Journal of Computational Science 62 (2022) 101707

Contents lists available at ScienceDirect

Journal of Computational Science

journal homepage: www.elsevier.com/locate/jocs

Using Machine Learning at scale in numerical simulations with SmartSim: An application to ocean climate modeling

Sam Partee [a,*], Matthew Ellis [a], Alessandro Rigazzi [b], Andrew E. Shao [c], Scott Bachman [d], Gustavo Marques [d], Benjamin Robbins [a]

[a] Hewlett Packard Enterprise, Seattle, WA, USA
[b] Hewlett Packard Enterprise, Switzerland
[c] Canadian Centre for Climate Modelling and Analysis, Victoria, BC, Canada
[d] National Center for Atmospheric Research, Boulder, CO, USA

ARTICLE INFO

Keywords:
Deep learning
Numerical simulation
Climate modeling
High performance computing
SmartSim

ABSTRACT

We demonstrate the first climate-scale, numerical ocean simulations improved through distributed, online inference of Deep Neural Networks (DNN) using SmartSim. SmartSim is a library dedicated to enabling online analysis and Machine Learning (ML) for high performance, numerical simulations. In this paper, we detail the SmartSim architecture and provide benchmarks including online inference with a shared ML model, EKE-ResNet, on heterogeneous HPC systems. We demonstrate the capability of SmartSim by using it to run a 12-member ensemble of global-scale, high-resolution ocean simulations, each spanning 19 compute nodes, all communicating with the same ML architecture at each simulation timestep. In total, 970 billion inferences are collectively served by running the ensemble for a total of 120 simulated years. The inferences are used to predict the oceanic eddy kinetic energy (EKE), which is a variable that is used to tune different turbulence closures in the model and thus directly affects the simulation. The root-mean-square of the error in EKE (as compared to an eddy-resolving simulation) is 20% lower when using the ML-prediction than the previous state of the art. This demonstration is an example of how machine learning methods can be integrated into traditional numerical simulations, replace prognostic equations, and preserve overall simulation stability without significantly affecting the time to solution.

1. Introduction

Advances in machine-learning (ML) algorithms have spurred research and development for combining data-driven approaches and traditional numerical simulations to improve both efficiency and accuracy. The codebases of these numerical models are typically written in Fortran/C/C++ and run on high-performance computing platforms (HPC) via OpenMP and/or MPI parallelization. New software solutions are thus needed to connect these compiled language codebases to rapidly evolving ML and data analytics libraries, typically written in Python. Currently, the diversity of programming languages, dependence on file input/output (I/O), and large variance in compute resource requirements for scientific applications makes it difficult to perform analysis, training, and inference with most ML and data analytics packages at the scale needed for HPC numerical simulations.

On its surface, the problem of being able to interface HPC applications with ML libraries is one of language interoperability and software interface design. However, for the full convergence of these two disparate paradigms, the true difficulty (and opportunity) in bridging these workloads needs to be reformulated in terms of data exchange. That is, how is data passed between a simulation and ML model at scale while making efficient use of heterogeneous computational resources? Current approaches to addressing this problem can be roughly broken down into two categories: offline (the ML and numerical components of a simulation do not exchange data directly) and online (the ML component is called while the simulation is running). Note that in this work, this definition of "online" pertains to the process of inferring from a trained machine learning model, not continuously updating ML model parameters which is sometimes referred to as "online learning".

To illustrate the differences between online and offline approaches, we review recent studies that couple ML and numerical models with a focus on computational fluid dynamics (CFD) and climate modeling domains due to the application presented in this work.

* Corresponding author.
E-mail address: sam@partee.io (S. Partee).

https://doi.org/10.1016/j.jocs.2022.101707
Received 2 September 2021; Received in revised form 30 March 2022; Accepted 9 May 2022
Available online 18 May 2022

1.1. Offline vs. online machine-learning applications

Offline ML surrogate modeling is the process by which a ML model is trained on data previously generated by a simulation. The surrogate is validated through the incorporation of surrogate inference data into a simulation through the filesystem. In this paradigm, a simple workflow would be to run the numerical simulation, store the output to disk, and train/validate a ML model on the stored result. Online modeling, in contrast, runs ML training and/or inference concurrently with the numerical simulation itself. This can lead to several advantages including higher spatiotemporal sampling of the simulation and reducing file I/O-related latency. Perhaps most importantly, it allows ML to directly influence the simulation itself.

Some ML libraries (e.g. Tensorflow and PyTorch) provide compiled language APIs so that ML can be "hard-coded" into a simulation. For example, this approach has been used to enable ML solutions in a C++ based numerical CFD model, OpenFOAM [1], by compiling in the C-based APIs for PyTorch [2] and TensorFlow [3]. However, these popular libraries do not include Fortran APIs and so this approach cannot be followed for the myriad of numerical models written in that language. While recent additions to the Fortran standard have formalized Fortran/C interoperability, the simulations themselves are often not written with such interoperability in mind, potentially requiring large refactors of simulation codebases and necessitating developers who are conversant in both C and Fortran. In addition, ML development generally evolves more quickly than the numerical simulations due to the large-scale investment by industry and broader general interest. Developers and maintainers of numerical simulations must then divert their own relatively limited resources to maintain compatibility with ML libraries.

Other approaches use the file-system as an intermediary between training a ML model and the inference process from within a simulation. For example, [4] uses a random-forest to replace a parameterization of atmospheric convection in a climate model. The random forest model is trained offline and saved into NetCDF files which are loaded and called in the simulation by a custom Fortran module.

Another approach, Fortran Keras Bridge (FKB) [5], enables the usage of Keras-based ML models for online inference in Fortran. FKB builds on a NN library called Neural Fortran which implements a subset of modern ML methods in Fortran. To utilize FKB, Keras-based models must be trained, saved to file, and then loaded into Fortran simulations at runtime. These approaches are specific to Fortran and certain ML models/libraries, and involve similar requirements to embedding the TensorFlow or Pytorch C API into a simulation. Both approaches also lack the ability to utilize co-located or adjacent heterogeneous compute (CPU and GPU) capabilities that are becoming more prevalent in new HPC systems.

Another approach is rewrite the simulations (or portions thereof) in a ML-friendly language. For example, the CliMA project [6] relies heavily on the Julia programming language which by design generally has interoperability with Python. Given time and development resources, re-writing simulations in ML-friendly, performance oriented (e.g. not Python) languages is recognized as a potential avenue for the inclusion of ML in simulation at scale; however, many simulation codebases, developed over decades of research, are not as amenable to porting to a new language.

In a study combining ML and the Finite Volume version 3 atmospheric generical circulation model [7], a Python wrapper was written to call the main timestepping routines. This approach has the advantage that the integration of the simulation can be controlled from Python. Additionally, they also provided interfaces to set and receive the state of the model via Python–Fortran interfaces allowing for the direct diagnosis of the simulation state during the course of the simulation (e.g. online analysis). Writing these interfaces however required significant and specialized technical expertise. An additional complication of using a Python driver to interface with the underlying Fortran model is that Python's library management system scales poorly when hundreds or thousands of clients are importing libraries stored on a network-mounted filesystem [8] (containers can partly mitigate this problem but add complexity to the running of the application and may not supported on all HPC platforms).

1.2. Motivating the use of ML for oceanic turbulence

Climate change poses an ongoing and escalating threat to social and economical welfare, and has been characterized by the United Nations as the defining issue of our time [9]. Beginning with [10], numerical climate models (comprised of physical and biogeochemical models of the atmosphere, ocean, land, and cryosphere) have become an invaluable tool to understand both historical and future change in the Earth's climate. Both numerical models and observational studies confirm that the ocean is the primary sink of the excess heat and carbon dioxide associated with climate change. It is responsible for the delay in realizing the global warming expected from our present CO_2 concentrations [11] and absorbing between 30 and 40% of anthropogenic carbon emitted since the dawn of the industrial age [12,13].

The ocean component of these climate models represents a significant computational expense which is primarily a function of the model's spatial resolution. Of the models submitted to the most recent Coupled Model Intercomparison Project [14], most are typically run at spatial resolutions of O(100 km) with timesteps of O(1 h) to accommodate experiments that span centuries to millennia of simulated time. However, these spatial resolutions do not resolve the large-scale, turbulent structures, known as mesoscale eddies, which dominate the ocean's kinetic energy field and transport heat, salt and biogeochemical tracers [15–18]. Climate-scale ocean general circulation models (OGCMs) would need to be run at roughly 8 to 16 times higher resolution than present models to resolve mesoscale eddies, increasing the computational cost by a factor of 512 to 4096. Accurate and robust turbulence *parameterizations*, which estimate the effects of the unresolved eddies, are crucial for providing skillful representations of the ocean [19] and the climate system as a whole.

