# Toward designing effective exascale scientific computing workflows: experiences and best practices

Mark Coletti
Russell B. Davidson
Ada Sedova
colettima@ornl.gov
davidsonrb@ornl.gov
sedovaaa@ornl.gov
Oak Ridge National Laboratory
Oak Ridge, Tennessee

## ABSTRACT

Many fields within scientific computing have embraced advances in big-data analysis and machine learning, which often requires the deployment of large, distributed and complicated workflows that may combine training neural networks, performing simulations, running inference, and performing database queries and data analysis in asynchronous, parallel and pipelined execution frameworks. Such a shift has brought into focus the need for scalable, efficient workflow management solutions with reproducibility, error and provenance handling, traceability, and checkpoint-restart capabilities, among other needs. Here, we discuss challenges and best-practices for deploying exascale-generation computational science workflows on resources at the Oak Ridge Leadership Computing Facility (OLCF). We present our experiences with large-scale deployment of distributed workflows on the Summit supercomputer, including for bioinformatics and computational biophysics, materials science, and deep learning model optimization. We also present problems and solutions created by working within a Python-centric software base on traditional HPC systems, and discuss steps that will be required before the convergence of HPC, AI, and data science can be fully realized. Our results point to a wealth of exciting new possibilities for harnessing this convergence to tackle new scientific challenges.

## CCS CONCEPTS

• **Software and its engineering → Software design engineering**.

This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable,world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/ downloads/doe-public-access-plan).

ACM acknowledges that this contribution was authored or co-authored by an employee, contractor, or affiliate of the United States government. As such, the United States government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for government purposes only.

*SEA 2022, April 4-8, 2022, Virtual*

© 2022 Association for Computing Machinery.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

## KEYWORDS

Scientific computing, workflows, high performance computing, software design

**ACM Reference Format:**
Mark Coletti, Russell B. Davidson, and Ada Sedova. 2022. Toward designing effective exascale scientific computing workflows: experiences and best practices. In *Proceedings of SEA's Improving Scientific Software Conference (SEA 2022) (SEA 2022).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

Emerging scientific computing efforts that make use of leadership-class high performance computing (HPC) resources have begun to incorporate complex, heterogeneous tasks that may run asynchronously on the resources, and may require multiple different tasks that each use the resources in different ways. This shift away from large, synchronous, homogeneous computing patterns that have traditionally been deployed on leadership systems is due in part to the increasing use of artificial intelligence (AI), such as deep learning, together with traditional HPC simulations [6, 33, 54]. The use of AI together with HPC simulation promises to provide increases in both accuracy and efficiency, and has been demonstrated for climate modeling [38], molecular simulation and materials science [30, 34], fluid dynamics modeling [10] and plasma simulations [3]. In addition to these types of hybrid simulation/AI workloads, some computational methods employ complex strategies to efficiently sample parameter spaces or simulated potential energy surfaces. Examples include optimization algorithms and hyperparameter tuning [13, 14, 40] and replica-based "enhanced sampling" schemes [9] in molecular dynamics simulations [11, 25, 28, 32, 36, 49, 51].

As leadership-class scientific computing enters exascale, these large hetergeneous workloads will require efficient workflow management schemes to enable both effective deployment and the ability to address fault tolerance, and error handling and provenance analysis when running tens of thousands of potentially asynchronous tasks of different types with complex dependencies on each other. Here we describe our experiences deploying several different examples of such workflows at scale on the pre-exascale Summit supercomputer at the Oak Ridge Leadership Computing Facility (OLCF), and discuss how these experiences will translate into efficient workflows on larger systems such as the Frontier exascale

supercomputer soon to be deployed at the OLCF. However, while our experiences are derived from work targeting leadership-scale deployments, our software approach, strategies, and lessons learned are applicable much more broadly, and will be useful for deployment of workflows on cloud resources and academic clusters.

## 2 BACKGROUND

The term “workflow" can have a number of different meanings from serial tasks in a particular order, to a set of jobs that may be performed on different resources. For business applications, workflows are generally seen as software that provides a formal way to describe a small set of repetitive tasks performed in a particular order [23]. Scientific workflows, in an analogous way, can be described as a defined set of tasks that must be completed in order for a scientist to go from a given set of data or a scientific problem to a scientific product. Scientific *computing* workflows will often involve many computationally intensive, heterogeneous tasks and are often data-oriented rather than control-flow oriented [5]. Here, in the context of HPC-based scientific computing workflows, we focus on large, parallel workloads that are deployed on the same parallel computing system, potentially having some components that may be performed in consecutive steps. Generally, these workflows will use a dataflow execution model, where compute resource units are given the next task in a queue when they become available. Any workloads that are complex or large enough will often require the use of a workflow manager, or top-level program that manages parallel distributed tasks and then reduces the data into a final form. For simple sets of completely parallel tasks, home-made scripts can be used. However, as the workflow becomes more complex these scripts create impractical amounts of work and potentially can back-up shared launch-nodes and file systems. Therefore, workflow management software created to explicitly manage task dispatch, work distribution, error analysis and fault tolerance, and other needs, is developed.

### 2.1 Workflow management software

In the realm of scientific computing workflows, a number of solutions have been developed, some of which are focused on more specific needs such as simulations for materials science or climate modeling [1, 18, 19, 21, 29, 50]. Some domain-specific tools for molecular simulations in biophysics aimed at increasing the sampling of the physical phase space have also emerged [8, 41]. Several of these solutions have used database management software as the underlying engine that manages connections between tasks.

*Use of database management software for workflows.* As far back as 1998, Ailamaki and co-workers noted that a database management system (DBMS) has many essential properties of a workflow management system and with a data-object view of scientific workflows the DBMS could be used to manage parallel computational tasks [4]. Since then, several scientific workflow management programs have incorporated DBMSs into their software design. For instance, Copernicus [42] uses the Python interface with sqlite, and Fireworks [29] and Radical Pilot [35, 48] use MongoDB for task management. Efficiency trade-offs stemming from the use of DBMS in scientific workflow programs has been discussed in recent literature within the context of HPC and massively parallel distributed solutions [46], noting that the analytical capabilities that the DBMS provides, while useful for analyzing workflow output and performance, can reduce efficiency of the workflow.

### 2.2 Scientific Workflows on HPC Systems

It is now being realized that a need for HPC and leadership-scale workflow managers exists. Some recent solutions have focused on task dispatch rate for very large numbers of short tasks, and scaling to large numbers of tasks [7, 52]. The Swift/T program is such a solution, but requires the use of a special-purpose compiler to compile the workflow into the MPI framework. Reported task dispatch rate exceeded 60 thousand tasks/sec and Swift/T was able to use 64 thousand cores at greater than 90 % efficiency. However, the use of MPI reduces the ability of the workflow to be fault tolerant to the failure of workers [20].

