# Colmena: Scalable Machine-Learning-Based Steering of Ensemble Simulations for High Performance Computing

Logan Ward,*‡ Ganesh Sivaraman,* J. Gregory Pauloski,† Yadu Babuji,† Ryan Chard,*
Naveen Dandu,‡ Paul C. Redfern,‡ Rajeev S. Assary,‡ Kyle Chard,† Larry A. Curtiss,‡
Rajeev Thakur,* and Ian Foster*†
*Data Science and Learning Division, Argonne National Laboratory, Lemont, IL, USA
†Department of Computer Science, University of Chicago, Chicago, IL, USA
‡Joint Center for Energy Storage Research, University of Chicago, Chicago, IL, USA

***Abstract*—Scientific applications that involve simulation ensembles can be accelerated greatly by using experiment design methods to select the best simulations to perform. Methods that use machine learning (ML) to create proxy models of simulations show particular promise for guiding ensembles but are challenging to deploy because of the need to coordinate dynamic mixes of simulation and learning tasks. We present Colmena, an open-source Python framework that allows users to steer campaigns by providing just the implementations of individual tasks plus the logic used to choose which tasks to execute when. Colmena handles task dispatch, results collation, ML model invocation, and ML model (re)training, using Parsl to execute tasks on HPC systems. We describe the design of Colmena and illustrate its capabilities by applying it to electrolyte design, where it both scales to 65 536 CPUs and accelerates the discovery rate for high-performance molecules by a factor of 100 over unguided searches.**

***Index Terms*—Machine learning, Computational Steering, Many Task Computing**

## I. Introduction

High performance computing (HPC) campaigns involving repeated runs of simulations codes are being applied widely in science and engineering, for example to simulate molecules or proteins in different configurations to build models of how properties change with temperatures (model fitting) [1] or to evaluate many engine designs to find those with optimal efficiency (optimization) [2]. While experiments themselves are independent and can be run in parallel, fixed computing budgets and ever larger search spaces make it increasingly important to use results from completed experiments to steer such campaigns, i.e., to inform decisions about which experiments to perform next. Machine learning (ML) is an emerging tool for writing steering applications that can learn from new data faster than can human experts.

The core of algorithms for steering ensemble simulations, in broad terms, is a process that selects inputs to simulations and determines allocations of resources to tasks. Steering policies vary significantly in complexity. For example, genetic algorithms select new experiments by combining characteristics of previous simulations, typically dedicating resources equally among experiments [3], while Optimal Experimental Design (OED) methods determine new experiments based on the predictions and uncertainty of an internal statistical model, with resources allocated to tasks using methods such as all-to-one task, batches of experiments [4], and streaming [5]. Some methods deploy only a single type of task, and others can select among different task types or levels of accuracy [6].

Steering algorithms that employ ML are particularly problematic to deploy on HPC due to the need to manage the execution of not only simulations but also steering (e.g., model update, experimental design) tasks. Achieving good results requires balancing two often conflicting needs: performing steering tasks often enough, and fast enough, to enable good choices of simulation tasks, and maintaining high HPC utilization. At a simple level, both challenges can be solved by reallocating resources—a common capability of many workflow systems. On a deeper level, the challenges involve more subtle questions, such as how many resources to allocate to steering and when to retrain ML model(s). For example, one may want to enforce a limited budget for steering tasks or decide when to retrain models by comparing simulation outputs to ML predictions. The problem of expressing such policies and deploying them to HPC requires much innovation.

Existing methods for steering HPC simulations are purpose-built tools that combine a specific class of steering algorithm with methods for deploying computations across resources. An early example, Nimrod/O [7], used built-in optimization algorithms to choose which simulations to schedule over distributed clusters. More recently, the CANDLE Supervisor library [3] uses the Swift/T workflow engine [8] to distribute tasks selected by an optimization algorithm that determines new tasks after results are placed in an output queue [3]. Rocketsled works by adding a special "steering" task at the end of a Fireworks workflow definition that submits new work after completion (e.g., to launch a Bayesian optimization task) [9]. Variations of these patterns exist in other steering tools, including LibEnsemble [10], CAMD [11], DeepHyper [12], SuperLearner [2], and DeepDriveMD [13]. Each steering system uses a different way to express planning policies and are integrated differently with workflow engine, all of which

provide informative initial steps towards learning how to best steer ensembles with AI. Further advancement in ensemble simulations will require systems that provide greater flexibility in policies for orchestrating machine learning and simulation tasks together.

We present Colmena, a general-purpose library for steering ensembles of experiments on HPC computing systems. Colmena is an open-source Python code that permits writing complex agents for steering ensembles of simulations and executing them across diverse computational resources, with a particular focus on applications that use ML to design computational campaigns. In this paper, we formalize the experiment steering process and describe the design principles for the Colmena library. We then demonstrate the ability to use Colmena to steer simulations on 1024 nodes (65 536 cores) and illustrate its use on a molecular design challenge. We intend that Colmena will facilitate experimentation with advanced algorithms for steering experiments across diverse computing resources and, eventually, automated laboratories.

The main contributions of this paper are:

- An abstract formulation of the computational campaign steering problem.
- The introduction of Colmena, a Python library for steering computational campaigns on HPC.
- A demonstration of using Colmena to design molecular materials using quantum chemistry and ML.
- An analysis of the scaling performance of Colmena on a Cray XC40.

## II. The Problem

We first provide an abstract definition of the problem that we seek to solve and then describe the example application used in our experiments.

### A. Abstract Formulation

Let us assume that we have a (typically large) set of **entities**, $e \in \mathcal{E}$, each with **properties** $p \in \mathcal{P}$; a set of **assays** (e.g., simulations or laboratory experiments) $a \in \mathcal{A}$, each of which can be used to estimate a property $P(a) \in \mathcal{P}$ of an entity; and a **scoring function** $\mathcal{S}$, that when applied to available data on an entity's properties returns either a numeric score or $\phi$ if data are inadequate to assign a score. Note that multiple assays may exist for the same property, each with different cost and accuracy characteristics.

Given $\mathcal{E}$, $\mathcal{P}$, and $\mathcal{A}$, we can determine entity properties by performing a series of **tasks**, each involving the application of an assay $a \in \mathcal{A}$ to an entity $e \in \mathcal{E}$ to obtain an estimated value $v$ for property $p = P(a)$. Given a record $D$ of such tests, with each $d \in D$ defined by a tuple $(e, a, p, v)$, we can assign a score to $D$. For example, we might define the score as that of the single highest-scoring entity:

$$V(D) = \max_{x \in \mathcal{E}} S(\{d : d \in D \text{ and } d.e = x\})$$

We can also determine the cost incurred to produce $D$ by summing the costs incurred to obtain each value

$$C(D) = \sum_{d \in D} c(d)$$

**Experiment design problem.** When the number of entities, $|\mathcal{E}|$, is large and/or assays are expensive, it becomes impractical to evaluate every possible entity-property combination. The quality of the answers obtained then depends on which assays have been performed. Thus we have an experiment design problem. If $\mathcal{D}$ is every possible combination of tests, and $B$ is a resource bound, then we want to identify a set of tests $D$ such that:

$$\max_{D \in \mathcal{D}} V(D) : C(D) \leq B$$

The order in which tests are performed then matters. For example, do we focus on lower-cost assays that may identify promising entities, or on higher-cost assays that may confirm (or eliminate) promising entities?

**Static vs. learned assays.** We distinguish between *static assays*, which have fixed behavior over the course of an experiment (e.g., a simulation code) and *learned assays*, which can be improved as more data are added to the record. An example of the latter is an ML model that approximates results obtained from an expensive simulation.

**Training.** This additional kind of task is used to generate a new version of a learned assay given the current record: $a' = \texttt{retrain}(a, D)$. As ML model accuracy generally increases with training data quantity and diversity, we have another dimension to our experiment design problem—whether to perform assays designed to increase training data diversity or to characterize promising entities.