While a number of ML applications have been envisioned for the climate domain [20], one particular avenue of active research is the use of ML as a surrogate model for turbulence parameterization or simulation component emulation (e.g. [21–24]). A more recent study [25] trained a deep neural network on one year's worth of data from the Super Parameterized Community Atmospheric Model version 3.0 to generate a ML-based parameterization of cumulus, deep convection. Many of these studies involved training a ML model on simulation data offline to create a surrogate model for a parameterization. A primary motivation of these studies is to reduce the time-to-solution as compared to reference simulations (used for training data) by replacing a numerically expensive portion of the simulation with an ML surrogate. However one of the major restrictions of the offline approach is that it is necessarily a nonconcurrent workflow, that is the ML and numerical simulations are completely decoupled, inhibiting the ability of each component to influence the other. In addition, for large scale simulations using file I/O storing diagnostics of the simulation can itself become a bottleneck, particularly for large-scale simulations.

In addition to the problem of language incompatibility discussed previously, the implementation of ML models into numerical models of the climate system have been impeded due to numerical concerns. The PDEs that govern that atmosphere and ocean are quite stiff and so errors and noise from ML predictions can rapidly develop into numerical instabilities. Two recent studies [26,27] embed machine-learning models into models of oceanic flow by directly influencing the momentum equations. While these simulations were successful, they were done in idealized configurations for a simplified set of equations as compared to the realistic simulations used in climate simulations. In this study, we pursue a different approach by augmenting an existing, widely-used used parameterization of ocean turbulence with

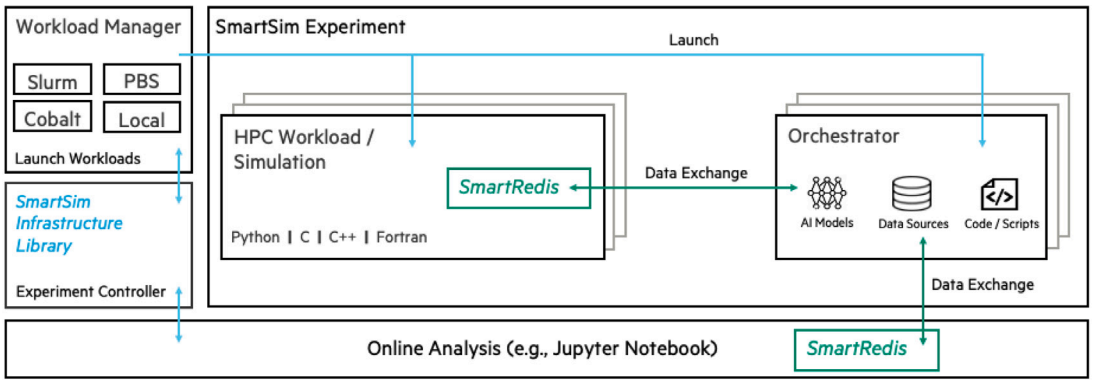

Fig. 1. The architecture for SmartSim is provided for a given use case. In this instance, the Infrastructure Library (IL) is being used to launch the Orchestrator alongside a simulation embedded with the SmartRedis clients. In addition to communication with the simulation, the Orchestrator is also sending data to an analysis environment for online analysis, visualization and/or training of a machine learning model.

a machine-learning model. This approach allows us to improve the accuracy of the parameterization while also relying on a wealth of research spanning 30 years that helps us account for failure modes that might arise from ML errors. By using SmartSim, we are able to run this simulation online with the rest of the simulation, exchanging data between the MPI ranks and the databases used for inference, with minimal computational cost.

1.3. Outline

In this paper, we describe a new software framework SmartSim that enables the convergence of numerical simulations and ML workloads on heterogeneous architectures. Synthetic scaling studies are performed to demonstrate the performance of the system. We then embed SmartSim into an ocean climate model that uses online inference to augment an existing turbulence parameterization. The computational performance and stability of this approach is evaluated by integrating a 12-member ensemble of global ocean simulations for 10 simulated years. Finally, we conclude by discussing the broader applicability of SmartSim beyond online ML inference.

2. SmartSim architecture

SmartSim is comprised of two libraries:

- SmartSim Infrastructure Library: A Python-based workflow library that facilitates dynamic execution of HPC simulations and ML infrastructure
- SmartRedis: A lightweight client library used in applications to communicate with infrastructure started by SmartSim.

A representative workflow which utilizes both of these is demonstrated in Fig. 1.

2.1. SmartSim infrastructure library

The Infrastructure Library (IL) is a framework for simplifying the deployment of complex, multi-stage experiments on HPC systems. In basic terms, the IL provides a Python interface where users can define their workload and launch it on HPC systems. Instead of static configuration files, the IL provides methods to create Python objects that encapsulate workflow components which can be used to start, stop and monitor workloads from interactive mediums like Jupyter Notebooks. By controlling the creation and execution of workloads from a Python interface, the IL enables dynamic customization and augmentation of workloads which is difficult to achieve through HPC batch schedulers, like Slurm, alone.

The `Experiment` object is the primary user interface of the IL. Experiments are used to create references to HPC applications referred to as a `Model`. Model instances contain all parameters for executing a given application on a system, such as compute resource parameters, model parameters, and input files. In addition to models, users can create `Ensemble` objects which are collections of `Model` instances that are treated as a single workload.

Once created, `Model` and `Ensemble` instances are used as references to workloads that can be started, monitored, stopped, and restarted through the `Experiment` interface. The experiment allows users to launch workloads asynchronously such that after applications are successfully launched, users can interactively monitor and analyze their workload.

2.1.1. Orchestrator

A core capability of the IL is launching the Orchestrator. The Orchestrator is a key–value database that can be used to stage data between components of a workload. The Orchestrator is based on Redis [28] but is compatible with any database that uses the Redis API and module system. The Orchestrator can be deployed on separate compute hosts from the application (standard deployment) or placed on the same compute hosts as a `Model` instance (co-located deployment) when launched through the IL.

The Orchestrator resides in-memory and can be distributed across compute hosts providing low-latency access to many clients in parallel. User-defined `Models` created by the IL can be coupled, meaning exchange data, at runtime by passing data through the Orchestrator using SmartRedis Clients. This loosely coupled paradigm, similar to the architecture of client–server microservices, allows Fortran, C, C++ and Python applications to communicate at scale without writing to the filesystem or using tightly coupled MPI communications such as a shared MPI communicator.

In addition to providing data staging between coupled applications, the Orchestrator provides ML capability to HPC workloads at runtime. ML models can be stored in the Orchestrator and served to HPC applications. Despite being written in Python, all models are executed in a C runtime on CPU or GPU. The Orchestrator supports models written with TensorFlow, Keras, PyTorch, or models saved in an ONNX format (e.g. sci-kit learn) through the use of the RedisAI module [29]. Section 3.3 shows the inference performance of the Orchestrator in various settings.

There are a few benefits of the ML capability that SmartSim provides that differentiate it from other approaches. Unlike [5], users of SmartSim can switch between ML frameworks and implementations without needing to make any changes to their application code. The same SmartRedis client calls work across all supported ML frameworks, and models. Second, ML libraries like PyTorch evolve quickly and supporting direct integration of even a single ML library, as is the case in [2,3], is a daunting integration and maintenance task for mature codebases. SmartSim enforces no dependencies on the user's

application other than SmartRedis which is comparatively lightweight. Lastly, HPC applications and ML models are written in the language best suited for the task: Python for ML – C/C++/Fortran for HPC. Hence, SmartSim provides a performant way to utilize Python-based ML without re-writing [6] or wrapping [7] a user application in an "ML-friendly" language like Julia or Python.

The application presented in Section 4.4 utilized the IL to deploy a distributed Orchestrator (i.e. the 'standard deployment' alongside an `Ensemble` of SmartRedis-enabled ocean models. The Orchestrator hosted an ML model (PyTorch) which was called from Fortran to augment and improve the Eddy Kinetic Energy parameterization of each ensemble member in parallel.

2.2. SmartRedis

2.2.1. Tensors

The SmartRedis clients use an n-dimensional tensor data structure for transferring data, storing data, evaluating scripts, and evaluating models. To minimize the code changes in applications, the SmartRedis tensor data structure is opaque to the user and native n-dimensional arrays in the host language (C, C++, Fortran, Python) are used in the user-facing API functions. For example, in Python, the SmartRedis client works directly with NumPy arrays. For C and C++, both nested and contiguous memory arrays are supported. Only contiguous arrays are supported in Fortran, but with care taken to preserve the row-major convention. The reader is encouraged to consult the SmartRedis documentation for a detailed description of supported data types and API functions.