Radical Pilot was deployed at scale on the Titan supercomputer at the OLCF for molecular dynamics simulations [36]. The MongoDB DBMS was tested for resiliency with respect to the connections required to launch over 15,000 tasks on the OLCF Summit supercomputer [37]. In later work, FireWorks was deployed at scale on OLCF Summit for structure-based drug discovery simulations for COVID-19 research [26]. While the resiliency tests succeeded on Summit, they required the launching of MongoDB on a compute node in order to access the ability to set the node `ulimit` to a large value (~64K). Launching of FireWorks was performed through an external interface with another compute cluster known as the Slate Marble system, a container orchestration system built on Kubernetes and OpenShift.[1] In this case, `ulimit` settings and connection failures caused some inefficiency and a more efficient dispatch was provided with an in-house workflow script using an in-memory DBMS.

Most recently, solutions based on pure Python with no third-party DBMS software have been deployed on HPC systems. The Parsl parallel scripting library enables the flexible creation of parallel workflows and has been shown to scale to 250000 workers across more than 8000 nodes [7]. The Dask parallel library also provides a scripting API for parallel task dispatch [43]. Dask has been used on Summit for launching large HPC workflows for evolutionary algorithms including the LEAP library [15].

### 2.3 Workflow Needs for Evolutionary Algorithms

Evolutionary algorithms (EAs) provide a scalable, efficient means for performing optimization and parameter exploration on parallel computing resources by not only exploiting the naturally parallelizable nature of EAs, but by doing so in an asynchronous manner [45]. On HPC systems (as opposed to cloud resources), a job allocation must be efficiently used without long periods of idle resources. Therefore, efficient deployment of EAs on HPC systems require some algorithmic modifications to prevent idle nodes. One such solution is the asynchronous steady-state evolutionary algorithm (ASEA), described in detail in section 3.1.1 below. More complex combinations of different EA-driven neural network training schemes can require the workflow to enable communication

[1] https://docs.olcf.ornl.gov/services_and_applications/slate/overview.html

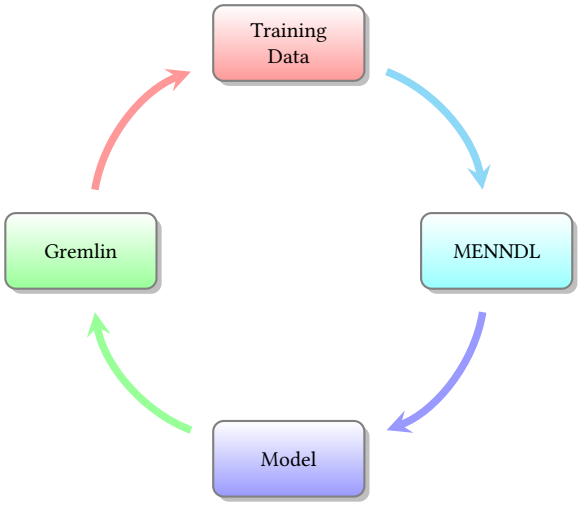


**Figure 1: This depicts the circular workflow between MENNDL and Gremlin. MENNDL learns a model from an initial set of training data. Gremlin finds feature sets where the model performs poorly. The training data is updated to include more examples that correspond the feature sets. The cycle then continues with MENNDL resuming training from**

between several different programs that each run on a large resource set.

One example of a complex High Performance Computing (HPC) workflow between two ML components is shown in Fig. 1, and shows how gaps in training data can be filled in by using an adversarial EA to find those gaps. There, the EA Multi-node Evolutionary Neural Networks for Deep Learning (MENNDL) trains a deep learner (DL) model using the given training data; it will also perform hyperparameter and architecture optimization by evaluating thousands of different DL model configurations in parallel on Oak Ridge National Laboratory (ORNL)'s Summit supercomputer [39, 40, 53]. Then Gremlin takes the best model from MENNDL, which finds a set of features where the model performs poorly. This information is exploited to adjust the training data to add more examples corresponding to those poor performing feature sets. The cycle continues when MENNDL then resumes training with the updated training set. The challenge here is that these two EAs can be run consecutively, but it may be more efficient to have them running simultaneously with a work-flow tool intelligently managing their respective resources and ensuring that the two EAs work with one another effectively. Implementing such a complex HPC workflow on a system like Summit, and its successor Frontier, remains a challenge.

### 2.4 Workflow Needs in Computational Biosciences

Biological molecular processes are some of the most complex to model, as they are highly non-linear, multiscale, and often modeled non-deterministically. Exponential growth in the number of known sequenced genomes, driven by decades of advances in gene sequencing technology, has created a widening technology gap: that of extracting evolutionary information and biological function from these sequences. In order to break through the data processing barrier in bioscience research, we require a multi-pronged computing strategy that can deal with heterogeneous problem sizes of computational molecular evolution, protein structure prediction, biomolecular simulation and structural analysis. Simulation tools, methods to analyze the flood of sequence and structure data, and the recent incorporation of AI have enabled advances in our ability to design and predict biomolecular systems for medicine, bioenergy and biosecurity. HPC can provide a way to help bridge some of the technology gaps caused by the data explosion. HPC workflows in biology can use a combination of tools such as deep learning based protein structure prediction, sequence alignment, and modeling/simulation, deployed at the genome scale [24].

### 2.5 Workflow Needs in Computational Chemistry and Materials Science

A recent explosion of data and AI driven methods in chemistry and materials science [2] has opened the door for the combination of HPC simulation, data science and machine learning [16, 54]. Recent work in this direction by our team has involved the deployment of programs for training neural network models of physical potentials for running fast molecular simulations with higher accuracy on larger systems. This creates a requirement for deploying complex workflows that launch training and simulation in parallel in order to create an optimized model that represents as much of the complete physical phase space as possible.

### 2.6 Software Considerations for HPC Workflows

In the following, we will describe some of the programming considerations, software design strategies and optimizations we developed to enable the creation and deployment of workflows on HPC systems for a number of different efforts, and work-arounds for problems we encountered, using the Dask parallel library. We discuss steps we took to deploy these workflows at large scale on the Summit supercomputer, and envision potential strategies for even larger deployments on upcoming exascale systems based on our experiences.

## 3 USING DASK TO SUPPORT LARGE SCIENTIFIC WORKFLOWS

Dask is a popular python package for distributed processing that scales from laptops to supercomputers [17]. In this section we describe two Dask-supported scientific workflows that have been demonstrated to scale on HPC platforms. We also relate difficulties encountered while scaling Dask along with associated workarounds and solutions.