**Generating candidates**. If the number of entities is large or even innumerable (e.g., all possible polymers), we may introduce a generator $G$ that when called produces one or more new candidate entities based on the record.

**Decision problem**. If actions are taken one at a time, then system state at each step is captured by the sets of known entities $E$, associated data $D$, and assays (including learned assays) $A$. Initially, each may be empty or alternatively may be prepopulated to provide some initial knowledge of the problem. At each step, the next action is one of:

- generate one or more new entities, $e = G(D)$;
- run a task $a(e)$ for some $a$ and $e$; or
- (re)train a learned assay $a$ to generate a new $a'$.

### B. Our Example Application

As an illustrative example, we apply Colmena to a problem in electrolyte design for next-generation batteries. In this application, entities are molecules; properties of interest include atomization energy, ionization potential, toxicity, stability, and synthesizability; assays include a variety of computational methods with varying costs and accuracies; and a scoring function might impose toxicity and stability thresholds and then sum the other properties.

More specifically, we present results for a version of this problem that involves a fixed search space of molecules and a single property to optimize: specifically, $10^5$ molecules (represented as SMILES strings) from the QM9 dataset [14], [15]; a single property, ionization potential; and two assays, namely a quantum chemistry (QC) simulation and an ML model; and ionization potential as the quantity to maximize. While simple, this configuration allows us to explore many relevant tradeoffs.

We implement these two assays as follows. For the QC assay, we use the NWChem simulation code [16]. We first parse the SMILES string and generate an approximate geometry using RDKit [17]. We then use NWChem through the QCEngine Python interface [18] to compute the equilibrium geometry for the neutral and oxidized molecule and then compute the vibrational modes for each molecule. All computations are performed at the B3LYP/3-21G level of accuracy and typically require six node-hours per molecule on four nodes of the Theta system at the Argonne Leadership Computing Facility, as described in Section IV-B.

The ML assay uses an ensemble of message-passing neural networks (MPNNs) [19] implemented in Tensorflow, each trained using a different subset of the training data. We employ an ensemble of models to produce both a mean and an estimate of model uncertainty for each prediction. The initial ensemble of 16 MPNN models were trained using 2563 oxidation potentials computed with the QC assay. We use this additional dataset and any new data in subsequent retraining tasks, which we limit to 15 minutes on a single node. To apply the ML model, we first use RDKit to parse the SMILES string, featurize the data in a form used by a Tensorflow ML model, and then evaluate the MPNN ensemble. It requires $3 \times 10^{-6}$ node-hours to evaluate a single molecule, which equates to 100 molecules per node-second.

## III. Our Approach

We formulated the decision problem abstractly as a sequential process, where planning and simulation tasks are performed serially. However, in order to use highly parallel computers to accelerate the exploration process, we want to allow simultaneous execution both of *multiple instances of the same activity* (for example, applying an assay to multiple entities) and of *different activities* (e.g., running a simulation-based assay, retraining an ML model with simulation results, running an ML assay, deciding which entities to explore next). Running multiple instances of the same activity at once is important because, at least in the applications that we consider here, no single action can scale efficiently to use all of a large parallel computer. Running different activities at the same time is important because different actions vary greatly in their computational demands; thus, strict sequencing would reduce parallel efficiency. However, achieving high parallel efficiency is difficult in practice due to competing needs for efficient execution (demanding high parallelism, modest communication), resource management (dynamically reallocating resources), and timeliness of information (making results from one computation available rapidly to computations deciding on next actions). We have implemented Colmena to solve these challenges.

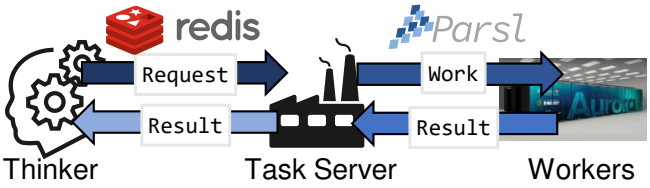


Fig. 1: Illustration of the architecture of a Colmena application. The **Thinker** communicates task requests to the **Task Server**, which distributes computations across many **Workers**. Communication between Thinker and Task Server occurs via Redis; Task Server and Workers communicate via ZeroMQ channels managed by Parsl.

### A. Colmena Architecture

Applications built using Colmena are formed of three types of independent processes: a **Thinker**, a **Task Server**, and one or more **Workers** (as shown in Figure 1).

The user-supplied **Thinker** implements the decision-making policy used to generate new tasks, record assay results ($D$), and update ML models ($E$). As described in Section II-A, tasks can include performing a new assay on a specified entity, updating a learned assay, and generating new entities. The Thinker communicates task requests to the Task Server; the results of those tasks, once available, are returned from the Task Server to the Thinker. The Thinker makes decisions in response to results being communicated or other events (e.g., availability of compute resources).

The **Task Server** matches each task request to the corresponding task definition (e.g., assay definition) and dispatches the resulting task to an appropriate Worker. The Task Server itself holds the assay definitions ($A$), information about available computational resources, and details of which assays can run on what resources. Task requests are received from the Thinker and can be executed in any order.

Each **Worker** receives a sequence of tasks ($\{$*task input*, *task definition*$\}$ pairs) from the Task Server. It executes each task that it receives and provides results back to the Task Server.

From a decision perspective, the challenge is to allow these activities to run with maximum concurrency and performance.

### B. Colmena Implementation

Our implementation of Colmena uses Redis [20] for asynchronous communication between the Thinker and Task Server, and the Parsl [21] parallel programming library to manage execution of tasks on the Workers.

*1) Thinker:* The Thinker decides which tasks are run and how to allocate resources among them. It communicates with the Task Server by writing task requests to, or reading results from, Redis queues. Requests and results are communicated as JSON objects and contain the inputs to a task, the outputs of the task, and a variety of profiling data (e.g., task runtime,

time inputs received by Task Server). We provide a Python API for the message format, with utility operations such as accessing the positional or keyword arguments for a task and serializing inputs and results.

A Thinker is composed of multiple Agents, implemented as threads. As we demonstrate in Section IV, separating the decision process into multiple cooperative agents simplifies expressing resource allocation and task selection strategies. Each agent can communicate with the Task Server via Redis and with other agents via Python's threading library. Each agent typically responds to a different kind of event (e.g., resources becoming available, completion of a certain type of assay). For example, one agent might submits task requests to the Task Server from a task queue as resources become available, and a second respond to results arriving from the Task Server by adding new tasks to the task queue.

The `BaseThinker` class in Colmena simplifies writing Thinkers with multiple agents. Agent processes run as separate Python threads that communicate with the Task Server via the Redis queue and with each other via Python threading tools. Agents are implemented as methods of a `BaseThinker` subclass and then marked with the `@agent` decorator, as illustrated in Listing 1. There are also special-purpose types of agents, such as a "result processor," that run after certain conditions are met, such as a result completing. The `BaseThinker` class provides a function that launches all of the decorated functions to run concurrently.

Colmena also provides a class for monitoring, controlling, and allocating resources between different assays. The resource allocation object, `ResourceTracker`, stores a fixed count of resources that are available to a Thinker class and how they are assigned into different pools. Agent threads may query the availability of resources in each pool, acquire or release them, and change the levels of allocation between different pools. The class uses Python's Lock and Semaphore objects so that resource requests can occur and be fulfilled concurrently.

*2) Task Server:* The Task Server is a stateful entity that performs high-throughput task processing. It receives task requests from an *inputs* queue and posts results asynchronously to an *outputs* queue when a task is complete. The Task Server requires a robust and performant backend for managing the execution of a diverse set of potential assays—from short-running, single-core inference tasks to long-running, multi-node MPI simulations. Further, the Task Server should provide an intuitive way to represent diverse assays, transparently serialize and transfer inputs/outputs to/from Workers, dynamically provision resources in heterogeneous environments (e.g., clusters and clouds, CPUs and GPUs), scale and make efficient use of large-scale systems, elastically adapt resource configurations in response to workload, and provide fault tolerance to reliably execute assays with performance monitoring, error capture, and checkpoint/retry.