2.2.2. Datasets

In many scientific applications, multiple n-dimensional tensors are naturally grouped together as they have some contextual relationship. Additionally, there is often metadata associated with the tensors (e.g. dimension names) or the simulation from which they come (e.g. time step information) that should be stored alongside the tensors. The SmartRedis DataSet API allows users to group n-dimensional tensors and metadata into a single data structure that can be accessed or manipulated in the Orchestrator with a single key. Specifically, users need not know where the tensors and metadata within the DataSet object are stored once they have been sent to the Orchestrator. Users only need to know the name given to the dataset when constructing the DataSet object.

2.2.3. Model inference and data processing

The SmartRedis clients support the remote execution of Pytorch, TensorFlow, Keras, and ONNX models that are stored in the Orchestrator. With this capability, embedded SmartRedis clients can augment simulations with machine learning models stored in the in-memory database.

SmartRedis clients support storing, retrieving and executing ML models with the aforementioned ML frameworks. When a call to `client.set_model()` is performed, a copy of the model is distributed to every node of the database to leverage all available hardware. When performing the remote execution of a model through a SmartRedis client, the model chosen for execution is the model co-located with some or all of the model input data. In the case where all of the input data or output for a model execution is not on the same node of the database, the SmartRedis client will move temporary copies of the input or output data to the node. The movement of data between nodes for model inference is completely opaque to the user and is handled internally by the SmartRedis client.

Similar to model execution, SmartRedis provides an API for storing, retrieving, and executing TorchScript programs inside of the Orchestrator for data processing tasks. The scripts are JIT-traced Python programs that can operate on any tensor data and dataset tensors stored in the Orchestrator. The API for storing, retrieving, and executing scripts follows the same behavior and naming conventions as the API for models.

ML models and PyTorch scripts stored in the Orchestrator can be executed on CPU or GPU. The outputs of models and scripts are stored within the Orchestrator until requested by the user. In this way, scripts and models can be executed sequentially to form computational pipelines for distributed processing and inference.

3. Scaling SmartSim applications

A synthetic scaling simulation was run on a Cray XC50 to quantify Orchestrator inference performance. In this scaling analysis, a set of synthetic simulations was run with a fixed number of Orchestrator (database) nodes and varying number of SmartRedis clients and another set of synthetic simulations was run with varying number of database nodes and a fixed number of SmartRedis clients. In order to make the synthetic scaling as reproducible as possible, we use a common ML model and dataset: ResNet-50 [30] and ImageNet [31] respectively. The data is a 3D tensor of shape $224 \times 224 \times 3$.

During the synthetic simulation, each client connected to the Orchestrator repeatedly sets a tensor (image), calls a PyTorch script to process the tensor stored in the database, calls the ResNet model on the processed tensor, and retrieves the output vector. It is important to note that processing and inference both execute inside the Orchestrator on GPU, but are invoked by the SmartRedis clients remotely. Through this repeated set of SmartRedis client API calls, the ability of the SmartSim infrastructure to handle uncoordinated tensor, model, and script requests on a busy network can be assessed.

3.1. Hardware configuration

The number of database nodes in the scaling experiment was varied from 4 to 16. Each database node was equipped with an Nvidia P100 GPU, 64 GB DDR4-2400 memory, and 18-core 2.3 GHz Intel Broadwell processors. The number of clients simultaneously connected to the database varied from 960 to 7680 using computational nodes containing 48-core 2.1 GHz Intel SkyLake CPUs with 192 GB DDR4-2666 memory. The Cray XC50, used for this scaling study, utilizes the Aries interconnect.

RedisAI, the Redis module that provides the ML runtimes to the Orchestrator, contains several methods for tuning performance for given workloads such as the number of I/O threads, GPU and CPU worker threads, and background threads. In this study, 4 threads per GPU (one P100 per compute node), one I/O thread, and one background thread were unbound and free to schedule on the 18 cores of the Broadwell CPU.

3.2. Software configuration

The synthetic scaling simulation is a C++ application that utilizes the SmartRedis client API to set tensors, execute models, execute scripts, and retrieve tensors. The code snippet in Listing 1 shows the primary loop in the synthetic scaling simulation that contains the SmartRedis API calls. Note that during the scaling study, each SmartRedis API call and the outer loop were enclosed by timing calculations, but these have been removed from the excerpt to improve readability. Also, note that the code excerpt shows that a total of 100 iterations of the SmartRedis API calls is performed by each MPI rank, which is consistent with the results that will be shown in Section 3.3.

The synthetic simulation code excerpt is executed by all of the MPI ranks in the scaling study. Note that the model and script referenced in the code excerpt API calls is set by the SmartSim Python script through the SmartRedis Python client API. In this way, the model and script are set in the database before the C++ synthetic simulation is executed. The only optional parameter specified when setting the model with `client.set_model()` is a batch size of 10,000. By setting a large

```
for (int i=0; i<100; i++) {

// Create an input tensor key using MPI rank
// and iteration number
std::string in_key = "resnet_input_rank_" +
                     std::to_string(rank) + "_" +
                     std::to_string(i);

// Create an script output tensor key using
// MPI rank and iteration number
std::string script_out_key =
        "resnet_processed_input_rank_" +
        std::to_string(rank) + "_" +
        std::to_string(i);

// Create a model output tensor key using
// MPI rank and iteration number
std::string out_key = "resnet_output_rank_" +
                      std::to_string(rank) +
                      "_" + std::to_string(i);

// Put a tensor in the database
client.put_tensor(in_key, array, {224, 224, 3},
                  SmartRedis::TensorType::flt,
                  SmartRedis::MemoryLayout::nested);

// Run the script
client.run_script(script_name, "pre_process_3ch",
                  {in_key}, {script_out_key});

// Run the model
client.run_model(model_name, {script_out_key},
                 {out_key});

// Retrieve the tensor
client.unpack_tensor(out_key, result, {1,1000},
                     SmartRedis::TensorType::flt,
                     SmartRedis::MemoryLayout::nested);
}
```

Listing 1: Excerpt of synthetic scaling study main loop.

value for the batch size, the RedisAI module will group together tensors that arrive close together into a single model execution. RedisAI will attempt to batch as many batches as possible up to the provided batch size. A minimum batch size can be set to ensure a scalar value of batches are grouped before execution, however, no minimum batch size was used in this scaling study.

The synthetic scaling simulation (C++ application) is executed by a SmartSim Python driver script that sets the total number of SmartRedis clients connecting to the database and the number of databases in each execution of the C++ application. For every permutation reported here, a fresh database cluster is launched to maintain uniformity across all executions.

3.3. Results

Fig. 2 show the scaling behavior of SmartRedis API calls for select combinations of database node counts and SmartRedis client counts. Recall that in each run of the synthetic simulation there are between 960 and 7680 MPI ranks, and each MPI rank has a SmartRedis client connection to the database. Additionally, each MPI rank executes one