### 3.1 Two Dask-supported HPC Scientific Workflows

Here we share two HPC workflows that we have implemented using Dask. The first workflow describes is where new tasks are added during a run – e.g., for an evolutionary algorithm where new individuals have to be added to the task queue for evaluation. The

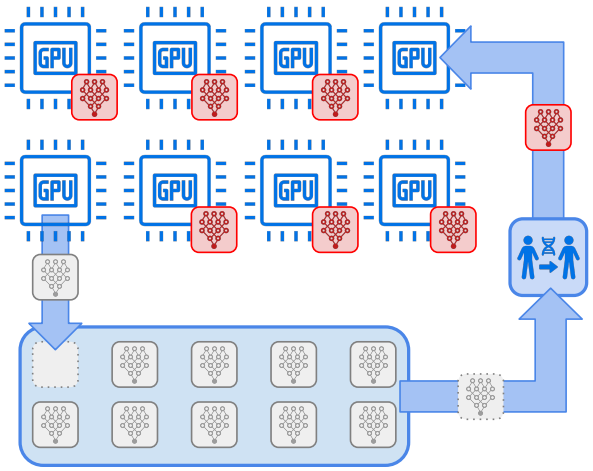


**Figure 2: This figure shows an example of an HPC workflow where new tasks are added during the run — in this instance this is for the ASEA workflow. The rounded box is a population of evaluated individuals, where an individual represents a specific set of features for a problem of interest. The GPU icons represent Dask workers that evaluate offspring. Grey individuals have been evaluated where red are undergoing evaluation. The gray dotted outline individual represents a parent that was selected with replacement to be used to create a new offspring by being cloned and then mutating that clone.**

second workflow style entails processing a single, large batch of data for a fixed number of pre-defined tasks, and which was used to manage biomolecular workloads running structure prediction with AlphaFold2 [31] and molecular dynamics simulation with OpenMM [22] on Summit.

*3.1.1 Dynamic workflows with Dask.* Figure 2 depicts the workflow for an ASEA, which is an example of a workflow where new tasks are dynamically added while the workflow progresses. In this workflow, a single population, shown in the rounded box, is updated during a run. Individuals are evaluated on Summit nodes, represented by the GPU icons. When an individual is finished evaluation, it is compared with a randomly selected individual in the current population; the best of the two gets to stay in the population. The HPC resource used to evaluate that individual is now idle, so a random individual is selected from the population, cloned, and that cloned mutated to create a new, unevaluated offspring, which is then assigned to that resource to get it active again.

The code in Listing 1 is an implementation for the ASEA workflow shown in Fig. 2. In lines 1–2 an initial random population is created and then distributed to all the Dask workers via the `client.map()` call, where `eval_ind()` is a function that evaluates a single individual, and `init_pop` is a list of the initially generated random individuals. `client.map()` returns a list of `futures` that is used to get a Dask iterator in line 3; this iterator is used in the for loop in the next line to cycle through each individual as a Dask

```
1  init_pop = create_initial_pop()
2  futures = client.map(eval_ind, init_pop)
3  as_compl_iter = as_completed(futures)
4  for finished_task in as_compl_iter:
5      result = finished_task.result()
6      print(f'Evaluated individual: {result}')
7      update_population(population, result)
8      offspring = create_offspring(population)
9      future = client.submit(eval_ind,
10                            offspring)
11     as_compl_iter.add(future)
```

**Listing 1: This python code shows how Dask can be used to implement the ASEA workflow shown in Fig. 2.**

worker completes an evaluation. Normally a Dask user would call `Client.gather()` to wait on all the futures, but we want to capture each individual as it is evaluated so that we can consider inserting it into the population, and then we want to create and assign a new offspring to its Dask worker so that it is busy again. Lines 5–6 updates the population with the new individual, either by replacing a current individual in the population with the new individual or discarding the new individual if its fitness is inferior. Line 8 a new individual, or offspring, is created, usually by randomly selecting an individual with replacement from the population, then that individual, or parent, is cloned, and then mutated, thereby creating an entirely new individual. This individual is then assigned to the previously idle Dask worker associated with `finished_task` so that it resumes work; this is done by calling `client.submit()` with the new offspring and the previously used `eval_ind()` function. However, of particular note is that the returned future is then used in line 11 to notify the iterator, `as_compl_iter`, of the new task. This is why we captured that iterator in line 3 so that we could later call its `add()` member function. (For those that use python 3.8, or greater, you can use the new walrus operator to assign and use the iterator variable in the `for` loop declaration. [2])

This workflow is better suited for HPC environments than the more traditional by-generation EAs. That is, progress to the next generation with the by-generation approach cannot proceed until all the new individuals of that generation have been evaluated; this is problematic if some individuals finish processing early, which means their corresponding HPC resources will idle until the next generation. By contrast, with the ASEA approach when an individual is finished processing a new offspring is immediately generated and assigned to that resource, thus minimizing idle time [45]. This workflow has been successfully scaled to run on the Summit supercomputer [14].

*3.1.2 Static workflows with Dask.* The workflow described in §3.1.1 took a great deal of trial and error to get fully functional on ORNL's Summit supercomputer, primarily because of its Power9 architecture, but also because Dask required some tuning to get it to scale to 1,000 nodes, which are described in more detail in §3.2.2. However, evidence of Dask's flexibility came when a need arose to scale Alphafold2 to run on Summit in that existing code for the ASEA implementation was repurposed to support AlphaFold2 in only a

[2] https://docs.python.org/3/whatsnew/3.8.html

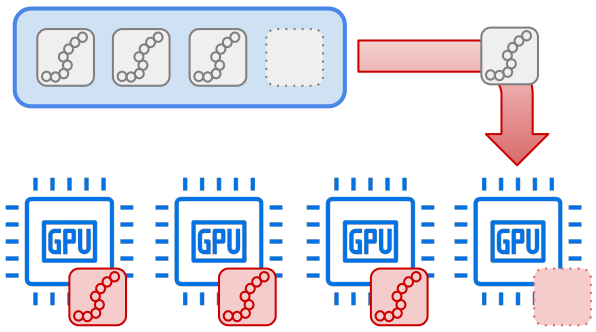


**Figure 3: This diagram depicts a common workflow for processing biophysics data, in this case proteins, which are represented by the icons of strands of circles. The rounded box represents a queue of proteins awaiting evaluation, and the GPU icons correspond to Dask workers processing a protein. When a worker finishes processing, the next protein is popped from the queue and assigned to it to begin processing.**

day and a half. Moreover, this proved to be a template of sorts in that this code was repurposed again to support massively parallel OpenMM runs on Summit. We describe this simpler, but useful, workflow here.

Figure 3 shows this workfow, that is, one for a *static* workflow instead of the dynamic workflow described earlier. That is, we would know ahead of time the entirety of data that needs processed, and no new data to be processed will arise during a run's course. Here, a simple flat file of proteins is read, then they are fanned out to Dask workers for processing — in this case, AlphaFold2 was called as as Python subprocess to work on a given protein. When a task was completed, the next protein was popped from the Dask scheduler's queue, and assigned to the completed tasks's Dask worker, thus ensuring it remained busy. This process would continue until the queue was empty.

```
1 all_items = read_items()
2 futures = client.map(process_items, all_items)
3 for finished_task in as_completed(futures):
4     result = finished_task.result()
5     print(f'Finished processing: {result}')
```

**Listing 2: Example code demonstrating how to implement the workflow shown in Fig. 3.**

Listing 2 shows corresponding python code implementing this workflow, and which is much simpler than the code shown in Listing 1. Here, in line 1, `read_items()` will read the flat file of items to be processed. (In the case of AlphaFold2, this was a list of proteins.) Then, as before, these items are fanned out to available Dask workers via `client.submit()` where `process_items()` will compute on each item in parallel. If the number of items exceeds the number of queues, then items not yet processed are held in the Dask scheduler's queue. Items are popped from that queue and assigned to a Dask worker for processing as a worker completes a task.