While there are several potential parallel and distributed computing toolkits and workflow systems (e.g., funcX [22], Ray [23], Swift/T [8]) that could be used for this purpose, we implement the Task Server using Parsl [21]. Parsl

```python
from colmena import thinker

PARALLEL_TASKS = 3
TOTAL_TASKS = 10

class Thinker(thinker.BaseThinker):
  def __init__(self, queues):
    super().__init__(queues)
    self.next_task = None
    self.results = []

  @thinker.agent
  def planner(self):
    # Submit initial tasks
    for _ in range(PARALLEL_TASKS):
      self.queues.send_task(random(),
                            task='simulate')
    # Until enough work is done
    while len(self.results) < TOTAL_TASKS:
      # Get ideas from the old results
      good_idea = f(self.results)

      # Update the next task
      self.next_task = good_idea

  @thinker.result_processor
  def consumer(self, result):
    # Store the result in the database
    self.results.append((result.args,
                         result.value))
    # Submit the next task in queue
    self.queues.send_task(self.next_task,
                          task='simulate')

thinker = Thinker(queues)
thinker.run()
```

Listing 1: An example Thinker that implements the simple policy, "run 10 tasks in total, three at a time, generating a new task based on previous results as each task completes." Its implementation comprises two agents, `planner` and `consumer`. The `planner` first sends three initial task requests to the Task Server, and then continually computes the best-possible next task to perform, given the current state. The second `consumer`, invoked whenever a task completes, stores the result and submits the next task. The decorator `@agent` causes the `planner` to be launched as a thread when `thinker.run()` is called; the decorator `@results_processor` indicates that `consumer` should be run, also as a thread, each time that a task completes.

is a parallel programming library for Python that extends Python's native `concurrent.futures` interface to enable high-performance, distributed computation and dataflow-based workflows. Parsl provides the runtime infrastructure to execute Python tasks asynchronously on various compute resources. It is able to serialize Python functions and input arguments, transfer those functions and arguments to a remote system, execute the function in the configured Worker environment, and retrieve results and errors. Parsl's modular design and

standard interfaces enable workloads to be executed using various *executors*, such as via pilot jobs or a distributed MPI job, and by interacting with various job schedulers and cloud APIs, such as Slurm, PBS, and Amazon Web Services (AWS). The ability for Parsl to interface with job schedulers and cloud APIs, in particular, opens the possibility for writing application adjust the amount of resources devoted to a problem during the course of an application (e.g., reducing the simulation resources while ML models are retraining).

Users create a Task Server by providing a list of tasks and specifying, using Parsl's Python-based notation, the target computational resources. The tasks are defined as Python functions, which we wrap using Parsl's `PythonApp` to allow the functions to be executed remotely. Each assay can be mapped to different computational resources, making it possible to run assays on different resources (e.g., specialized hardware) or use different types of Workers (e.g., single-node vs multi-node) on the same resource.

*3) Communication:* The Thinker and Task Server communicate via Redis queues, with distinct *request*/*result* queue pairs for different task types (e.g., different assays, ML training). The Thinker writes requests to the appropriate *request* queue, to be received by the Task Server; when task execution completes, the Task Server writes the result to the corresponding *result* queue, from which it is read by the Thinker. This use of different queues for different task types simplifies implementation of Thinkers with multiple sub-agents.

Upon receiving a *task* request, the Task Server creates and launches a corresponding Parsl task. Parsl uses a hierarchical communication model, with ZeroMQ channels to efficiently distribute tasks to its Workers. As the Task Server receives results from Workers over these channels, it posts each to the appropriate *result* queue.

For tasks with large input or result values, Colmena uses a **Value Server** to pass values directly from the Thinker to the Worker—bypassing the Task Server. The Value Server uses Redis as the backend key-value store and exposes a lazy object proxy interface. Lazy object proxies simplify interaction with the Value Server because (1) the proxies behave as the wrapped object so users do not need to modify code to accommodate the proxies, (2) the proxies automatically handle retrieving data from the value store once the data are first needed, and (3) the lazy aspect of the proxies can amortize communication costs with the Value Server.

Any arbitrary object $v$ can be wrapped in a proxy. The process of wrapping $v$ involves placing $v$ into the Value Server and returning a proxy $p$ that stores the key associated with $v$ in the Value Server and some additional metadata. Proxies behave like the underlying object, e.g., `isinstance(p, type(v))==True`. The proxy $p$ is lazy in that it acts as a reference to $v$ until $p$ is accessed. Thus, $p$ is cheap (in terms of serialization and communication costs) to include as a task input in the {*task input*, *task definition*} pair. When first used, $p$ is *resolved*, meaning $v$ is retrieved from the Value Server and stored inside $p$ such that $p$ can be used as $v$ would be.

Colmena can automatically proxy input and result values larger than a user-defined threshold, and/or users can manually proxy large objects. The Value Server has a Worker-level cache to speed up tasks that reuse the same inputs (e.g., the model for ML inference tasks). Proxies can be asynchronously resolved, allowing for the overlap of Value Server access and computation. Colmena starts asynchronously resolving all proxies in a task's input prior to the task being executed on a Worker. Thus, the communication with the Value Server is overlapped with the task's execution. The start of a task often involves some initialization or importing of libraries, such that by the time a value is needed by the task, the corresponding proxy has already been resolved in the background.

## IV. Application Experiments

We conducted experiments to evaluate the performance of our molecular design application and then further studied the performance of components that were bottlenecks in the molecular design application.

### A. Application Description

As introduced in Section II-B, our example application involves an ML-guided search of $10^5$ molecules for those that match one of our design criteria for molecular electrolytes, namely high ionization potential (IP). At a high level, our application is an adaption of Bayesian Optimization to HPC. We determine a score of each task using the Upper Confidence Bound (UCB), a value based on the mean and confidence interval of the predictions from an ensemble of MPNNs trained to predict the IP from the bonding network of a molecule. UCB defines the highest scoring molecules as those with large means and large confidence intervals [24], which are those likely to both have high performance and provide data that will improve the models (i.e., because they are in regions where the model is least certain). Molecules are evaluated in order of descending scores. The data from completed simulations are used to update the MPNNs and, through the updated models, produce better estimates of molecular properties. Many simulations are performed in parallel as NWChem scales poorly for the small molecule sizes considered in this study.

More formally, the application uses two assays: a more expensive and accurate QC assay and an inexpensive but less accurate ML assay trained on QC results. As we are using a pre-defined search space of molecules, no generator is needed to expand the set of molecules considered progressively during execution. Instead, the application's Thinker maintains two data structures—a **molecule queue** of {*molecule*, *ML-score*} pairs, ordered by *ML-score*, and a **results record** of {*molecule*, *QC-score*} pairs—that are manipulated by the following three pairs of agents (see Figure 2):

- The **Trainer** periodically sends to the Task Server a `retrain` task to retrain the ML model based on data in the results record; the companion **Updater**, upon receiving results for such a task, updates the weights of the ML model.
- The **ML-Scorer**, whenever the ML assay is updated, sends `ML-assay` tasks to the Task Server to re-score