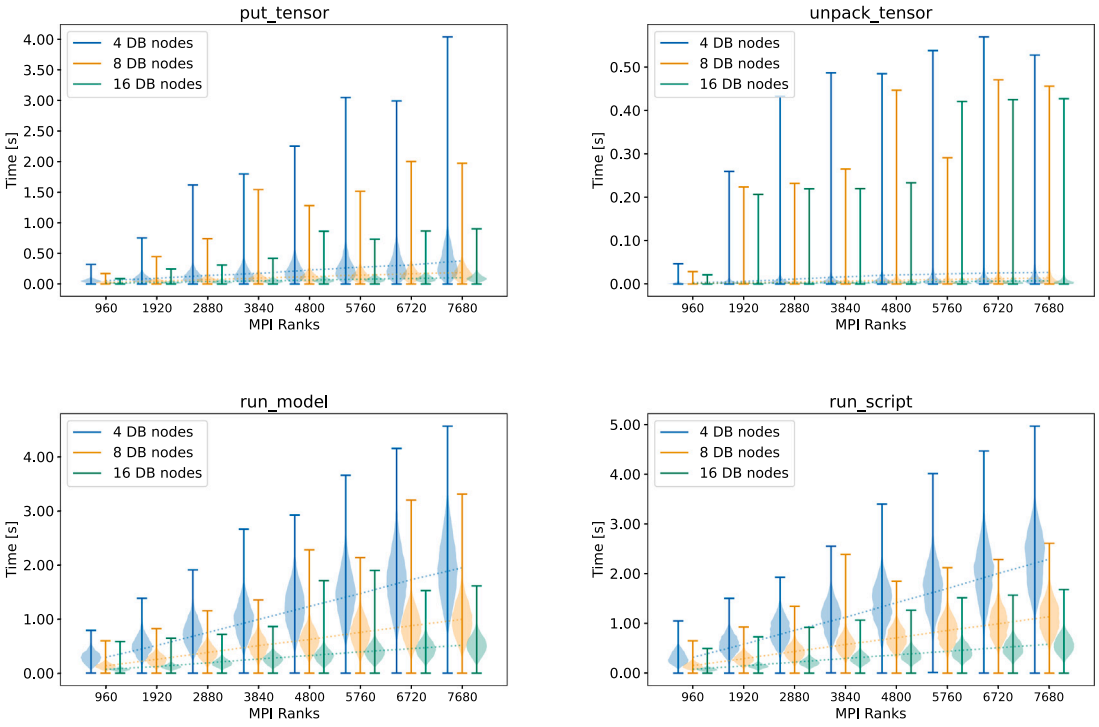

Fig. 2. Execution time of SmartRedis API calls using Redis clusters of 4, 8, and 16 DB nodes, with varying client connection count. Violin plots show timing distributions and whiskers extend from minimum to maximum recorded timings. Dashed lines connect mean values.

hundred iterations of SmartRedis API calls. As a result there is a distribution of run time across and within each MPI rank, with the aggregate behavior across all ranks and iterations shown in the same plots. No synchronization of ranks happens between iterations, however, an MPI barrier is used at the very start of the client program so that all ranks start simultaneously.

In Fig. 2, the mean run times for varying client counts with fixed database node counts are connected with dashed lines. At each data point violin plots show timing distribution, with whiskers extending from minimum to maximum recorded timings. Some variability between executions is expected and can be attributed to shared system network congestion, eager batching in the database, kernel involvement in TCP communication, and database node to application node ratio.

For the 3 most time-consuming API calls, `put_tensor`, `run_model`, and `run_script`, the mean runtime of SmartRedis API calls scales linearly as the number of SmartRedis clients is increased with a fixed database node count. Adding additional database nodes reduces both the mean and maximum times. The reduction in maximum time of a rank is especially important for applications where `run_model` would be a blocking part of the algorithm. Asynchronous versions of `run_model`, and `run_script` are promising future improvements to SmartRedis that could overlap simulations with ML computation.

The M to N ratio of database compute nodes and application compute nodes provides an interesting point of discussion. The smallest ratio (20:16) is represented by the 20 application node (960 rank) to 16 database node run in Fig. 2. The study done with 160 nodes (7680 rank) used for the client program and 4 database nodes is the largest ratio (160:4). While more research is needed to prescribe ratios for practical applications, the authors generally suggest staying below a ratio of 60:1 (application to database nodes) when using at most 48 clients per application node. However, this ratio is expected to grow as future optimizations are made to client communication strategies.

4. Application: Parameterizing oceanic turbulence using SmartSim

4.1. Motivation

The partial differential equations (PDEs) solved numerically by OGCMs are a specific application of the Navier–Stokes equations that express the laws of conservation of mass, momentum, and energy in continuum mechanics form. Parameterizations in these equations generally take the form of extra terms that are added to account for physical processes that occur at scales smaller than the model grid. Without additional constraints these terms potentially violate the fundamental conservation laws, or, at minimum, improperly represent the ways in which these quantities are added, removed, or transferred within the model domain [32].

Expressing the conservation laws in terms of their effects on the kinetic and potential energies of the flow can clarify how these parameterizations should behave. Viewed through this lens, parameterizations should account for how sub-gridscale (“eddy”) energy is exchanged with the energy of the resolved flow. Modern ocean modeling theory thus considers the eddy energy to be a lynchpin variable mediating essentially all of the parameterizations in the model [33]. However, as with all other sub-gridscale variables, a method for obtaining the eddy energy must be developed separately from the solution of the fundamental PDEs.

A PDE describing the evolution of the eddy kinetic energy can be derived from theory, but includes many terms that cannot be expressed using only quantities associated with a model’s resolved flow. The present state-of-the-art method for obtaining the eddy energy thus invokes an extra PDE that approximates its true mathematical form [34]. The modifications to the true PDE are severe; many terms are dropped and multiple others are parameterized, limiting the fidelity of the equation to truly represent eddy effects. In this study we use a data-driven approach using machine-learning to replace the PDE for eddy kinetic energy, with the goal of improving the accuracy of the simulation. This approach needs to be sufficiently performant to not impede the ocean model from being used in multi-millennial climate simulations.

4.2. Numerical model description

This study uses three global configurations of the Modular Ocean Model version 6 (MOM6), an OGCM that has been used for ocean climate simulations (e.g. [35]). The first 'eddy-resolving' (ER) configuration, uses a spatial resolution of 1/10° in both the latitudinal and longitudinal directions (about 10 km, resulting in 7.5 million ocean grid points). This resolution is sufficient to resolve mesoscale eddies between the equator and mid-latitudes (≈ 40° N/S). The other two configurations use a spatial resolution of 1/4° (about 25 km, resulting in 2.7 million ocean grid points), which falls into the so-called 'eddy-permitting' regime where the eddies are partly resolved but turbulence parameterizations are still needed. These two configurations are identical except for the eddy parameterizations that are employed: one uses the Mesoscale Eddy Kinetic Energy (MEKE) closure suggested by [34], a prognostic eddy kinetic energy equation that represents the current state-of-the-art, while the other uses "SmartSim-EKE", the trained neural network (NN) (detailed in Section 4.3) to infer the eddy kinetic energy. The estimates of EKE are then converted to a coefficient used to control the strength of the Gent–McWilliams [36] parameterization of eddy effects on the resolved circulation.

To generate the data used to train the NN and to provide a benchmark of comparison for EKE, the ER simulation is integrated for 20 years to allow the eddy field to come into equilibrium. EKE at various scales is calculated by coarsening the output by a factor of 2–10.

The features used to predict eddy kinetic energy were chosen with the requirement that each must be rotationally-invariant, meaning that it is unchanged under a uniform rotation of the model's coordinate system. This is needed to make the results of the training as agnostic as possible to different global grid geometries, which can vary substantially from model to model and for different applications. Physical principles and appealing to the phenomenology of ocean turbulence also helped to inform the final choices of the four predictors that were used in these experiments (e.g.[37]), to train the NN:

- Surface mean kinetic energy (MKE): A source/sink of eddy kinetic energy in the inverse energy cascade
- Surface relative vorticity (RV): Similar to MKE but for angular momentum
- Column-averaged isopycnal slope: A measure of the potential energy available to generate turbulence
- The Rossby radius of deformation divided by the square root of horizontal grid area: A measure of whether the length scale of the eddies can be resolved by the model grid. Values significantly greater than 1 indicate that the model can resolve turbulence.

The last two parameters are non-dimensional and were specifically chosen in an attempt to make the NN 'scale-aware' i.e. the predictions should not predict high EKE in regions where the numerical model resolution is sufficient to fully resolve the eddy field. Additionally, these fields were chosen since they have no direct correlation to geographic location to ensure that the NN was not inadvertently learning a spatial pattern.

4.3. Neural network architecture and training

To predict the EKE value, we used a small NN mainly consisting of residual blocks; these are sequences of 2D convolutional layers, in which skip-connections sum the output of one internal layer to the output of the block: they are typically found in ResNet-derived networks [38–40]. Such building blocks were chosen primarily for two reasons: 2D-convolutions are computed efficiently by the GPUs employed for training and inference, and residual blocks (and their skip-connections) show advantages during the gradient back-propagation phase of the NN training. Since the input of the NN is a single point with 4 features, the first layers are transposed convolutions (also called deconvolutions), which extend the width and height of each sample to 7, so that the convolutional layers in the residual blocks can operate on it. Different types and numbers of residual blocks were tested, and the best results (in terms of time to accuracy and robustness) were obtained with three bottleneck residual blocks [38] performing group convolutions. Attempts to use fewer residual blocks resulted in faster inference, but also in some numeric errors (where EKE could not be computed). The final part of the NN consists of two fully connected layers that output the predicted surface EKE value. Throughout the rest of this work, the NN topology (shown in Fig. 3) will be referred to as EKEResNet.