We could have used `client.gather()` instead of the `for` loop, but the loop allows for tracking completed tasks. (Alternatively we could have implemented a Dask worker plugin that would monitor completed tasks[3], but this approach is simpler and easier to understand.)

This workflow has also proven to be flexible enough to be readily applicable to other, similar problems. For example, it has successfully be reapplied to allowing for massively parallel processing of molecular dynamics models using OpenMM. We anticipate using this as a form of template for other problems that have similar computational needs.

## 3.2 Encountered Dask issues and their solutions

While scaling our Dask applications to work on Summit, we would occasionally encounter problems. Here, we describe some of those problems and their respective solutions or work-arounds.

*3.2.1 Invoking Dask with class member functions.* As seen earlier, submitting tasks to Dask for work involves invoking either `client.submit()` or `client.map()` with a single function and an item or items on which to apply that function per Dask worker. However, it may be the case that from time to time one will need to pass in a *class member function* to those functions, instead. This was the case for our EA that used the Library for Evolutionary Algorithms in Python (LEAP) toolkit because one requirement is implement a class that has a member function, `evaluate()` that is, in turn, directly called by Dask workers.

```
1  class ExampleLEAPProblem(ScalarProblem):
2      def __init__(self, model_fpath: str):
3          super().__init__()
4          # Load the model here so we don't
       reload for each evaluate(),
5          # but this is actually bad.
6          self.model = Model()
7          self.model.load(model_fpath)
8
9      def evaluate(self, phenome) -> int:
10         """
11         Evaluate the phenome with the given
       model.
12         :param phenome: is named tuple
       describing state
13         :returns: score for model performance
       for this state
14         """
15         fitness = self.model.predict(list(
       phenome))
16         return fitness
```

**Listing 3: Example code showing a LEAP Problem class.**

Listing 3 shows an example of a LEAP `Problem` class where a model must be run with a given set of features per Dask worker.

[3] https://distributed.dask.org/en/stable/plugins.html#worker-plugins

That is, the task a Dask worker gets is to accept a set of features, here the phenome, and then calls the associated `evaluate()` with that argument. Models are generally large, so this version of the class tries to mitigate the load times by loading the model once in the constructor, and then just referring to the model in the function call.

However, though this is what normally happens, things are a little different using Dask. In this case, when the Dask worker gets a new phenome, it actually copies over the entire object to invoke it, which means the model comes along for every single invocation. And since this sometimes means being transmitted over the network, this could greatly impair performance. Fortunately, Dask is good at warning about this scenario when it detects that an onerous amount of data is being sent to workers for each new task.

One solution that works is to move the model creation and load, which corresponding to lines 6–7, into `evaluate()`. This is not satisfying because now the model is created and loaded for every evaluation, so network load has been exchanged for additional computing burden.

```
1     def evaluate(self, phenome) -> int:
2         worker = get_worker()
3         fitness = worker.model.predict(list(
    phenome))
4         return fitness
```

**Listing 4: Updated evaluate after use of client.scatter().**

Another solution would be to use `client.scatter()`. That is, on the Dask client the model is created and loaded, and then `client.scatter()` is used to transmit that model to the workers so that it is stored locally. Then `evaluate()` can refer to the model stored in the worker, thus saving network and compute resources. The updated `evaluate()` that assumes the Dask client has done this is shown in Listing 4. Now the model is indeed loaded once in each worker, but as the `scatter()` documentation warns [4], this is still not the ideal approach because the model still has to be transmitted to all the Dask workers, though admittedly this only occurs once.

```
1  def setup_worker(model_path=None):
2      """ setup worker to run model """
3      worker = get_worker()
4      worker.model = Model()
5      worker.model.load(model_path)
6
7  with Client() as client:
8      client.run(setup_worker,
9                 model_path='some/model.pkl',
10                wait=True)
11     # ... regular Dask tasking follows
```

**Listing 5: Using run to have Dask workers load models at start of run.**

Listing 5 shows the ideal solution. There, a helper function, `setup_worker()`, will instruct a given Dask worker to create and load the model. Then, later, the `client.run()` invokes that function for all the workers, and which is done before the actual desired tasking starts. Make particular note of line 10, `wait=True`, because by default `client.run()` works asynchronously such that it may be the case that a worker has not yet loaded a model before it gets tasking that expects its model to be loaded and ready. Using that `wait=True` means that the `run()` will not proceed until *all* the workers have loaded their respective models.

*3.2.2 Tuning Dask configuration files.* When an HPC application that uses Dask is run for the first time, it creates two YAML-formatted configuration files in your home directory. These are:

**`~/.config/dask/dask.yaml`** General Dask configuration
**`~/.config/dask/distributed.yaml`** Distributed Dask configuration

Most of the values in these files will be commented out, which is convenient to see what variables exist, and maybe a hint as to their function. Adjusting some of the configuration variables found in these files becomes critical on very large HPC platforms like Summit. We will give suggestions for some variables to set and why, but we strongly encourage performing your own sensitivity studies since these settings will be tightly coupled to the systems on which Dask tasks are run. As an aside, these configuration variables are documented on the Dask web site[5], and we strongly encourage reviewing them before trying to adjust these settings.

`~/.config/dask/dask.yaml` is the smallest of the configuration files, and the only option of interest to us is `temporary-directory`. By default, Dask will create scratch directories in the current directory from which the executable processed is launched. On a system like Summit, this usually means the high-speed GPFS filesystem. However, Dask I/O is better served by, instead, writing to `/tmp` since, in the case of Summit, that is on the same node as the Dask worker. This may be similar to your HPC setup for running jobs. A viable alternative is to use a very fast solid-state drive that is local to workers, if such is available on your systems. For Summit, the burst-buffer is just such a device, with the handy side-effect of automatically cleaning up the Dask scratch directories upon job completion.