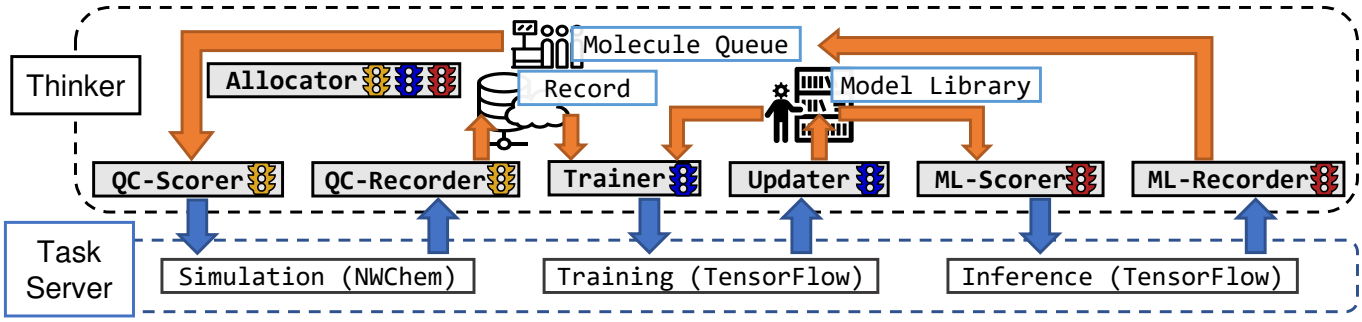


Fig. 2: Implementation of molecular design application with Colmena. Agents within the Thinker application are gray boxes, and the three different available tasks are listed as white boxes. Blue arrows indicate communication between Thinker and Task Server; orange arrows illustrate how information flows between agents. Different colored traffic lights indicate resource pools used for different task types. The Allocator agent reallocates resources between different pools.

every entity in the molecule list; the companion **ML-Recorder**, upon receiving results of these tasks, computes the Upper Confidence Bound (UCB) for each and reorders the molecule list with the new information.

- The **QC-Scorer** repeatedly removes a molecule from the front of the molecule list and sends a `QC-assay` task to the Task Server to determine its QC score; the companion **QC-Recorder**, upon receiving results for these tasks, stores them in the results list if they pass validation.

The application's precise behavior thus depends (in addition to the total number of available resources) on the allocation of available resources to `retrain`, `ML-assay`, and `QC-assay` tasks: increasing the fraction allocated to `QC-assay` tasks leads to relatively more QC assays being performed, while increasing the fraction allocated to `retrain` and `QC-assay` tasks increases the timeliness of the ML-based scores, and thus in principle leads to more relevant QC assays being performed.

The strategy for controlling the resource allocations is implemented as an **Allocator** agent. The application's **Allocator** balances competing demands for resources with a policy that, after an initial set of ML assay results are obtained with a pretrained ML model, (a) retrains the ML model each time that the results record increases in size by an amount $n\_retrain$, (b) reruns the ML assay on all entries in the molecule list whenever the ML model is updated, and (c) reallocates resources between `retrain`, `ML-assay`, and `QC-assay` tasks as needed to ensure that `retrain` and `ML-assay` tasks are run as fast as possible when generated, with resources otherwise being used for `QC-assay` tasks. Resource reallocations are performed by controlling the amount of requests sent to the Task Server, which generates requests to the Parsl backend to stop or start Workers. Resources allocated to different purposes thus change in increments of the largest number of nodes needed for a single task, which in this case is four nodes for the `QC-assay` task.

In addition to the molecule list and results record already mentioned, these processes also share a resource counter (used to track resource availability) and a library of ML models.

### B. Experimental Setup

We used the Theta supercomputer at the Argonne Leadership Computing Facility (ALCF) [25], a 11.69-petaflop system based on the second-generation Intel Xeon Phi "Knights Landing" (KNL) processor. Its 4392 nodes each have a 64-core processor with 16 GB MCDRAM and 192 GB of DDR4 RAM, for a total of 281 088 cores; nodes are interconnected with a high speed Cray Aries network. The application was configured so that the Thinker process ran on the machine-oriented miniserver (MOM) node, each NWChem task was allocated four nodes, and each Tensorflow task ran on one node each. Tensorflow inference tasks were grouped into batches of 4096 for efficient multithreaded execution on each node.

### C. Application Evaluation

We evaluate the performance of our molecular design application from the two perspectives of computational performance (specifically, computational efficiency) and the quality of the results obtained.

*1) Performance Evaluation:* We evaluate the performance of the application by measuring the fraction of the time worker processes spend performing the computational tasks requested by the thinker (e.g., simulation, ML inference) and not communicating work to/from the Task Server.

As shown in Figure 3, we maintain near 100% utilization for most of a 1024-node run of our application. We note two major sources of under-utilization. The first is the start-up time for inference Workers, which is a median of 3 minutes. The startup cost can be reduced by unpacking Python libraries to node-local memory before launching Parsl Workers [26]. The second source of under-utilization is simulation tasks that do not complete within the timescale of the job, which leads to the associated resources being counted as unutilized even though the calculations are running (as verified via Parsl logs). This source of under-utilization can be mitigated by periodically checkpointing simulation tasks or by splitting simulation tasks into smaller steps, such as computing the neutral geometry first and then computing the ionization potential. The latter approach has the advantage that the Thinker can use interme-

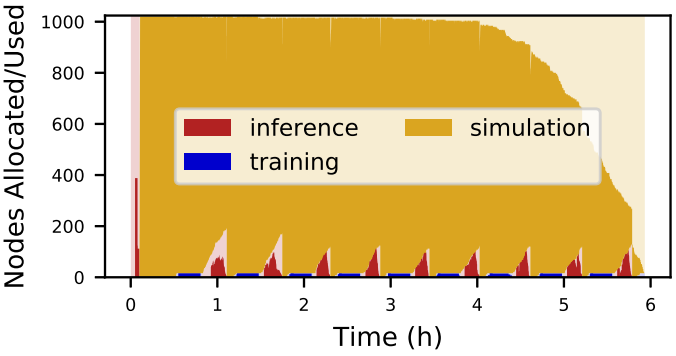


Fig. 3: Allocation and utilization of compute resources during a run of the molecular design application on 1024 Theta nodes. Light shades indicate the nodes that are allocated to a problem; dark shades are nodes that are both allocated and are being used to execute assays. Different colors indicate different types of tasks. The large decrease in utilization at later times is a result of trailing tasks [27].

diate results to decide whether to continue a computation (e.g., if it is likely to complete before the end of an allocation).

The overheads due to communication are minimal for `QC-Assay` tasks. The median cost for launching a new simulation is 1015 ms: 620 ms for the result to be received by the Thinker, 35 ms for submitting the new task to the Task Server, and 360 ms for the task to be launched on a Worker. The cost here is minimal (0.03%) compared to the median simulation runtime of 3275 s. We note that there is a significant variation between the largest cost, result communication, with that of failed tasks often 10× shorter than successful tasks. The difference can be attributed to the amount of data transferred, with failed tasks typically sending 0.5 KB and successful tasks communicating a median of 4.3 MB.

The communication cost of the `ML-Assay` tasks is partly hidden by prefetching tasks to Workers, but we note two issues that lower utilization and could inhibit further scaling. One is the startup time for the Workers used for ML assays. We find that it takes a median of 175 s for a Worker to begin work after being sent its first task from the Task Server, due to the startup time noted earlier. The second startup cost is the time it takes to compile a TensorFlow model before execution. The first task run by a Worker requires a median of 100 s, and the second task requires only a median of 80 s. We observe similar startup issues for the training tasks.

A second issue for inference is the time required to communicate results from a Worker to the Task Server. The communication time for an inference task request is 500 ms (0.6% of the median execution time), which can be hidden by prefetching work to the Workers. While this cost has not been a problem in the simulations run to date, it may become so at larger scales or problems. As with the `QC-Assay` tasks, we associate the long communication times with transferring large data objects. The training tasks require 30 s to communicate the $>$ 50 MB updated ML model, a small (3%) fraction of the median run time of 850 s.

In summary, we find the Colmena Task Server fulfills the performance requirements required for our application. We can change the allocation between node-parallel and single-node tasks during the run, with the latency dominated by the time to start a Python interpreter ($\sim$100 s) for single-node tasks. The latency for launching new tasks after a simulation completes is low ($<$1s), though we note this latency can increase to several seconds for tasks with large inputs (e.g., entire deep learning models). Overall, the system was able to maintain a total utilization of 85% during the run with the largest source of underutilization resulting from the trailing tasks.