As mentioned above, each input sample has four features (predictors). The features have different statistical distributions, and an *ad hoc* pre-processing step is needed to avoid numerical problems when training the NN. MKE and isopycnal slope approximately follow a log-normal distribution, requiring a natural log transformation. The distribution of relative vorticity is symmetric, with a very narrow peak around zero, but a range of several orders of magnitude. To reduce the range without losing information around zero, the function

$$f_p(x) = \begin{cases} -\ln|x| - C & \text{if} x < 0, \\ 0 & \text{if } x = 0, \\ \ln x + C & \text{else} \end{cases} \tag{1}$$

is applied to relative vorticity. Intuitively, the effect of f_p is to apply the logarithm to both the positive and the negative domain. The result obtained on the negative domain, where the absolute value of x has to be taken, is multiplied by -1 to ensure monotonicity. This is not sufficient to make the function injective, as values around zero diverge to $\pm\infty$. As injectivity is desirable to avoid loss of information, a constant value C is added (subtracted) to the results obtained on the positive (negative) domain. C is chosen so that $C > \ln\epsilon$, where ϵ is a cutoff parameter representing the smallest non-zero value which can be encountered in the distribution. This value could be enforced by setting x to 0 when $x < \epsilon$, or it could be the machine accuracy corresponding to the floating point precision chosen for the NN training. C was set to 36 for this work.

After the pre-processing step the mean and standard deviation of each feature is stored and then every feature is standardized. Notice that the Rossby radius of deformation is not pre-processed otherwise, as standardizing it is sufficient to avoid numerical problems or loss of accuracy.

As shown in Fig. 4, the EKE values appear to approximately follow a log-normal distribution, thus the network is trained against ln(EKE).

EKEResNet was trained with Stochastic Gradient Descent, with a learning rate of 4×10^{-3} and a step-wise schedule with peak value reached at the fifth epoch, out of a total of 100. The mean square error was employed as a loss function, with $L2$ penalty of 2×10^{-4}. The training was performed in parallel on 8 GPUs (Nvidia P100 or V100), with a local batch size of 512 samples. Parallelism was achieved using HPE DL-plugin [41].

With the described setting, the NN shows a strong tendency towards overfitting values close to the mean of the distribution. This could be due to the abundance of samples belonging to the ln(EKE) distribution mean, and the scarcity of samples belonging to the distribution tails. This is a problem not only because the resulting distribution is different from the target one, but also because the most important EKE values for the simulation are those close to the highest values of the distribution, and a model trained with this approach tends to miss them.

To mitigate this problem, a weighted sampling scheme was used: to each data-point in the training set a weight corresponding to the inverse of its distribution probability density is assigned, this weight is proportional to the probability of drawing each sample during a training epoch. For each epoch, only one-tenth of the data set was used. With this approach, the probability of drawing samples of each portion of the ln(EKE) domain becomes more uniform, and the distribution of the predicted ln(EKE) values has a broader peak and extends more to the tails of the distribution, as shown in Fig. 5.

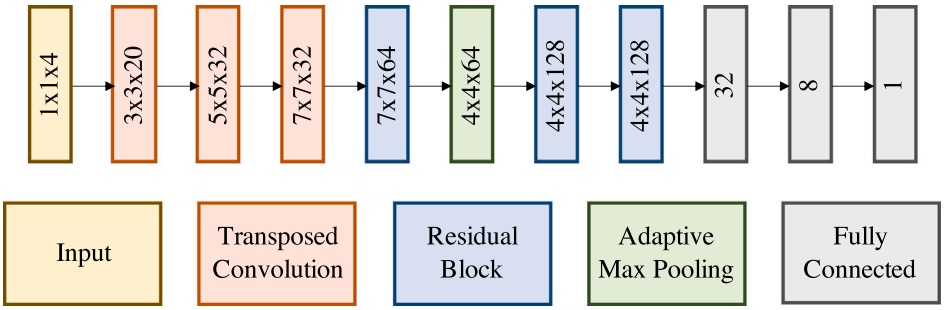

Fig. 3. Simplified view of EKEResNet topology. For each constitutive block, the dimension of the output tensor is shown as WxHxC. Residual blocks are implemented as bottleneck blocks.

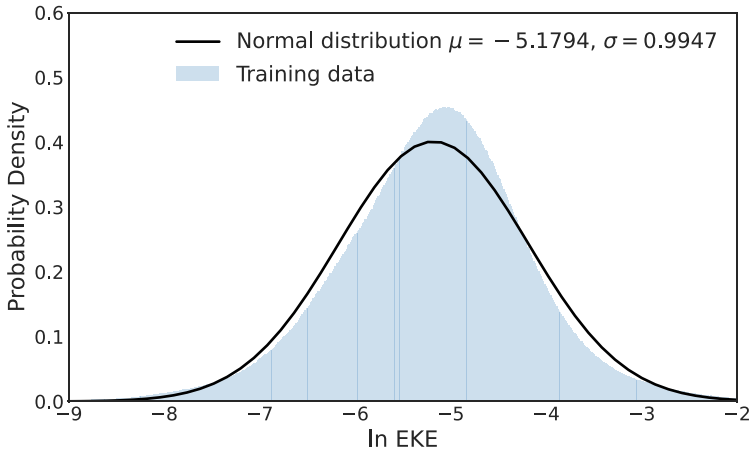

Fig. 4. Distributions of ln(EKE) on the whole training dataset and fitted normal distribution.

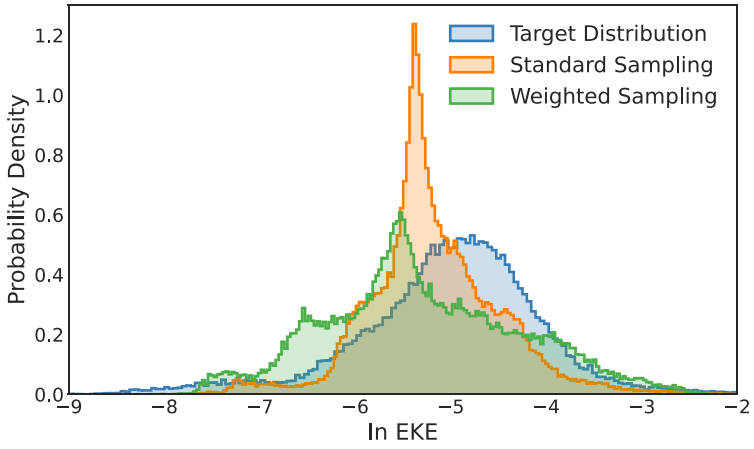

Fig. 5. Distributions of ln(EKE) values as predicted by the NN trained with uniform and weighted sampling, compared to ground truth for one test sample.

While it is true that the weighted sampling gives a qualitatively better distribution of ln(EKE), it also has a negative effect on the achieved accuracy: the minimum mean square error attained during the training was 0.55 using uniform sampling and 0.60 with weighted sampling.

4.4. Results

An ensemble of 12 SmartSim-EKE simulations were run to demonstrate the scalability and performance of SmartSim, evaluate the accuracy of the ML-EKE parameterization, and estimate its computational cost at scale. These types of ensembles are used to characterize uncertainty in weather predictions and climate projections. Each member was integrated for 10 simulated years using 910 physical cores (10,920 cores total, using a combination of Intel Skylake and Cascadelake processors) with 16 Nvidia P100 GPUs dedicated for the shared Orchestrator. The ensemble members were branched from a previous 20-year spin-up simulation that used the MEKE parameterization. A spatially-random 0.0001 °C perturbation was added to the 3D temperature field of each member to differentiate it from the others in the ensemble. (Note that we were limited to 10 simulated years due to the computational resources that could be dedicated during this experiment. Ongoing work is being done to extend the simulation to centennial-scale simulations.)

The online inference portion of MOM6 (i.e. the SmartRedis-based calls to EKEResNet) is called 8 times per simulated day (about every

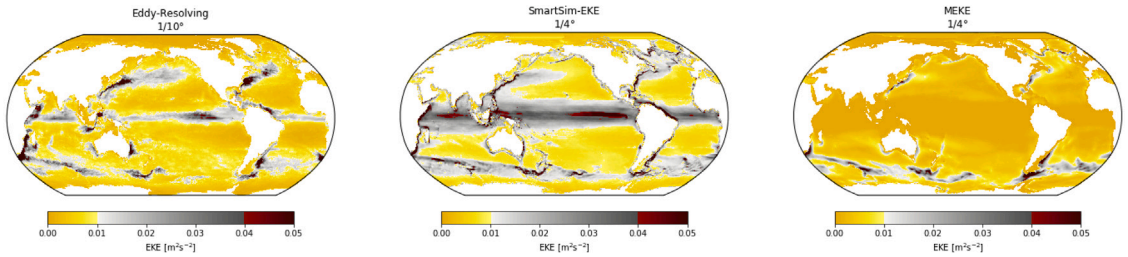

Fig. 6. Eddy kinetic energy (EKE), averaged over the last year of each simulation, calculated from the eddy-resolving (ER) 1/10° simulation (a), inferred online using EKEResNet (referred to as SmartSim-EKE), averaged over all 12 ensemble members (b), and the current state-of-the-art MEKE parameterization (c). Both SmartSim-EKE and MEKE use a 1/4° grid which is slightly coarser than the factor of 2 coarsening shown in (a); the ER EKE thus represents a lower-bound on what the 'true' EKE should be at that resolution.