However, most of your attention will be on tuning parameters found in `~/.config/dask/distributed.yaml`. In that file most of the attention should be given to variables that control timeouts, heartbeats, and time-to-live (TTL). The timeouts are fairly self-descriptive, but the heartbeats bear some discussion. First, the client will periodically send a "heartbeat" to the Dask scheduler; if the scheduler has not heard from the client in a span of time, then it will assume the client is gone and unregister that client. This span of time is controlled by `distributed.client.heartbeat` and defaults to 5 seconds, and for Summit we increased that to 30 seconds. However, it is not the client heartbeat that will be the cause of the most potential woe, but rather a similar heartbeat for the Dask workers. Just as with the client if the scheduler does not get a heartbeat from a worker within a span, it will assume the worker is dead, and so will unregister it and reassign its task to another worker. The variable controlling this is `distributed.scheduler.worker-ttl`,

[4] https://distributed.dask.org/en/stable/api.html#distributed.Client.scatter

[5] https://docs.dask.org/en/latest/configuration.html#configuration-reference

which is a little counterintuitive since naturally you would be looking for `worker-heartbeat`, which does not exist. In any case, when running jobs with hundreds or even thousands of workers, the number of heartbeats the scheduler has to "listen" to can be overwhelming, so dialing up this variable may be very helpful, particularly if it is observed during sensitivity runs that workers are getting unregistered when they are actually still active. (However, if your Dask workers appear to be mysteriously dying, we suggest looking at this site for guidance: https://distributed.dask.org/en/stable/killed.html) It also may take a while for connections to be made especially if a lot of workers, say thousands, are coming online. So other variables to address are `distributed.comm.timeouts.connect` and `distributed.comm.timeouts.tcp`, which both default to 30 seconds. For Summit, we have had to greatly increase those to 1800 seconds for large numbers of workers, but those values will differ for other systems, of course. We also recommend ensuring that worker stealing is on by setting `distributed.scheduler.work-stealing` to `True`; there is more discussion on worker stealing in the next section.

## 4 PERFORMANCE TESTING OF DASK WORKFLOWS AT SCALE

There are a number of parameters that can be tuned to enable Dask to both run successfully on large numbers of nodes, and to use these resources efficiently. An important functionality within Dask is the ability for workers to steal tasks from the pre-determined task lists that are created on initialization and assigned, and may be sub-optimal.[6] With a dataflow execution model, if tasks have a variability in runtime, this can help significantly with reducing worker idle time and filling all available compute resources with work, especially if the initial task list creates an unbalanced assignment. Exact details about initial task assignments and work stealing policies are decided within settings connected with the scheduler, and may be difficult to tune and understand exactly. Some of the factors that Dask uses to determine how work stealing is performed is based on metrics that are used to determine runtimes of a particular function, and detectable locations of the data the workers will require to perform their task. Dask uses a moving average over function runtimes to obtain an approximation for how long a task is expected to run. If a particular function can have highly variable runtime based on input type, it is possible that this method will provide an inaccurate estimate. In addition, Dask tries to understand the communication requirements of distributed calculations and data locality, for treating partitioned matrix calculations and other distributed tasks with dependencies and data-sharing and communication requirements; for completely parallel tasks, these calculations may not be helpful for performance. In addition to the Dask decisions about task dispatch, it is possible that user-modifications to the list passed to Dask could also influence performance. To test these factors, we create a list of simple tasks with a large variation in time to solution, by specifically including a waiting step of a variable number of seconds drawn randomly from a distribution. We used this task list to query the final task order that Dask dispatched, and how much idle time resulted. We also tested the effects of turning the work stealing function on and off, and how a task list pre-sorted by runtime affects the Dask workflow that is dispatched.

Figure 4 shows the results of running the test workflow with work stealing turned on and on an input list of tasks that was sorted by runtime size, and one that was not. In many cases an approximate runtime can be predicted if a calculation scales as some function of the input's characteristics. For protein structure prediction, for instance, runtime is dependent on sequence length. Figure 4 A shows the distribution that was drawn from to create the task runtimes for the test workload. B shows the distributions of idle times for each worker when using the sorted (green) and unsorted (blue) input list. Here, the sorted input list added to the overall efficiency by reducing the worker idle times. C and D show the worker tasks that were run for each set-up; the first set of tasks elucidates the initial task list that is created by Dask. In the case where wok stealing is turned on and sorting is applied, it seems the initial task list distributes in a round-Robbin manner, assigning a task from the list to each worker before returning to add a new task to the first worker's list. Thus, the larger tasks are assigned to all workers first. Following this, some smaller tasks seem to have made their way in between two consecutive large tasks, and it would appear that some rearrangement of the task list must have occurred. For the unsorted input, the initial tasks appear to also be random, and for some workers during the workflow run, several consecutive large tasks were also assigned to workers.

Figure 5 shows the measured idle time for workers, for both the sorted and unsorted input lists, when work stealing is turned off. Behavior is very different this time, with the sorted input leading to substantially more idle time for the workers. An inspection of the two initial task assignments showed that in the sorted situation, tasks were not distributed in a round-Robin manner from the list, but rather the first, largest tasks were assigned to the first 45 workers. The last workers received only small tasks, and without the work stealing option turned on, no optimization of the queue was performed and many workers remained idle after completion of all of their tasks, while the first workers remained occupied with large jobs. On a system like Summit, this inefficiency is also expensive, in that there is no elasticity in the batch job scheduling system and a large job request will continue to use all requested nodes even when these nodes may be idle.

While these tests have demonstrated default Dask behavior, there are a number of parameters that can be set to control how the scheduler makes task lists, decides on work stealing procedures, and how work is prioritized when there are more tasks than workers. It is clear that performance and efficiency relies heavily on all of these setting and the decisions that Dask is making about how to distribute and execute work in the workflow. We see that in this case, it is more efficient to use the work stealing features even when tasks have no dependencies, no communication needs, and almost no data to be moved.

### 4.1 Deployment at Scale for Molecular Dynamics Simulations

Atomic-resolution structural models of biomolecules, such as proteins, nucleic acids, and membranes, can provide key insights into the system's function and relevance in biochemical processes. Yet,