*2) Molecular Design Performance:* We assess the performance of the application at solving our target problem by studying the scores of the molecules in the record over a run. We perform several runs using 256 nodes where we either retrain the ML assay on-demand during the run (as in Figure 3) or only once at the beginning of the run. For context, we also compare both versions of the application to a run where we select tasks randomly.

Figure 4 shows QC results as a function of time for Thinkers with different policies. The two Thinkers that use ML models to select molecules for QC simulation perform much better than the one that selects molecules at random, finding over 100× more high-performing molecules with ionization potentials above 10 V. Normalizing by the total number of molecules evaluated during the run, the random agent finds a high-performance molecule with a success rate of 0.5%, whereas the success rates are 78% and 64% for the runs with and without retraining tasks. In short, we find a significant advantage in using ML assays to prioritize the order in which we consider a molecular design space and also an advantage to reprioritizing the list of simulations during a run. The application where we respond to new simulation results finds 10% more molecules with large ionization potentials even though it spends 5% less time on simulations.

The benefits of active learning are clearest at the end of the run, where the application uses data from the largest number of previous simulations to select new inputs. In the last hour of each run, the average ionization potential of the application with retraining was significantly higher than the run without (10.5 V vs. 9.8 V), clearly illustrating that retraining leads to an improved ability to identify high-value simulations. The effect can potentially be traced to improvements in the machine learning models. The initial models have a mean absolute error (MAE) of 0.395 V on 185 molecules selected at random of the search space and the MAE of the models is reduced to 0.389 V by the second re-training event and 0.382 V by the last batch of the run. The performance gains are subtle but the increased search performance illustrates how even minor improvements in a model can lead to an improved ability to select new molecules. The challenge then becomes being able to perform model updates quickly.

Our application provides reasonable response times between when a simulation completes and its data are used to select the next simulations. The time-to-solution for retraining is an

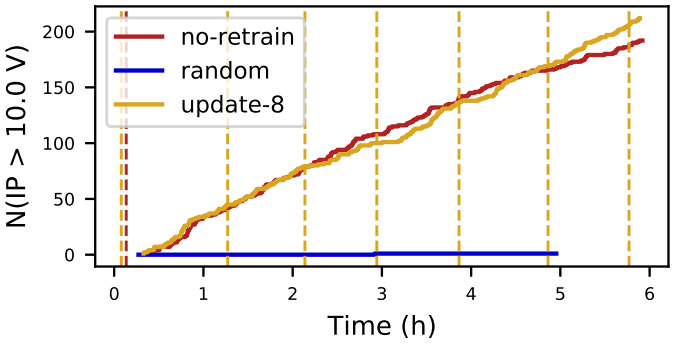


Fig. 4: The number of high-performing molecules (i.e., with ionization potential greater than 10 V) evaluated over time by the Colmena molecular design application when using three different Thinker strategies: *random*, which selects molecules for QC simulations at random; *no-retrain*, which selects molecules for QC simulations based on an ML model that is trained once; and *update-8*, which selects molecules based on an ML model that is retrained after 8 QC simulations complete successfully. Vertical dashed lines indicate the times at which the molecule list was reordered after model rerunning `ML-Assay` tasks.

average of 57 minutes between when a simulation completes and the task list is updated based on its results, in which time an average of 63 new simulations are submitted—15% of the 537 submitted during the entire run. Such results demonstrate the need to dedicate specific resources to the ML tasks to keep up with the rate data can be produced by the simulation codes. Dedicating fewer nodes to the retraining or inference task would slow how quickly the application responds to new data, to the detriment of the active learning process.

### D. Component Evaluation

As discussed in Section IV-C1, we observed that transferring large requests or result objects is a major source of communication costs, and the `ML-Assay` tasks are the most susceptible to the effect of this bottleneck. Consequently, we first evaluate a system that optimizes large ($>$ 10 MB) result transfers and then evaluate its effect on `ML-Assay` tasks. This is the done through inclusion of the Value Server, which reduces the requirement of serialization and deserialization of task data. We discuss the performance improvement through the inclusion of a Value Server using a synthetic problem. We then discuss the improvement that can be achieved in real-world production runs for the electrolyte design problem.

*1) Synthetic Application:* We built a synthetic application, SynApp, to permit the systematic evaluation of Colmena communication overheads. This application uses a Thinker plus $N$ workers, one per node; the Thinker generates $T$ identical tasks, each with duration $D$, unique (and thus non-cacheable) input of size $I$, and producing a result of size $O$. This Thinker first submits one task per worker and then continues to submit a new task each time that it receives a result, until $T$ tasks have been submitted. We use this application to measure costs for different $\{T, D, I, O, N\}$ combinations.

To evaluate the impact of the Value Server, we first run SynApp for 200 zero-length tasks with 1 MB inputs on eight nodes (i.e., $\{T$=200, $D$=0, $I$=1 MB, $O$=0, $N$=8$\}$), both with and without the Value Server, while measuring task overheads. We see in Figure 5 that the use of the Value Server reduces, in particular, task communication times between the Thinker client and Task Server, and serialization times. The cost of transferring input data from the Value Server to the Worker is reduced by the use of the asynchronous data retrieval explained in Section III-B3. (Note that if input values were all identical, this cost would be largely eliminated due to caching.)

To further study how the benefit of the Value Server varies with input size $I$, we repeat the experiments of Figure 5 but while varying $I$ from 1 KB to 10 MB. The results, shown in Figure 5 as percentage improvement in communication overhead time *with* Value Server relative to the time *without* Value Server, show that for small inputs ($<$10 KB), the additional cost of communicating with the Value Server is larger than the cost of passing the input data through the Task Server—but that as the input size increases, the cost of passing input data through the Task Server increases rapidly and the Value Server yields large improvements.

*2) Scaling ML Assays:* As discussed in Section IV-D1, there is a clear benefit to using the Value Server for tasks with large inputs or large results. We now study a particular sub-problem within the electrolyte design application: running machine learning inference tasks. The input task size was varied by changing the total number of molecules evaluated as a function of the number of nodes allocated for ML inference. The resulting evaluation rate, in molecules per second, is presented in Figure 6.

As noted in Section IV-C1, individual ML inference tasks do not take long to run. Nevertheless, the Value Server can deliver significant benefits even in this case when inference results must be transferred from many nodes. In Figure 6, we examine the mean time taken to transfer results from Worker to Thinker, with and without the Value Server. We see that, without the Value Server, it takes up to 100 s to transfer ML inference results from more than 100 nodes. With the Value Server, however, the mean transfer time as a function of increasing node count remains constant.

We observe significant improvements in the evaluation rate at 1024 nodes with the Value Server. The time to communicate results from a completed job remains $\sim$100 ms at 1024 nodes when using the Value Server, in contrast to the $\sim$100 s transfer time without (Figure 6), indicating that the workflow engine in the Task Server is not getting overloaded. Consequently, we maintain ideal scaling up to at least 1024 nodes and reasonable performance at 2048 nodes.