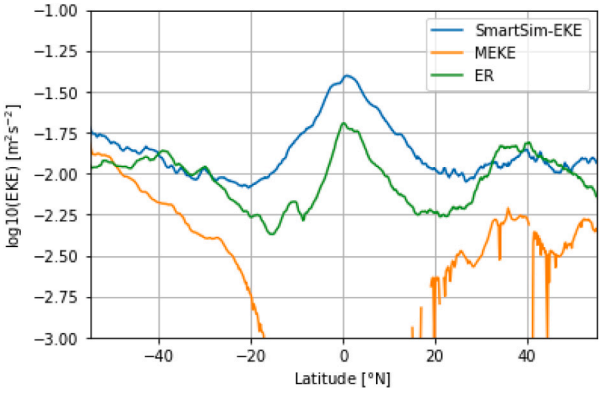

Fig. 7. Zonally averaged EKE (on a log10 scale) as a function of latitude from the ER, SmartSim-EKE, and MEKE. Note that within 20 degrees of the equator MEKE computes a near-zero value for EKE.

4 s of walltime) on 2.7 million grid points spread across 910 cores. Because MOM6 must wait for every online inference loop to complete, the most accurate way to compare the overall expense of the SmartSim-EKE-based approach is to use the timings of the slowest subdomain. Examining the timings for ensemble member 1 (which is representative of the ensemble as a whole), the total elapsed time in `put_tensor` was 24s whereas `run_model` was about 2 h. Based on a total wall time of 136 h, this means that SmartSim-EKE incurred an additional 1.5% overhead to the computational cost of the model, which we consider to be an acceptable tradeoff to gain an enhanced representation of EKE. We note that this 1.5% overhead was also seen when running an ensemble member individually, suggesting that the Orchestrator was not being overtaxed. Integrated over the entirety of the ensemble simulation, approximately 970 billion inferences were performed resulting in an average rate of 1.86 million online inferences per second.

SmartSim-EKE and MEKE are compared to the "true" EKE calculated directly from ER (Figs. 6 and 7). The ER simulation correctly shows elevated EKE in the regions where vigorous eddy activity is expected: western boundary currents (e.g. the Gulf Stream and the Kuroshio currents), the Southern Ocean, and the eastern equatorial Pacific. In comparison, SmartSim-EKE generally overestimates the extent of the equatorial EKE, but otherwise reasonably captures the magnitude and large-scale patterns seen in ER (Fig. 7). In the Equatorial Pacific, the eddy field is driven by a unique set of dynamics due to the near-absence of the Coriolis force, and is not well-represented by the features that we used to predict the occurrence of baroclinic turbulence. The resulting errors in the prediction do not end up affecting the simulation because a separate scaling function ensures eddy parameterizations are not applied in the near-equatorial regions where the eddies are well-resolved.

SmartSim-EKE is an improvement over MEKE particularly in the subtropics. In the extensions of the western boundary currents, SmartSim-EKE shows high values of EKE whereas MEKE's values are too low. In the rest of the gyres SmartSim tends to have EKE patterns more similar to the ER simulation as compared to MEKE which has large areas of the ocean with near-zero values of EKE. Overall, the root-mean-square error of EKE between the ER and SmartSim-EKE cases is 0.012 m^2 s^{-2} a reduction of about 20% compared to the MEKE simulation (0.015 $m^2 s^{-2}$). Excluding the tropics (which as mentioned above is a region where eddies are resolved by the simulation), yields an even more dramatic improvement as compared to MEKE (0.008 vs 0.012 $m^2 s^{-2}$), a 39% improvement. This suggests that some of the omitted or parameterized terms in MEKE's prognostic EKE equation result in significant structural biases.

The 10-year integrations shown here are not sufficient to evaluate whether the improved representations of EKE result in a more skillful representation of the ocean for weather prediction or climate. 10 years however is sufficient to demonstrate that the inclusion of machine-learning still leads to stable, realistic simulations of the large-scale ocean circulation. In the Atlantic basin, the Gulf Stream separates the clockwise flow of the subtropical gyre from the subpolar North Atlantic gyre. These two gyres can be seen clearly in the ensemble average of sea surface height (SSH) (Fig. 8a) as positive (red) and negative (blue) colors, respectively. The standard deviation of SSH (Fig. 8) is also elevated (blue) in the expected locations where strong eddy activity modulates the path of the North Atlantic Current. The SSH anomalies in two of the ensemble members (Figs. 8c,d) show trains of positive and negative anomalies indicating the presence of eddies. This suggests that the SmartSim-EKE parameterization is not so strong that it is suppressing resolved eddies, a well-known pitfall in eddy-permitting simulations [42]. We note that all 12 ensemble members completed their 10-year simulations with no evidence of numerical instability.

Based on these results, the approach used for SmartSim-EKE demonstrates a effective means of using machine-learning to improve the accuracy of ocean simulations without incurring a significant computational cost. Additionally, by offloading a GPU-intensive computation onto database nodes we are able to efficiently use CPU-only and CPU/GPU nodes on a heterogeneous cluster — an especially important consideration for the types of HPC platforms becoming commonplace at climate modeling centers. The overall methodology can be readily extended to atmospheric, land, and ice models, opening the possibility for machine-learning augmented, fully-coupled climate models.

5. Discussion

This study introduces a new software solution, SmartSim, that is composed of an Infrastructure Library and a Client Library that can couple existing High Performance Computing (HPC) simulations written in Fortran/C/C++ to Machine Learning (ML) and data analysis libraries online and at scale. We demonstrated that, with minimal code changes, SmartSim clients can leverage the SmartRedis API to support the remote execution of TensorFlow, Keras, ONNX, and PyTorch models and scripts for distributed, online inference.

In addition to the describing the SmartSim architecture, we demonstrated the particular use case of integrating SmartSim with Modular

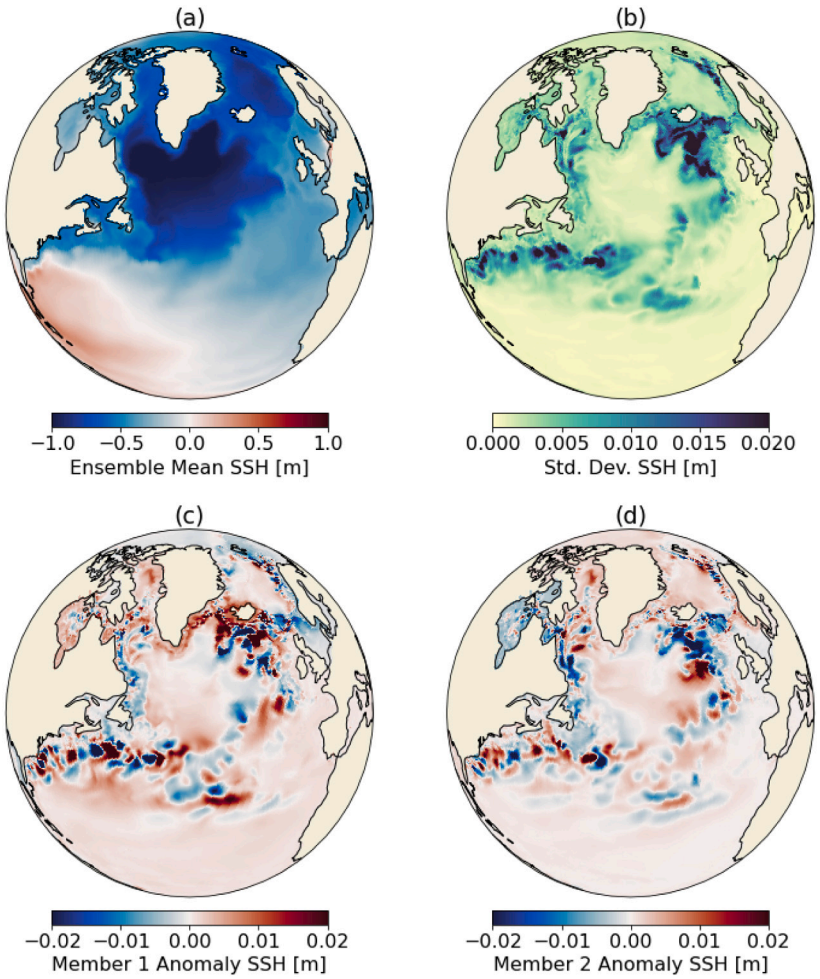

Fig. 8. Mean (a) and variability (b) of sea surface height (roughly the streamlines of the large-scale flow) across the SmartSim-EKE ensemble as diagnosed by sea surface height on the last day of the simulation. Panels (c) and (d) show the difference in SSH between ensemble members 1 (c) and 2 (d) and the ensemble mean.

