multiple model formats, etc. Third-party server such as SmartRedis can even support traditional ML models for a Fortran application. It offers more opportunities for resource optimization especially for heterogeneous platforms and broader applicability.

Performance Comparing the Python interface, the framework C++ API provides limited performance tuning support. But if the application is latency-bound and there are sufficient resources such as GPUs available, good performance can be straightforwardly obtained via a one-to-one tightly-coupled deployment. On the other hand, if aggregate throughput is desired and/or the compute nodes on the platform are non-homogeneous (e.g., CPU and GPU partitions), then inference servers can perform better. Note that by taking advantage of concurrent inferences, SmartRedis may achieve an effective latency close to the framework API, and hence provide the overall best performance by comparison.

VI. CONCLUSION

We have performed a broad suite of evaluations on integrating deep learning surrogate models with simulations on HPC platforms. We considered different deployment architectures, including tightly-coupled, semi tightly-coupled, and loosely-coupled approaches, and evaluated their performances in terms of throughput, latency, scaling, and application time-to-solution. For latency-bound applications such as Monte Carlo simulations, tightly-coupled deployment is desired. We presented three types of implementations with associated pseudo-code for integrating surrogate models with simulations, including the framework built-in API, framework inference server, and a third-party API approach. The third-party API with SmartRedis provides good usability and better overall performance when inference requests are made by parallel executing threads. We showed a real application use case with surrogate modeling for thermodynamics calculation in materials science, and evaluated the performance with four models with sizes ranging from a few thousand to millions of parameters. These can serve as baseline results when considering the cost of deploying various sizes of models. Finally, we compared the performance of surrogate model incorporation on two platforms and DL stacks: Summit with Nvidia V100 on CUDA stack and Crusher with AMD MI250X on the AMD ROCm stack. Crusher shows a ∼2.3x speedup per device on inference, and better overall scaling for the application.

ACKNOWLEDGMENT

This research was sponsored by and used resources of the Oak Ridge Leadership Computing Facility (OLCF), which is a DOE Office of Science User Facility at the Oak Ridge National Laboratory supported by the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. The publisher, by accepting the article for publication, acknowledges that the U.S. Government retains a non-exclusive, paid up, irrevocable, world-wide license to publish or reproduce the published form of the manuscript, or allow others to do so, for U.S. Government purposes. The DOE will provide public access to these results in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

REFERENCES

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. URL https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[2] Thorsten Kurth, Sean Treichler, Joshua Romero, Mayur Mudigonda, Nathan Luehr, Everett Phillips, Ankur Mahesh, Michael Matheson, Jack Deslippe, Massimiliano Fatica, Prabhat, and Michael Houston. Exascale deep learning for climate analytics. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18. IEEE Press, 2018.

[3] Nouamane Laanait, Joshua Romero, Junqi Yin, M. Todd Young, Sean Treichler, Vitalii Starchenko, Albina Borisevich, Alex Sergeev, and Michael Matheson. Exascale deep learning for scientific inverse problems. *ArXiv*, abs/1909.11150, 2019.

[4] Drew Schmidt, Bronson Messer, M. Todd Young, and Michael Matheson. Towards the development of entropy-based anomaly detection in an astrophysics simulation. *arXiv*, 2009.02430, 2020.

[5] Hyungro Lee, Matteo Turilli, Shantenu Jha, Debsindhu Bhowmik, Heng Ma, and Arvind Ramanathan. Deepdrivemd: Deep-learning driven adaptive molecular simulations for protein folding. In *2019 IEEE/ACM Third Workshop on Deep Learning on Supercomputers (DLS)*, pages 12–19, 2019. doi: 10.1109/DLS49591.2019.00007.

[6] Andrew E Blanchard, John Gounley, Debsindhu Bhowmik, Mayanka Chandra Shekar, Isaac Lyngaas, Shang Gao, Junqi Yin, Aristeidis Tsaris, Feiyi Wang, and Jens Glaser. Language models for the prediction of sars-cov-2 inhibitors. *bioRxiv*, 2021. doi: 10.1101/2021.12.10.471928.

[7] Francesco Di Natale, Harsh Bhatia, Timothy S. Carpenter, Chris Neale, Sara Kokkila-Schumacher, Tomas Oppelstrup, Liam Stanton, Xiaohua Zhang, Shiv Sundram, Thomas R. W. Scogland, Gautham Dharuman, Michael P. Surh, Yue Yang, Claudia Misale, Lars Schneidenbach, Carlos Costa, Changhoan Kim, Bruce D'Amora, Sandrasegaram Gnanakaran, Dwight V. Nissley, Fred Streitz, Felice C. Lightstone, Peer-Timo Bremer, James N. Glosli, and Helgi I. Ingólfsson. A massively parallel infrastructure for adaptive multiscale simulations: Modeling ras initiation pathway for cancer. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362290. doi: 10.1145/3295500.3356197.

[8] Weile Jia, Han Wang, Mohan Chen, Denghui Lu, Lin Lin, Roberto Car, Weinan E, and Linfeng Zhang. Pushing the limit of molecular dynamics with ¡i¿ab initio¡/i¿ accuracy to 100 million atoms with machine learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020. ISBN 9781728199986.