## V. Related Work

Colmena requires methods for creating tasks and for monitoring and managing their execution. It needs to support a wide range of types and scales, from multi-hour, many-node QC

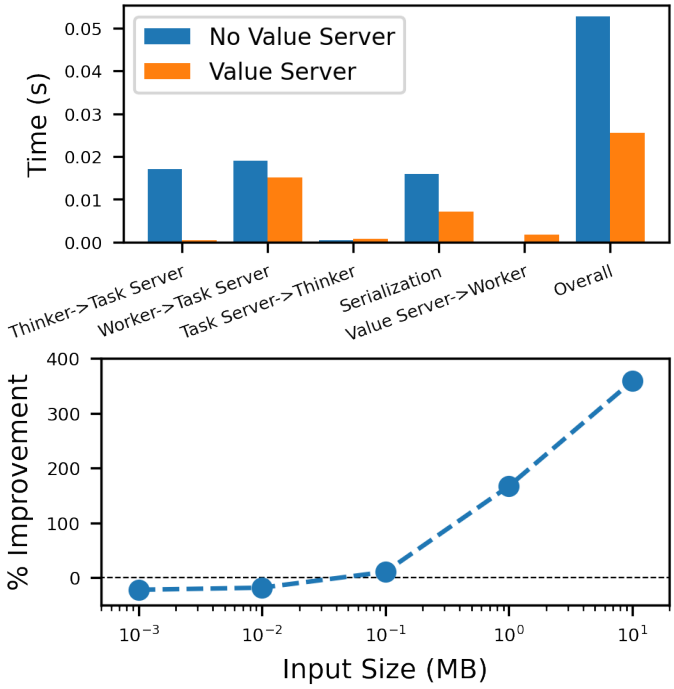


Fig. 5: (top) Median per-task durations for components in the Colmena task life cycle on Theta, with and without the Value Server, as measured for SynApp with eight workers, zero-length tasks, 1 MB inputs, and 0 B outputs. Use of the Value Server reduces time spent serializing, communicating, and deserializing task data. (bottom) Percent reduction in SynApp overheads for configuration $\{T$=200, $D$=0, $I$, $O$=0, $N$=8$\}$ on Theta, with vs. without the Value Server, as a function of input size $I$. The Value Server provides performance benefits when task inputs are larger than around 0.1 MB.

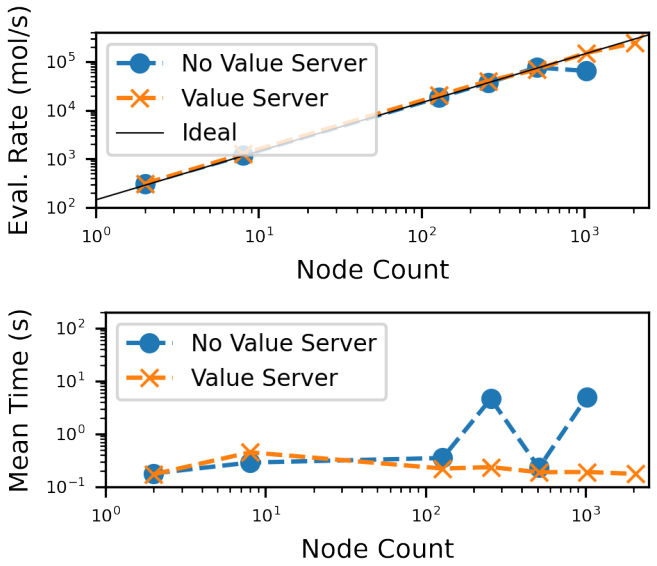


Fig. 6: (top) ML inference task performance (molecule evaluations per second) on Theta vs. number of nodes (one worker per node). The inference rate is measured starting from the time the first worker begins computation (i.e., after loading libraries) to when all inference tasks have completed. (bottom) Time to transfer inference results from Workers to the Thinker. Without the Value Server, results may take up to 100 s to be sent back to the Thinker, while with the Value Server, the communication time remains more consistent.

tasks to minute-duration, single-node ML tasks. Accordingly, our work with Colmena builds upon much previous work.

**General Steering Frameworks**. Nimrod/O [7] is an early example of a system for automated (rather than manual [28]) steering of simulation ensembles. Several general-purpose toolkits for automated steering of ensembles have been proposed in recent years, each with different models for coupling steering and simulation. Here, we describe their key features and how they inspired our development of Colmena.

The DeepHyper [12] hyperparameter optimization system uses a centralized planning process to select tasks and distributes a single type of assay across multiple nodes using a workflow engine. The planning process submits tasks to the workflow engine (DeepHyper supports Balsam [29] and Ray [23]) and queries the engine for completed results. Supervisor [3] has a similar centralized architecture to DeepHyper, with a single node for communicating with the high-performance Swift/T [8] workflow engine via a queue. Proxima [30] uses ML methods to dynamically tune a surrogate-modeling configuration in response to real-time feedback from the ongoing simulation, but does not optimize for use of HPC. Colmena uses a similar centralized model for task planning; is designed, like DeepHyper, to support multiple workflow engines; and provides easier support for multiple task types.

LibEnsemble [10] expresses ensembles using a model where tasks produced by *task generation* (akin to steering) workers are executed by *simulation* workers, and simulation results are fed back to inform the generation tasks. The steering policy can be either centralized (single task generator) or decentralized (multiple task generators). A manager service deploys the task generation and simulation workers onto multiple nodes and routes data across workers. Colmena shares LibEnsemble's ability to dedicate more than one node to steering-related tasks and provides the ability to break policies into an unlimited number of worker types.

Ray [23] uses a similar distributed model to LibEnsemble but permits ensembles to use many types of interacting agents (beyond generator and simulator) with complex coordination policies. Colmena has a similar agent-oriented programming model but centralizes all agents to a single node so that they can communicate with shared memory.

Rocketsled [9] expresses the steering task as a step within a Fireworks [31] workflow that can add new nodes to the workflow graph. Colmena uses a different programming model where planning tasks need not be triggered by workflow events and was designed to separate planning logic from the task execution engine. Cray SmartSim [32] allows multiple planning scripts to independently launch jobs on a cluster and

coordinate among each other using Redis. The Colmena and SmartSim programming models are similar, in that multiple planning scripts can submit and receive work concurrently, and possible tasks can be enumerated as a list of assays. Colmena has utilities that simplify building planning strategies for ensemble steering (e.g., pulling from result queues, event-triggered resource reallocation), whereas SmartSim is capable of building other types of AI+simulation applications (e.g., online analysis, inference from simulation codes).

In short, the Colmena toolkit is purpose-built for expressing the complex policies needed to make efficient use of highly parallel supercomputers for computational campaigns. We designed Colmena to provide many of the features of current frameworks for steering computational campaigns, including the centralized programming model of codes such as DeepHyper or Supervisor, the ability to deploy planning tasks across multiple nodes illustrated by libEnsemble, and simple routes to building planning policies as cooperative agents in the style of Ray and SmartSim. Colmena presents a single package able to recreate the parallelization strategies of all current steering frameworks and provides the flexibility needed to explore even more sophisticated approaches.

**Active Learning on HPC**. Active learning methods obtain new training data via online querying of an information source, including (as here) computational simulations. The approaches just reviewed may be viewed as active learning methods.

**Reinforcement Learning on HPC**. Algorithms for training reinforcement learning models have much in common with those for steering ensemble simulations. Reinforcement learning agents gather data by using a policy to guide the evolution of different environments (e.g., simulations for physical systems) and periodically retrain this policy to better steer the environments towards desirable states. Computations that simulate environments, make policy decisions, and retrain policies can all occur asynchronously across distributed resources, akin to Colmena's Thinker/Task Server model for ensemble simulations. Consequently, the design of steering algorithms is related to approaches for distributed training of reinforcement learning (e.g., IMPALA [33]) and to toolkits for deploying reinforcement learning at scale (e.g., RLLib [34], ExaRL [35]). Such algorithms for training reinforcement learning policies are a class of approaches that can be expressed with Colmena.

**Specialized Steering Frameworks**. Specialized applications that dynamically create or reorder tasks are also prevalent in the literature. Various domain-specific tools (e.g., XtalOpt [36], Kombine [37], DeepDriveMD [13], CAMD [11]) engage different patterns for generating tasks in solving different classes of problems (e.g., optimization, parameter estimation). Tools that perform hyperparameter searches for neural network design, such as DeepHyper [12] and Tune [38], illustrate how to handle ML tasks at scale. Colmena adapts concepts from these tools; each of the algorithms in these tools can be implemented using Colmena.