Ocean Model 6 (MOM6), which is written in Fortran. We replaced an existing parameterization for eddy kinetic energy with a data driven, ML model. The JIT-traced, PyTorch model was queried at runtime (online) for the prediction of eddy kinetic energy by the Fortran SmartRedis Client. Additionally, we showed that SmartSim is capable of running ensembles of global ocean simulations utilizing the same ML infrastructure, with minimal impact on ensemble member (simulation) runtime. As stated, we believe the results shown here are the first demonstrations of using ML within freely running, realistic, global simulations of the ocean.

SmartSim facilitates the convergence of AI and numerical simulation workloads by removing the necessity for file I/O, providing clients to seamlessly connect applications across programming languages, and utilizing a ML library agnostic API between workloads. As mentioned, previous efforts to utilize ML in simulation models have not addressed impediments to the actual utilization of ML. Approaches that re-create ML libraries in simulation languages, embed large ML libraries or python interpreters, or use the filesystem as an intermediary lack the flexibility to support and benefit from the rapid advancements in the data science ecosystem.

Due to the massive size of compute resources, HPC applications have historically maintained highly controlled strategies for communication and data access. The result has been optimized, yet constrained, data flow driven by MPI communication barriers. Data extraction has almost entirely relied on parallel, networked filesystems which introduce meaningful delays in simulation model development, analysis and enhancement. Users have not, without major architectural changes, been able to easily couple applications across languages, runtimes, and heterogeneous processor types (CPU/GPU). The loosely coupled nature of the data communication in SmartSim enables new paradigms in application coupling, computational steering, real-time analysis, and the utilization of ML at scale.

The authors recognize that loosening the constraints on data communication in HPC applications may lead to degradation in application performance at extreme scales. We believe the added flexibility of communication is of greater importance given the types of applications and systems that are enabled through this approach. In addition, we show SmartSim scales linearly to thousands of processors in multiple configurations on modern, heterogeneous HPC systems. Our scaling study, application, and SmartSim codebase are all available with instructions for reproduction of this work.

In future work, the authors plan to investigate the utilization of SmartSim for continuous online training of ML models as well as support new features such as flash storage and asynchronous data communication. In conjunction, integrations of SmartRedis data-structures into popular open source data formats (ex. Xarray, VTK) and libraries (ex. Ray, Dask) will be explored.

CRediT authorship contribution statement

Sam Partee: Conceptualization, Methodology, Software, Investigation, Data curation, Writing – original draft, Writing – review & editing, Visualization, Supervision, Project administration. **Matthew Ellis:** Conceptualization, Methodology, Software, Investigation, Data

curation, Writing – original draft, Visualization. **Alessandro Rigazzi:** Conceptualization, Methodology, Software, Investigation, Data curation, Writing – original draft, Visualization. **Andrew E. Shao:** Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Writing – original draft, Writing – review & editing, Visualization. **Scott Bachman:** Conceptualization, Methodology, Validation, Formal analysis, Investigation, Writing – original draft. **Gustavo Marques:** Conceptualization, Validation, Formal analysis, Investigation, Writing – original draft. **Benjamin Robbins:** Resources, Writing – original draft, Supervision, Project administration.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The authors would like to acknowledge Elliot Ronaghan for his help with application performance, and RedisLabs for their continued support. AES acknowledges support from Mitacs as an Accelerate Research Fellow under IT15738.

References

[1] H.G. Weller, G. Tabor, H. Jasak, C. Fureby, A tensorial approach to computational continuum mechanics using object-oriented techniques, Comput. Phys. 12 (6) (1998) 620–631, http://dx.doi.org/10.1063/1.168744, arXiv:https://aip.scitation.org/doi/pdf/10.1063/1.168744, URL https://aip.scitation.org/doi/abs/10.1063/1.168744.

[2] N. Geneva, N. Zabaras, Quantifying model form uncertainty in Reynolds-averaged turbulence models with Bayesian deep neural networks, J. Comput. Phys. 383 (2019) 125–147, http://dx.doi.org/10.1016/j.jcp.2019.01.021, URL https://linkinghub.elsevier.com/retrieve/pii/S0021999119300464.

[3] R. Maulik, H. Sharma, S. Patel, B. Lusch, E. Jennings, A turbulent eddy-viscosity surrogate modeling framework for Reynolds-averaged Navier-Stokes simulations, 2020, [Physics] URL http://arxiv.org/abs/1910.10878, arXiv:1910.10878.

[4] P.A. O'Gorman, J.G. Dwyer, Using machine learning to parameterize moist convection: potential for modeling of climate,climate change,and extreme events, J. Adv. Modelling Earth Syst. 10 (10) (2018) 2548–2563, http://dx.doi.org/10.1029/2018MS001351, URL https://onlinelibrary.wiley.com/doi/abs/10.1029/2018MS001351.

[5] J. Ott, M. Pritchard, N. Best, E. Linstead, M. Curcic, P. Baldi, A fortran-keras deep learning bridge for scientific computing, 2020, [Cs] URL http://arxiv.org/abs/2004.10652, arXiv:2004.10652.

[6] T. Schneider, S. Lan, A. Stuart, J. Teixeira, Earth system modeling 2.0: a blueprint for models that learn from observations and targeted high-resolution simulations, Geophys. Res. Lett. 44 (24) (2017) 12,396–12,417, http://dx.doi.org/10.1002/2017GL076101, URL https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1002/2017GL076101 arXiv:https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1002/2017GL076101.

[7] J. McGibbon, N.D. Brenowitz, M. Cheeseman, S.K. Clark, J. Dahm, E. Davis, O.D. Elbert, R.C. George, L.M. Harris, B. Henn, A. Kwa, W.A. Perkins, O. Watt-Meyer, T. Wicky, C.S. Bretherton, O. Fuhrer, Fv3gfs-Wrapper: a Python Wrapper of the FV3GFS Atmospheric Model, Climate and Earth System Modeling, 2021, http://dx.doi.org/10.5194/gmd-2021-22, URL https://gmd.copernicus.org/preprints/gmd-2021-22/.

[8] Y. Feng, N. Hand, Launching python applications on peta-scale massively parallel systems, in: S. Benthall, S. Rostrup (Eds.), Proceedings of the 15th Python in Science Conference, 2016, pp. 137–143, http://dx.doi.org/10.25080/Majora-629e541a-013.

[9] IPCC, Summary for policymakers, in: T. Stocker, D. Qin, G.-K. Plattner, M. Tignor, S. Allen, J. Boschung, A. Nauels, Y. Xia, V. Bex, P. Midgley (Eds.), Climate Change 2013: The Physical Science Basis. Contribution of Working Group I to the Fifth Assessment Report of the Intergovernmental Panel on Climate Change, Cambridge University Press, Cambridge, United Kingdom and New York, NY, USA, 2013, pp. 1–30, http://dx.doi.org/10.1017/CBO9781107415324.004, URL www.climatechange2013.org.

[10] S. Manabe, R.T. Wetherald, Thermal equilibrium of the atmosphere with a given distribution of relative humidity, J. Atmos. Sci. 24 (3) (1967) 241–259, http://dx.doi.org/10.1175/1520-0469(1967)024<0241:TEOTAW>2.0.CO;2.

[11] K.E. Trenberth, J.T. Fasullo, M.A. Balmaseda, Earth's energy imbalance, J. Clim. 27 (9) (2014) 3129–3144.

[12] T. DeVries, M. Holzer, F. Primeau, Recent increase in oceanic carbon uptake driven by weaker upper-ocean overturning, Nature 542 (7640) (2017) 215–218.