[9] Jeongnim Kim, Andrew D Baczewski, Todd D Beaudet, Anouar Benali, M Chandler Bennett, Mark A Berrill, Nick S Blunt, Edgar Josué Landinez Borda, Michele Casula, David M Ceperley, Simone Chiesa, Bryan K Clark, Raymond C Clay, Kris T Delaney, Mark Dewing, Kenneth P Esler, Hongxia Hao, Olle Heinonen, Paul R C Kent, Jaron T Krogel, Ilkka Kylnp, Ying Wai Li, M Graham Lopez, Ye Luo, Fionn D Malone, Richard M Martin, Amrita Mathuriya, Jeremy McMinis, Cody A Melton, Lubos Mitas, Miguel A Morales, Eric Neuscamman, William D Parker, Sergio D Pineda Flores, Nichols A Romero, Brenda M Rubenstein, Jacqueline A R Shea, Hyeondeok Shin, Luke Shulenburger, Andreas F Tillack, Joshua P Townsend, Norm M Tubman, Brett Van Der Goetz, Jordan E Vincent, D ChangMo Yang, Yubo Yang, Shuai Zhang, and Luning Zhao. QMCPACK: an open sourceab initioquantum monte carlo package for the electronic structure of atoms, molecules and solids. *Journal of Physics: Condensed Matter*, 30(19):195901, apr 2018. doi: 10.1088/1361-648x/aab9c3.

[10] Maurcio Moralles, Carla C. Guimares, Daniel A. B. Bonifcio, Emico Okuno, Hlio M. Murata, Mrcio Bottaro, Mrio O. Menezes, and Valdir Guimares. Applications of the monte carlo method in nuclear physics using the geant4 toolkit. *AIP Conference Proceedings*, 1139(1): 51–56, 2009. doi: 10.1063/1.3157829.

[11] Junqi Yin and Zongrui Pei. Neural network-based order parameter for phase transitions and its applications in high-entropy alloys. *Nature Computational Science*, 1: 1–8, 10 2021. doi: 10.1038/s43588-021-00139-3.

[12] Massimiliano Lupo Pasini, Ying Wai Li, Junqi Yin, Jiaxin Zhang, Kipton Barros, and Markus Eisenbach. Fast and stable deep-learning predictions of material properties for solid solution alloys. *Journal of Physics: Condensed Matter*, 33(8):084005, dec 2020. doi: 10.1088/1361-648x/abcb10.

[13] Xianglin Liu, Jiaxin Zhang, Junqi Yin, Sirui Bi, Markus Eisenbach, and Yang Wang. Monte carlo simulation of order-disorder transition in refractory high entropy alloys: A data-driven approach. *Computational Materials Science*, 187:110135, 2021. ISSN 0927-0256. doi: https://doi.org/10.1016/j.commatsci.2020.110135.

[14] E.P. George, D. Raabe, and R.O. Ritchie. High-entropy alloys. *Nature Review Materials*, 4:515–534, 8 2019. doi: 0.1038/s41578-019-0121-4.

[15] David J. Earl and Michael W. Deem. Parallel tempering: theory, applications, and new perspectives. *Physical chemistry chemical physics : PCCP*, 7 23:3910–6, 2005.

[16] Suffian Khan and Markus Eisenbach. Density-functional monte-carlo simulation of cuzn order-disorder transition. *Physical Review B*, 93, 10 2015. doi: 10.1103/PhysRevB.93.024203.

[17] James Kirkpatrick, Brendan McMorrow, David H. P. Turban, Alexander L. Gaunt, James S. Spencer, Alexander G. D. G. Matthews, Annette Obika, Louis Thiry, Meire Fortunato, David Pfau, Lara Romn Castellanos, Stig Petersen, Alexander W. R. Nelson, Pushmeet Kohli, Paula Mori-Snchez, Demis Hassabis, and Aron J. Cohen. Pushing the frontiers of density functionals by solving the fractional electron problem. *Science*, 374(6573):1385–1389, 2021. doi: 10.1126/science.abj6511.

[18] Awais Khan, Hyogi Sim, Sudharshan S. Vazhkudai, Ali R. Butt, and Youngjae Kim. An analysis of system balance and architectural trends based on top500 supercomputers. In *The International Conference on High Performance Computing in Asia-Pacific Region*, HPC Asia 2021, page 1122, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450388429. doi: 10.1145/3432261.3432263. URL https://doi.org/10.1145/3432261.3432263.

[19] Sam Partee, Matthew Ellis, Alessandro Rigazzi, Scott Bachman, Gustavo Marques, Andrew Shao, and Benjamin Robbins. Using machine learning at scale in hpc simulations with smartsim: An application to ocean climate modeling, 2021.

[20] Wesley Brewer, Daniel Martinez, Mathew Boyer, Dylan Jude, Andy Wissink, Ben Parsons, Junqi Yin, and Valen-

tine Anantharaj. Production deployment of machine-learned rotorcraft surrogate models on hpc. In *2021 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC)*, pages 21–32, 2021. doi: 10.1109/MLHPC54614.2021.00008.

[21] OLCF. Summit. https://docs.olcf.ornl.gov/systems/summit_user_guide.html, 2022. Accessed: 2022-01-20.

[22] OLCF. Crusher. https://docs.olcf.ornl.gov/systems/crusher_quick_start_guide.html, 2022. Accessed: 2022-01-20.