[6] https://distributed.dask.org/en/stable/work-stealing.html

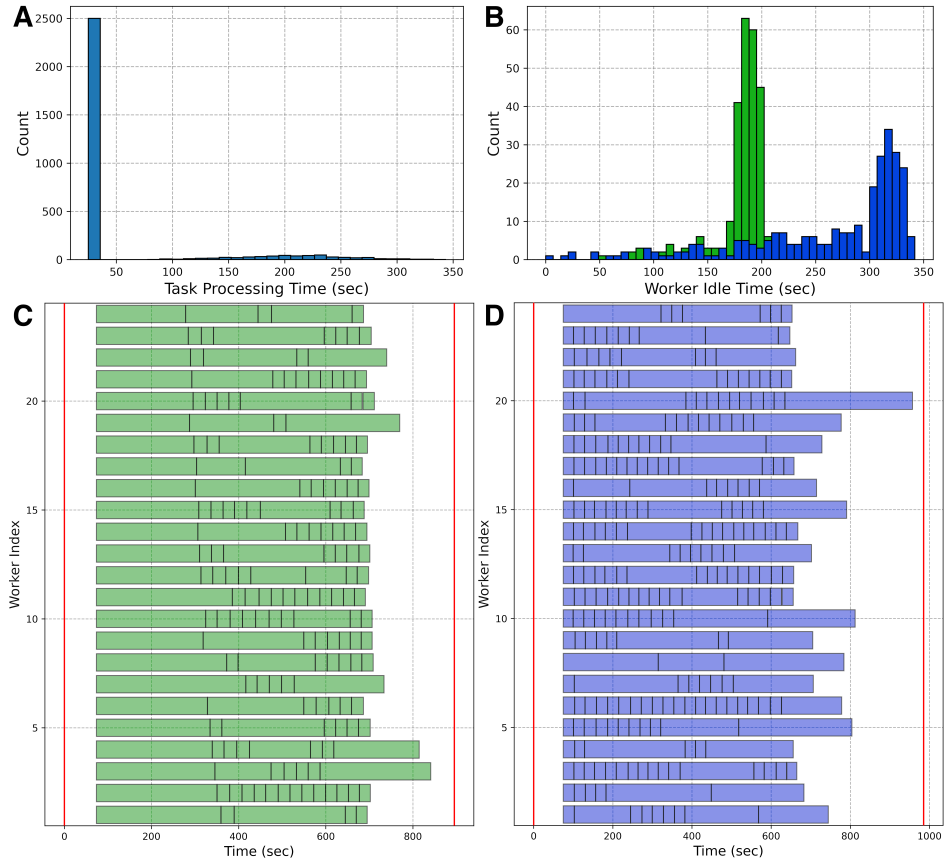


**Figure 4: Demonstration of task sorting. (A) Distribution of task processing times. A static set of 3000 tasks were performed to benchmark the behavior of distributing heterogeneous tasks evenly across all available workers. (B) Distribution of time where workers are idle after completing their allotted tasks for the sorted and unsorted tasks lists, shown in green and blue, respectively. (C) and (D) Worker schedules for the sorted and unsorted runs. Pre-sorting tasks based on their expected computational expense can aid the efficiency of the overall workflow. Work-stealing was active during this set of tests.**

these molecules are inherently dynamic with complex free energy surfaces that describe the system's conformational states. Molecular dynamics (MD) is an efficient method to sample the conformational states of biomolecules by numerically propagating Newton's equations of motion for atomic systems. The calculations performed during MD simulations are prime for GPU-acceleration and parallel computing methods on HPC resources. Furthermore, the Dask workflows presented here have been used to run massive amounts of MD simulations, making traditionally-expensive protocols like Replica Exchange MD, Adaptive Sampling, and other free energy calculations much more feasible.

To demonstrate a basic Dask workflow's capability, a series of MD simulations are run as tasks. The OpenMM [22] simulation engine was used for running NVT simulations of the SARS-CoV-2 Mpro enzyme with a bound ligand and a solvent box of TIP3P water molecules [44]. The Amber ff19SB[47] and GAFF force fields were used as parameter sets for the protein and ligand, respectively. The Langevin dynamics thermostat was used to maintain a 310 K temperature. Particle mesh Ewald (PME) was used to approximate long range interactions with an explicit nonbonding interaction cutoff of 12 Å. The integration time step was 2 fs. Trajectory frames were written every 50,000 steps with a total of 250,000 steps performed (0.5 ns of simulation per task). Each MD simulation is given 1 GPU and 1 CPU core as resources.

The Dask pipeline was tested on 1 to 1000 Summit nodes, with 6 workers per node. The number of tasks were scaled by the number of workers, 3 per worker. The client waited for all workers to spin up before tasks began processing. Figure 6 shows the worker schedule of the MD workflow running on 45 nodes (270 workers). Red lines indicate the start and end of the Dask workflow while green bars indicate times during which a worker is running a MD simulation. The empty slots represent overhead times during startup of the Dask scheduler, Workers, and Client as well as the shutdown overhead as tasks finish and communications are stopped. Minimal overhead is seen between tasks as they are handed off to workers. Table 1 shows the average and standard deviation of these overhead times for the MD Dask workflow across a range of nodes. Generally, all overhead times increase as the number of nodes increases although this behavior is not linear. As more

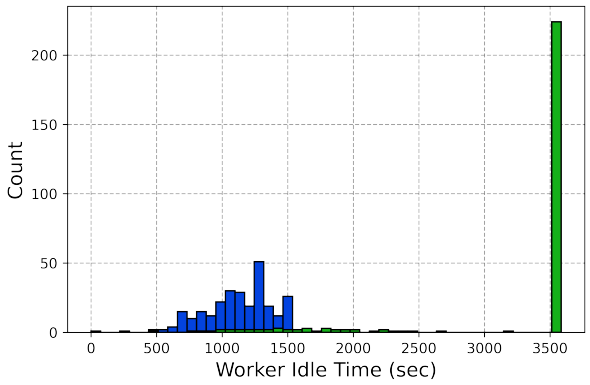


**Figure 5: Distribution of worker idle times for Dask workflows where work stealing is set to false (work-stealing : False) in the configuration file. The same set of tasks were run as shown in 4A. The same sorting is performed on the tasks shown in green, resulting in a small number of workers receiving an inordinate number of time consuming tasks.**

workers are provided to perform more tasks, the Dask Scheduler requires more CPU cores and communications between the Scheduler, Workers, and Client become more time consuming. For 1000 nodes, the `dask-scheduler` command was provided 20 cores rather than the 10 provided for the other runs of the workflow.

In terms of MD efficiency, the Mpro trajectories were obtaining $189.5 \pm 0.5$ ns day$^{-1}$. Since each task is performed independently, this simulation efficiency is not affected by scaling to large number of nodes. Therefore, each node can perform six trajectories of ≈190 ns in a day. Once scaled to hundreds or thousands of nodes, this Dask workflow can easily obtain hundreds of microseconds and up to milliseconds of MD simulation of biomolecular systems within a single day in the form of independent MD simulations, which overcomes potential pitfalls of an equivalently long single MD simulation. This amount of sampling is often required for experimentally-accurate quantitative analysis of a biomolecule's conformational free energy landscape and/or the system's kinetics [12, 27].

**Table 1: Worker overhead times in seconds.**

| Nodes (Workers) | Start | End | Between Task |
|---|---|---|---|
| 1 (6) | 62.6109 ± 0.0003 | 8. ± 4 | 1.6458 ± 0.0002 |
| 2 (12) | 64.2754 ± 0.0004 | 3. ± 4 | 0.6018 ± 0.0002 |
| 45 (270) | 78.763 ± 0.008 | 8. ± 2 | 3.748 ± 0.008 |
| 500 (3000) | 91.49 ± 0.09 | 100 ± 10 | 31.6 ± 0.2 |
| 1000 (6000)† | 203.4 ± 0.2 | 30 ± 20 | 27.9 ± 0.5 |

† CPUs provided to the Scheduler were increased to 20 cores to handle the increased communication overhead.