**Workflow Systems**. Colmena also relies heavily on workflow systems to distribute computations across many nodes. Applicable workflow systems include Ray [23], Balsam [29], RADICAL Cyber Toolkit [39], Parsl [21], Fireworks [31], and many others [40]. Such systems provide unique approaches to specifying tasks and runtime systems for executing them across distributed resources. Colmena is designed to make use of workflow tools and not to make any new contributions in the design of workflow systems.

**Process Management Systems**. Methods for deploying many concurrent, short-duration tasks is another area of active research. The challenges are well explained in a recent work that studied many-task performance on Summit [41]. The Process Management Interface (PMIx) [42] defines an API for such capabilities that is available on some supercomputers (e.g., Summit). There are also efforts, such as MPI_Comm_launch [43], to incorporate process management methods into the Message Passing Interface. Systems for launching, monitoring, and managing large numbers of tasks on HPC are going to be critical as we scale Colmena to larger computational resources.

## VI. Conclusions

We introduced Colmena, an open-source Python library for machine-learning-based steering of ensemble computations on HPC systems. We first formalized the steering process as a design problem where one must decide which computations to perform on what inputs to produce a record of simulations with maximal value at minimal cost. We then described how Colmena facilitates building such steering applications by permitting the construction and composition of a Thinker that implements the decision making processes used to define computational tasks and a Task Server that distributes execution across HPC resources. We illustrated the use of the Colmena library with a molecular design application that finds molecules with high resistance to oxidation at rates $100\times$ faster than naive searches by interleaving simulation and ML tasks. We demonstrated effective scaling on up to 1024 nodes (65 536 cores) and illustrate how to improve the scaling of the application further by using a separate subsystem for transferring large result objects.

We intend that Colmena provides a toolkit for exploring methods for steering ensemble simulations. The flexible, multi-threaded Thinker class permits implementing varied, complex policies for interleaving different types of computation. Our primary goal for creating Colmena is to support the expression of steering policies that use ML to augment human intelligence in designing and managing computational campaigns. Interfaced with a Task Server built using Parsl, users can execute these policies at large scales and across heterogeneous computing resources. As we learn more, we will build templates for common classes of decision problems (e.g., model-based optimization, reinforcement learning) that allow users to quickly deploy state-of-the-art steering policies. Through this work, we hope to enable computational campaigns that take fuller advantage of current and next-generation supercomputers.

## Data and Software Availability

The source code used in the this manuscript, logging information for each of the runs described in the paper, and Jupyter notebooks used to analyze logs and produce figures are all published via the Materials Data Facility.[44], [45] The source code and Jupyter notebooks associated with this manuscript is also available on GitHub at https://github.com/exalearn/electrolyte-design/, which will be updated as our work proceeds.

## Acknolwedgements

LW, GS, GP, RC, RT, and IF acknowledge support by the ExaLearn Co-design Center of Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, to develop Colmena and evaluate its performance on HPC. YB and KC were supported to integrate Parsl support into Colmena by NSF Grant 1550588 and ExaWorks Project within the Exascale Computing Project. GP and KC were supported to develop the value server by NSF Grant 2004894. LW, ND, PCR, RSA, and LAC were supported to define the electrolyte design problem and develop the computational workflows need to solve it by the Joint Center for Energy Storage Research (JCESR), an Energy Innovation Hub funded by the US Department of Energy, Office of Science, Basic Energy Sciences. This research used resources of the Argonne Leadership Computing Facility (ALCF), which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357, and was supported by the ALCF Data Science Program.

## References

[1] Q. Wu, B. He, T. Song, J. Gao, and S. Shi, "Cluster expansion method and its application in computational materials science," *Computational Materials Science*, vol. 125, pp. 243–254, Dec. 2016. [Online]. Available: https://doi.org/10.1016/j.commatsci.2016.08.034

[2] D. M. Probst, M. Raju, P. K. Senecal, J. Kodavasal, P. Pal, S. Som, A. A. Moiz, and Y. Pei, "Evaluating optimization strategies for engine simulations using machine learning emulators," *Journal of Engineering for Gas Turbines and Power*, vol. 141, no. 9, Jun. 2019. [Online]. Available: https://doi.org/10.1115/1.4043964

[3] J. M. Wozniak, R. Jain, P. Balaprakash, J. Ozik, N. T. Collier, J. Bauer, F. Xia, T. Brettin, R. Stevens, J. Mohd-Yusof, C. G. Cardona, B. V. Essen, and M. Baughman, "CANDLE/Supervisor: A workflow framework for machine learning applied to cancer research," *BMC Bioinformatics*, vol. 19, no. S18, Dec. 2018. [Online]. Available: https://doi.org/10.1186/s12859-018-2508-4

[4] S. Jiang, G. Malkomes, M. Abbott, B. Moseley, and R. Garnett, "Efficient nonmyopic batch active search," in *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., vol. 31. Curran Associates, Inc., 2018. [Online]. Available: https://proceedings.neurips.cc/paper/2018/file/a7aeed74714116f3b292a982238f83d2-Paper.pdf

[5] K. Kandasamy, A. Krishnamurthy, J. Schneider, and B. Poczos, "Parallelised Bayesian optimisation via Thompson sampling," in *21st International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, A. Storkey and F. Perez-Cruz, Eds., vol. 84. PMLR, 09–11 Apr 2018, pp. 133–142. [Online]. Available: http://proceedings.mlr.press/v84/kandasamy18a.html

[6] B. Peherstorfer, K. Willcox, and M. Gunzburger, "Survey of multifidelity methods in uncertainty propagation, inference, and optimization," *SIAM Review*, vol. 60, no. 3, pp. 550–591, 2018.

[7] D. Abramson, A. Lewis, and T. Peachey, "Nimrod/O: A tool for automatic design optimisation using parallel and distributed systems," in *Algorithms And Architectures For Parallel Processing*. World Scientific, 2000, pp. 497–508.

[8] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, and I. T. Foster, "Swift/T: Large-scale application composition via distributed-memory dataflow processing," in *13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, 2013, pp. 95–102.

[9] A. Dunn, J. Brenneck, and A. Jain, "Rocketsled: A software library for optimizing high-throughput computational searches," *Journal of Physics: Materials*, vol. 2, no. 3, p. 034002, Apr. 2019. [Online]. Available: https://doi.org/10.1088/2515-7639/ab0c3d

[10] S. Hudson, J. Larson, S. M. Wild, D. Bindel, and J.-L. Navarro, "libEnsemble users manual," Argonne National Laboratory, Tech. Rep. Revision 0.7.1, 2020. [Online]. Available: https://buildmedia.readthedocs.org/media/pdf/libensemble/latest/libensemble.pdf

[11] J. H. Montoya, K. T. Winther, R. A. Flores, T. Bligaard, J. S. Hummelshøj, and M. Aykol, "Autonomous intelligent agents for accelerated materials discovery," *Chemical Science*, vol. 11, no. 32, pp. 8517–8532, 2020. [Online]. Available: https://doi.org/10.1039/d0sc01101k

[12] P. Balaprakash, M. Salim, T. D. Uram, V. Vishwanath, and S. M. Wild, "DeepHyper: Asynchronous hyperparameter search for deep neural networks," in *25th International Conference on High Performance Computing*. IEEE, Dec. 2018. [Online]. Available: https://doi.org/10.1109/hipc.2018.00014

[13] H. Lee, M. Turilli, S. Jha, D. Bhowmik, H. Ma, and A. Ramanathan, "DeepDriveMD: Deep-learning driven adaptive molecular simulations for protein folding," in *IEEE/ACM Third Workshop on Deep Learning on Supercomputers (DLS)*. IEEE, Nov. 2019. [Online]. Available: https://doi.org/10.1109/dls49591.2019.00007

[14] R. Ramakrishnan, P. O. Dral, M. Rupp, and O. A. von Lilienfeld, "Quantum chemistry structures and properties of 134 kilo molecules," *Scientific Data*, vol. 1, no. 1, pp. 1–7, 2014.