[13] N. Gruber, D. Clement, B.R. Carter, R.A. Feely, S. Van Heuven, M. Hoppema, M. Ishii, R.M. Key, A. Kozyr, S.K. Lauvset, et al., The oceanic sink for anthropogenic CO2 from 1994 to 2007, Science 363 (6432) (2019) 1193–1199.

[14] V. Eyring, S. Bony, G.A. Meehl, C.A. Senior, B. Stevens, R.J. Stouffer, K.E. Taylor, Overview of the coupled model intercomparison project phase 6 (CMIP6) experimental design and organization, Geosci. Model Dev. 9 (5) (2016) 1937–1958, http://dx.doi.org/10.5194/gmd-9-1937-2016, URL https://www.geosci-model-dev.net/9/1937/2016/gmd-9-1937-2016.html.

[15] S.R. Jayne, J. Marotzke, The oceanic eddy heat transport, J. Phys. Oceanogr. 32 (12) (2002) 3328–3345.

[16] R. Ferrari, C. Wunsch, Ocean circulation kinetic energy: Reservoirs, sources, and sinks, Annu. Rev. Fluid Mech. 41 (2009).

[17] D.B. Chelton, M.G. Schlax, R.M. Samelson, Global observations of nonlinear mesoscale eddies, Prog. Oceanogr. 91 (2) (2011) 167–216.

[18] B.P. Kirtman, C. Bitz, F. Bryan, W. Collins, J. Dennis, N. Hearn, J.L. Kinter, R. Loft, C. Rousset, L. Siqueira, C. Stan, R. Tomas, M. Vertenstein, Impact of ocean model resolution on CCSM climate simulations, Clim. Dynam. 39 (6) (2012) 1303–1328, http://dx.doi.org/10.1007/s00382-012-1500-3.

[19] B.A. Boville, P.R. Gent, The NCAR climate system model, version one, J. Clim. 11 (6) (1998) 1115–1130.

[20] M. Sonnewald, R. Lguensat, D.C. Jones, P.D. Dueben, J. Brajard, V. Balaji, Bridging observations, theory and numerical simulation of the ocean using machine learning, Environ. Res. Lett. (2021) 28.

[21] V.M. Krasnopolsky, M.S. Fox-Rabinovitz, A.A. Belochitski, Using ensemble of neural networks to learn stochastic convection parameterizations for climate and numerical weather prediction models from data simulated by a cloud resolving model, Adv. Artif. Neural Syst. 2013 (2013) 1–13, http://dx.doi.org/10.1155/2013/485913, URL https://www.hindawi.com/journals/aans/2013/485913/.

[22] N.D. Brenowitz, C.S. Bretherton, Prognostic validation of a neural network unified physics parameterization, Geophys. Res. Lett. 45 (12) (2018) 6289–6298, http://dx.doi.org/10.1029/2018GL078510, URL http://doi.wiley.com/10.1029/2018GL078510.

[23] Y. Han, G.J. Zhang, X. Huang, Y. Wang, A moist physics parameterization based on deep learning, J. Adv. Modelling Earth Syst. 12 (9) (2020) http://dx.doi.org/10.1029/2020MS002076, URL https://onlinelibrary.wiley.com/doi/10.1029/2020MS002076.

[24] T. Beucler, S. Rasp, M. Pritchard, P. Gentine, Achieving conservation of energy in neural network emulators for climate modeling, 2019, [Physics] URL http://arxiv.org/abs/1906.06622, arXiv:1906.06622.

[25] S. Rasp, M.S. Pritchard, P. Gentine, Deep learning to represent subgrid processes in climate models, Proc. Natl. Acad. Sci. 115 (39) (2018) 9684–9689, http://dx.doi.org/10.1073/pnas.1810286115, URL http://www.pnas.org/lookup/doi/10.1073/pnas.1810286115.

[26] T. Bolton, L. Zanna, Applications of deep learning to ocean data inference and subgrid parameterization, J. Adv. Modelling Earth Syst. 11 (1) (2019) 376–399, http://dx.doi.org/10.1029/2018MS001472, URL https://onlinelibrary.wiley.com/doi/abs/10.1029/2018MS001472, _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1029/2018MS001472.

[27] A.P. Guillaumin, L. Zanna, Stochastic-deep learning parameterization of ocean momentum forcing, J. Adv. Modelling Earth Syst. 13 (9) (2021) http://dx.doi.org/10.1029/2021MS002534, e2021MS002534 URL https://onlinelibrary.wiley.com/doi/abs/10.1029/2021MS002534, _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1029/2021MS002534.

[28] URL https://github.com/antirez/redis.

[29] URL https://oss.redis.com/redisai/.

[30] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: 2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR, IEEE, Las Vegas, NV, USA, 2016, pp. 770–778, http://dx.doi.org/10.1109/CVPR.2016.90, URL http://ieeexplore.ieee.org/document/7780459/.

[31] Y. You, Z. Zhang, C.-J. Hsieh, J. Demmel, K. Keutzer, ImageNet training in minutes, 2017, arXiv:1709.05011.

[32] C. Eden, A. Iske, Energy Transfers in Atmosphere and Ocean, 2019, OCLC: 1091374853.

[33] L. Zanna, S. Bachman, M.F. Jansen, Energizing turbulence closures in ocean models, in: CLIVAR Exchanges/US CLIVAR Variations. Vol. 18, (1) 2020, pp. 3–8, http://dx.doi.org/10.5065/g8w0-fy32.

[34] M.F. Jansen, I.M. Held, A. Adcroft, R. Hallberg, Energy budget-based backscatter in an eddy permitting primitive equation model, Ocean Model. 94 (2015) 15–26, http://dx.doi.org/10.1016/j.ocemod.2015.07.015, URL https://www.sciencedirect.com/science/article/pii/S1463500315001341.

[35] A. Adcroft, W. Anderson, V. Balaji, C. Blanton, M. Bushuk, C.O. Dufour, J.P. Dunne, S.M. Griffies, R. Hallberg, M.J. Harrison, I.M. Held, M.F. Jansen, J.G. John, J.P. Krasting, A.R. Langenhorst, S. Legg, Z. Liang, C. McHugh, A. Radhakrishnan, B.G. Reichl, T. Rosati, B.L. Samuels, A. Shao, R. Stouffer, M. Winton, A.T. Wittenberg, B. Xiang, N. Zadeh, R. Zhang, The GFDL global ocean and sea ice model OM4.0: Model description and simulation features, J. Adv. Modelling Earth Syst. 11 (10) (2019) 3167–3211, http://dx.doi.org/10.1029/2019MS001726, URL https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/2019MS001726.

[36] P.R. Gent, J.C. Mcwilliams, Isopycnal mixing in ocean circulation models, J. Phys. Oceanogr. 20 (1) (1990) 150–155, http://dx.doi.org/10.1175/1520-0485(1990)020<0150:IMIOCM>2.0.CO;2.
[37] N.A. Phillips, Energy transformations and meridional circulations associated with simple baroclinic waves in a two-level, quasi-geostrophic Model1, Tellus 6 (3) (1954) 273–286, http://dx.doi.org/10.1111/j.2153-3490.1954.tb01123.x, URL https://onlinelibrary.wiley.com/doi/abs/10.1111/j.2153-3490.1954.tb01123.x, _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.2153-3490.1954.tb01123.x.
[38] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: 2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR, 2016, pp. 770–778, http://dx.doi.org/10.1109/CVPR.2016.90.
[39] S. Xie, R.B. Girshick, P. Dollár, Z. Tu, K. He, Aggregated residual transformations for deep neural networks, 2016, CoRR, arXiv:1611.05431.
[40] S. Zagoruyko, N. Komodakis, Wide residual networks, 2016, CoRR arXiv:1605.07146.
[41] A. Mathuriya, D. Bard, P. Mendygral, L. Meadows, J. Arnemann, L. Shao, S. He, T. Karna, D. Moise, S.J. Pennycook, K. Maschoff, J. Sewall, N. Kumar, S. Ho, M. Ringenburg, Prabhat, V. Lee, Cosmoflow: Using deep learning to learn the universe at scale, 2018, arXiv:1808.04728.
[42] R. Hallberg, Using a resolution function to regulate parameterizations of oceanic mesoscale eddy effects, Ocean Model. 72 (2013) 92–103.

Sam Partee is a Machine Learning Engineer at Hewlett Packard Enterprise. He specializes in the intersection of High Performance Computing (HPC) and Machine Learning (ML). Before HPE, Sam worked for Cray on ML research and the Chapel parallel programming language. Sam holds a Bachelor of Science in Computer Science from Haverford College.