Code is available to recreate this Dask workflow at https://github.com/BSDExabio/OpenMM-on-Summit

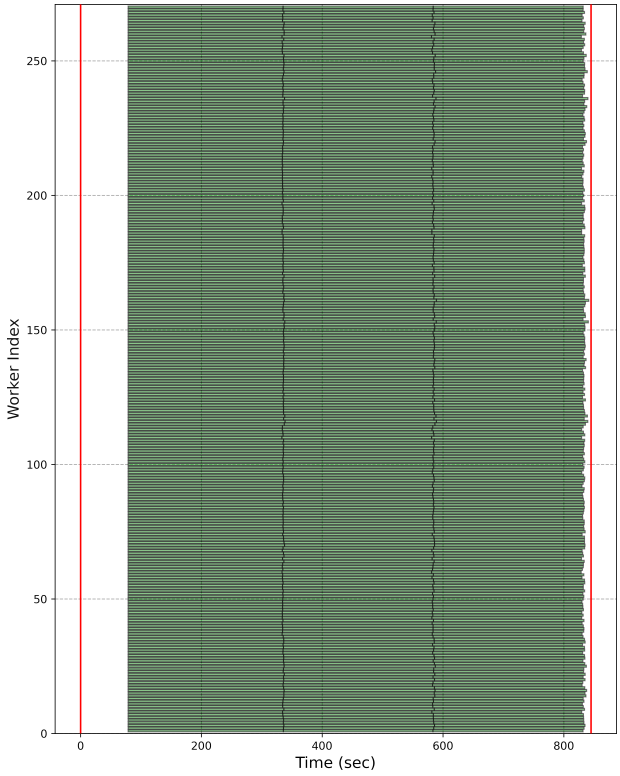


**Figure 6: Walltime plot of Dask workers processing MD simulation tasks on 45 Summit nodes. Red lines indicate the start and end times for the submission script that spins up the Dask scheduler, workers, and client. Tasks handled by the workers are run using a subprocess call to a separate python script. Overhead time can be seen at the beginning and end of the job, with negligible overhead in between tasks.**

## 5 DISCUSSION AND CONCLUSIONS

We have shown that the Dask parallel Python library can be efficiently used to launch large workflows on HPC systems both statically, where all tasks are defined completely at the launch of the workflow, and dynamically, where idle workers are assigned a new task that is created on-the-fly based on the current state of the workflow. The latter is a strategy for optimization algorithm deployment such as in the use of asynchronous steady-state evolutionary algorithms. There are many configuration parameters and run options that can be set or adjusted to enable better performance and scaling to larger numbers of compute nodes.

We also shared Dask-related problems related to using class member functions when managing onerous common resources, such as using a model to make predictions within a Dask task. We also related the necessity of tuning certain Dask configurations for timeouts, heartbeats, as well as specifying where Dask temporary files are written to address potential performance issues. Along the way we pointed the reader to supporting Dask documentation that may otherwise have been easily missed.

We found that enabling work stealing was important to performance when all other worker task assignment related parameters were used in their default settings, using benchmark tests with variable runtimes. Timeout settings and numbers of compute cores provided to the `dask-scheduler` helped prevent workflow failures for higher node counts. We demonstrated a large deployment of molecular dynamics simulations on OLCF Summit with 1000 nodes. While overhead times increased about 10 times when going from 45 to 1000 nodes, the total overhead is a small fraction of the total runtime for these simulations when used in production, and therefore is not a concern. Overall, it seems that using a Python parallel framework with no third-party database management software is an effective solution with a simpler installation for large-scale deployments on HPC systems. In conclusion, we have found that reusable, scalable workflows can be deployed with a similar, lightweight Python framework for many different types of scientific computing efforts across many disciplines, and this approach can provide a strategy to manage the complex, heterogeneous workloads that are emerging as HPC simulation converges with machine learning.

## ACKNOWLEDGMENTS

This research was sponsored by the Laboratory Directed Research and Development Program at Oak Ridge National Laboratory (ORNL), which is managed by UT-Battelle, LLC, for the U.S. Department of Energy (DOE) under Contract No. DE-AC05-00OR22725, and used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725. We thank .

## REFERENCES

[1] Carl S Adorf, Paul M Dodd, Vyas Ramasubramani, and Sharon C Glotzer. 2018. Simple data and workflow management with the signac framework. *Computational Materials Science* 146 (2018), 220–229.

[2] Ankit Agrawal and Alok Choudhary. 2016. Perspective: Materials informatics and big data: Realization of the"fourth paradigm" of science in materials science. *Apl Materials* 4, 5 (2016), 053208.

[3] Xavier Aguilar and Stefano Markidis. 2021. A Deep Learning-Based Particle-in-Cell Method for Plasma Simulations. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 692–697.

[4] Anastassia Ailamaki, Yannis E Ioannidis, and Miron Livny. 1998. Scientific workflow management by database management. In *Proceedings. Tenth International Conference on Scientific and Statistical Database Management (Cat. No. 98TB100243)*. IEEE, 190–199.

[5] Ilkay Altintas, Chad Berkley, Efrat Jaeger, Matthew Jones, Bertram Ludascher, and Steve Mock. 2004. Kepler: an extensible system for design and execution of scientific workflows. In *Proceedings. 16th International Conference on Scientific and Statistical Database Management, 2004.* IEEE, 423–424.

[6] Rick Archibald, Edmond Chow, Eduardo DâĂŹAzevedo, Jack Dongarra, Markus Eisenbach, Rocco Febbo, Florent Lopez, Daniel Nichols, Stanimire Tomov, Kwai Wong, et al. 2020. Integrating deep learning in domain sciences at exascale. In *Smoky Mountains Computational Sciences and Engineering Conference*. Springer, 35–50.

[7] Yadu Babuji, Anna Woodard, Zhuozhao Li, Daniel S Katz, Ben Clifford, Rohan Kumar, Lukasz Lacinski, Ryan Chard, Justin M Wozniak, Ian Foster, et al. 2019. Parsl: Pervasive parallel programming in python. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*. 25–36.

[8] Vivekanandan Balasubramanian, Iain Bethune, Ardita Shkurti, Elena Breitmoser, Eugen Hruska, Cecilia Clementi, Charles Laughton, and Shantenu Jha. 2016. Extasy: Scalable and flexible coupling of MD simulations and advanced sampling techniques. In *2016 IEEE 12th International Conference on e-Science*. IEEE, 361–370.

[9] Rafael C Bernardi, Marcelo CR Melo, and Klaus Schulten. 2015. Enhanced sampling techniques in molecular dynamics simulations of biological systems. *Biochimica et Biophysica Acta (BBA)-General Subjects* 1850, 5 (2015), 872–877.

[10] Mathis Bode, Michael Gauding, Konstantin Kleinheinz, and Heinz Pitsch. 2019. Deep learning at scale for subgrid modeling in turbulent flows: regression and reconstruction. In *International Conference on High Performance Computing*. Springer, 541–560.

[11] Nicolae-Viorel Buchete and Gerhard Hummer. 2008. Peptide folding kinetics from replica exchange molecular dynamics. *Physical Review E* 77, 3 (2008), 030902.

[12] John D Chodera and Frank Noé. 2014. Markov state models of biomolecular conformational dynamics. *Current Opinion in Structural Biology* 25 (apr 2014), 135–44. https://doi.org/10.1016/j.sbi.2014.04.002

[13] Mark A Coletti, Shang Gao, Spencer Paulissen, Nicholas Quentin Haas, and Robert Patton. 2021. Diagnosing autonomous vehicle driving criteria with an adversarial evolutionary algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. 301–302.