[15] "QM9 dataset," accessed August 30, 2021. [Online]. Available: http://quantum-machine.org/datasets/

[16] M. Valiev, E. J. Bylaska, N. Govind, K. Kowalski, T. P. Straatsma, H. J. Van Dam, D. Wang, J. Nieplocha, E. Apra, T. L. Windus *et al.*, "NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations," *Computer Physics Communications*, vol. 181, no. 9, pp. 1477–1489, 2010.

[17] G. Landrum, "RDKit: Open-source cheminformatics," https://www.rdkit.org. Visited May 1, 2021.

[18] D. G. A. Smith, D. Altarawy, L. A. Burns, M. Welborn, L. N. Naden, L. Ward, S. Ellis, B. P. Pritchard, and T. D. Crawford, "The MolSSI QCArchive project: An open-source platform to compute, organize, and share quantum chemistry data," *WIREs Computational Molecular Science*, vol. 11, no. 2, Jul. 2020. [Online]. Available: https://doi.org/10.1002/wcms.1491

[19] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *International Conference on Machine Learning*. PMLR, 2017, pp. 1263–1272.

[20] "Redis," accessed April 9, 2021. [Online]. Available: https://redis.io/

[21] Y. Babuji, A. Woodard, Z. Li, B. Clifford, R. Kumar, L. Lacinski, R. Chard, J. Wozniak, I. Foster, M. Wilde, D. Katz, and K. Chard, "Parsl: Pervasive parallel programming in Python," in *ACM International Symposium on High-Performance Parallel and Distributed Computing*, 2019.

[22] R. Chard, Y. Babuji, Z. Li, T. Skluzacek, A. Woodard, B. Blaiszik, I. Foster, and K. Chard, "funcX: A federated function serving fabric for science," in *29th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, Jun. 2020. [Online]. Available: https://doi.org/10.1145/3369583.3392683

[23] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A distributed framework for emerging AI applications," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 561–577. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/moritz

[24] T. Lai and H. Robbins, "Asymptotically efficient adaptive allocation rules," *Advances in Applied Mathematics*, vol. 6, no. 1, pp. 4–22, Mar. 1985. [Online]. Available: https://doi.org/10.1016/0196-8858(85)90002-8

[25] K. Harms, T. Leggett, B. Allen, S. Coghlan, M. Fahey, C. Holohan, G. McPheeters, and P. Rich, "Theta: Rapid installation and acceptance of an XC40 KNL system," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 1, p. e4336, 2018.

[26] T. Shaffer, Z. Li, B. Tovar, Y. Babuji, T. Dasso, Z. Surma, K. Chard, I. Foster, and D. Thain, "Lightweight function monitors for fine-grained management in large scale Python applications," in *IEEE International Parallel and Distributed Processing Symposium*, 2021.

[27] T. G. Armstrong, Z. Zhang, D. S. Katz, M. Wilde, and I. T. Foster, "Scheduling many-task workloads on supercomputers: Dealing with trailing tasks," in *3rd Workshop on Many-Task Computing on Grids and Supercomputers*. IEEE, 2010, pp. 1–10.

[28] J. D. Mulder, J. J. Van Wijk, and R. Van Liere, "A survey of computational steering environments," *Future Generation Computer Systems*, vol. 15, no. 1, pp. 119–129, 1999.

[29] M. Salim, T. Uram, J. T. Childers, V. Vishwanath, and M. Papka, "Balsam: Near real-time experimental data analysis on supercomputers," in *2019 IEEE/ACM 1st Annual Workshop on Large-scale Experiment-in-the-Loop Computing (XLOOP)*. IEEE, Nov. 2019. [Online]. Available: https://doi.org/10.1109/xloop49562.2019.00010

[30] Y. Zamora, L. Ward, G. Sivaraman, I. Foster, and H. Hoffmann, "Proxima: Accelerating the integration of machine learning in atomistic simulations," in *ACM International Conference on Supercomputing*, 2021, pp. 242–253.

[31] A. Jain, S. P. Ong, W. Chen, B. Medasani, X. Qu, M. Kocher, M. Brafman, G. Petretto, G.-M. Rignanese, G. Hautier, D. Gunter, and K. A. Persson, "Fireworks: A dynamic workflow system designed for high-throughput applications," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 17, pp. 5037–5059, 2015, cPE-14-0307.R2. [Online]. Available: http://dx.doi.org/10.1002/cpe.3505

[32] S. Partee, M. Ellis, A. Rigazzi, S. Bachman, G. Marques, A. Shao, and B. Robbins, "Using machine learning at scale in hpc simulations with smartsim: An application to ocean climate modeling," 2021.

[33] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. R. Dunning, S. Legg, and K. Kavukcuoglu, "IMPALA: Scalable distributed deep-RL with importance weighted actor-learner architectures," in *Proceedings of Machine Learning Research*, vol. 80, 2018, pp. 1407–1416.

[34] E. Liang, R. Liaw, R. Nishihara, P. Moritz, R. Fox, K. Goldberg, J. Gonzalez, M. Jordan, and I. Stoica, "RLlib: Abstractions for distributed reinforcement learning," in *35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, J. Dy and A. Krause, Eds., vol. 80. PMLR, 10–15 Jul 2018, pp. 3053–3062. [Online]. Available: http://proceedings.mlr.press/v80/liang18b.html

[35] "Exarl," accessed April 9, 2021. Still private as of paper submission. [Online]. Available: https://github.com/exalearn/ExaRL

[36] D. C. Lonie and E. Zurek, "XtalOpt: An open-source evolutionary algorithm for crystal structure prediction," *Computer Physics Communications*, vol. 182, no. 2, pp. 372–387, Feb. 2011. [Online]. Available: https://doi.org/10.1016/j.cpc.2010.07.048

[37] B. Farr and W. M. Farr, "kombine: a kernel-density-based, embarrassingly parallel ensemble sampler," accessed August 30, 2021. [Online]. Available: https://github.com/bfarr/kombine

[38] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica, "Tune: A research platform for distributed model selection and training," *arXiv preprint arXiv:1807.05118*, 2018.

[39] V. Balasubramanian, S. Jha, A. Merzky, and M. Turilli, "Radical-cybertools: Middleware building blocks for scalable science," 2019.

[40] "Existing workflow systems," accessed April, 2021. [Online]. Available: https://github.com/common-workflow-language/common-workflow-language/wiki/Existing-Workflow-systems

[41] M. Turilli, A. Merzky, T. Naughton, W. Elwasif, and S. Jha, "Characterizing the performance of executing many-tasks on Summit," *arXiv preprint arXiv:1909.03057*, 2019.

[42] R. H. Castain, J. Hursey, A. Bouteiller, and D. Solt, "PMIx: Process management for exascale environments," *Parallel Computing*, vol. 79, pp. 9–29, 2018.

[43] J. M. Wozniak, M. Dorier, R. Ross, T. Shu, T. Kurc, L. Tang, N. Podhorszki, and M. Wolf, "MPI jobs within MPI jobs: A practical way of enabling task-level fault-tolerance in HPC workflows," *Future Generation Computer Systems*, 2019.

[44] L. Ward, G. Sivaraman, J. G. Pauloski, Y. Babuji, R. Chard, N. Dandu, P. C. Redfern, R. S. Assary, K. Chard, L. A. Curtiss, R. Thakur, and I. Foster, "Dataset for colmena: Scalable machine-learning-based steering of ensemble simulations for high performance computing," 2021. [Online]. Available: https://petreldata.net/mdf/detail/colmena_mlhpc21_v1.1

[45] B. Blaiszik, L. Ward, M. Schwarting, J. Gaff, R. Chard, D. Pike, K. Chard, and I. Foster, "A data ecosystem to support machine learning in materials science," *MRS Communications*, vol. 9, no. 4, p. 1125–1133, 2019.