[14] Mark A. Coletti, Shang Gao, Spencer Paulissen, Nicholas Quentin Haas, and Robert Patton. 2021. Diagnosing Autonomous Vehicle Driving Criteria with an Adversarial Evolutionary Algorithm. In *Proceedings of the 2021 Genetic and Evolutionary Computation Conference Companion* (Lille, France) *(GECCO '21)*. Association for Computing Machinery, New York, NY, USA, 301–302. https://doi.org/10.1145/3449726.3459573

[15] Mark A. Coletti, Eric O. Scott, and Jeffrey K. Bassett. 2020. Library for Evolutionary Algorithms in Python (LEAP). In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion* (Cancún, Mexico) *(GECCO '20)*. Association for Computing Machinery, New York, NY, USA, 1571–1579. https://doi.org/10.1145/3377929.3398147

[16] Juan-Pablo Correa-Baena, Kedar Hippalgaonkar, Jeroen van Duren, Shaffiq Jaffer, Vijay R Chandrasekhar, Vladan Stevanovic, Cyrus Wadia, Supratik Guha, and Tonio Buonassisi. 2018. Accelerating materials development via automation, machine learning, and high-performance computing. *Joule* 2, 8 (2018), 1410–1420.

[17] Dask Development Team. 2016. *Dask: Library for dynamic task scheduling*. https://dask.org

[18] Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip J Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira Da Silva, Miron Livny, et al. 2015. Pegasus, a workflow management system for science automation. *Future Generation Computer Systems* 46 (2015), 17–35.

[19] Ewa Deelman, Karan Vahi, Mats Rynge, Gideon Juve, Rajiv Mayani, and Rafael Ferreira da Silva. 2016. Pegasus in the cloud: Science automation through workflow technologies. *IEEE Internet Computing* 20, 1 (2016), 70–76.

[20] Matthieu Dorier, Justin M Wozniak, and Robert Ross. 2017. Supporting task-level fault-tolerance in HPC workflows by launching MPI jobs inside MPI jobs. In *Proceedings of the 12th Workshop on Workflows in Support of Large-Scale Science*. ACM, 5.

[21] Lei Dou, Daniel Zinn, Timothy McPhillips, Sven Köhler, Sean Riddle, Shawn Bowers, and Bertram Ludäscher. 2011. Scientific workflow design 2.0: Demonstrating streaming data collections in Kepler. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 1296–1299.

[22] Peter Eastman, Jason Swails, John D Chodera, Robert T McGibbon, Yutong Zhao, Kyle A Beauchamp, Lee-Ping Wang, Andrew C Simmonett, Matthew P Harrigan, Chaya D Stern, et al. 2017. OpenMM 7: Rapid development of high performance algorithms for molecular dynamics. *PLoS computational biology* 13, 7 (2017), e1005659.

[23] Borko Furht (Ed.). 2008. *Workflow Computing*. Springer US, Boston, MA, 991–992. https://doi.org/10.1007/978-0-387-78414-4_266

[24] Mu Gao, Peik Lund-Andersen, Alex Morehead, Sajid Mahmud, Chen Chen, Xiao Chen, Nabin Giri, Raj S. Roy, Farhan Quadir, T. Chad Effler, Ryan Prout, Subil Abraham, Wael Elwasif, N. Quentin Haas, Jeffrey Skolnick, Jianlin Cheng, and Ada Sedova. 2021. High-Performance Deep Learning Toolbox for Genome-Scale Prediction of Protein Structure and Function. In *2021 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC)*. 46–57. https://doi.org/10.1109/MLHPC54614.2021.00010

[25] Angel E Garcia, Henry Herce, and Dietmar Paschek. 2006. Simulations of temperature and pressure unfolding of peptides and proteins with replica exchange molecular dynamics. *Annual Reports in Computational Chemistry* 2 (2006), 83–95.

[26] Jens Glaser, Josh V Vermaas, David M Rogers, Jeff Larkin, Scott LeGrand, Swen Boehm, Matthew B Baker, Aaron Scheinberg, Andreas F Tillack, Mathialakan Thavappiragasam, et al. 2021. High-throughput virtual laboratory for drug discovery using massive datasets. *The International Journal of High Performance Computing Applications* (2021), 10943420211001565.

[27] Pablo Herrera-Nieto, Adrià Pérez, and Gianni De Fabritiis. 2020. Characterization of partially ordered states in the intrinsically disordered N-terminal domain of p53 using millisecond molecular dynamics simulations. *Scientific Reports* 10, 1 (jul 2020), 1–8. https://doi.org/10.1038/s41598-020-69322-2

[28] Gerhard Hummer. 2005. Position-dependent diffusion coefficients and free energies from Bayesian analysis of equilibrium and replica molecular dynamics simulations. *New Journal of Physics* 7, 1 (2005), 34.

[29] Anubhav Jain, Shyue Ping Ong, Wei Chen, Bharat Medasani, Xiaohui Qu, Michael Kocher, Miriam Brafman, Guido Petretto, Gian-Marco Rignanese, Geoffroy Hautier, et al. 2015. FireWorks: a dynamic workflow system designed for high-throughput applications. *Concurrency and Computation: Practice and Experience* 27, 17 (2015), 5037–5059.

[30] Weile Jia, Han Wang, Mohan Chen, Denghui Lu, Lin Lin, Roberto Car, E Weinan, and Linfeng Zhang. 2020. Pushing the limit of molecular dynamics with ab initio accuracy to 100 million atoms with machine learning. In *SC20: International conference for high performance computing, networking, storage and analysis*. IEEE, 1–14.

[31] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. 2021. Highly accurate protein structure prediction with AlphaFold. *Nature* (2021), 1–11.

[32] Shankar Kumar, John M Rosenberg, Djamal Bouzida, Robert H Swendsen, and Peter A Kollman. 1992. The weighted histogram analysis method for free-energy calculations on biomolecules. I. The method. *Journal of Computational Chemistry* 13, 8 (1992), 1011–1021.

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

[34] Agnese Marcato, Gianluca Boccardo, and Daniele Marchisio. 2021. A computational workflow to study particle transport and filtration in porous media: Coupling CFD and deep learning. *Chemical Engineering Journal* 417 (2021), 128936.

[35] Andre Merzky, Matteo Turilli, Manuel Maldonado, and Shantenu Jha. 2018. Design and performance characterization of radical-pilot on Titan. *arXiv preprint arXiv:1801.01843* (2018).

[36] John Ossyra, Ada Sedova, Arnold Tharrington, Frank Noé, Cecilia Clementi, and Jeremy C Smith. 2019. Porting adaptive ensemble molecular dynamics workflows to the summit supercomputer. In *International Conference on High Performance Computing*. Springer, 397–417.

[37] John R Ossyra, Ada Sedova, Matthew B Baker, and Jeremy C Smith. 2019. Highly interactive, steered scientific workflows on hpc systems: Optimizing design solutions. In *International Conference on High Performance Computing*. Springer, 514–527.

[38] Sam Partee, Matthew Ellis, Alessandro Rigazzi, Scott Bachman, Gustavo Marques, Andrew Shao, and Benjamin Robbins. 2021. Using machine learning at scale in hpc simulations with smartsim: An application to ocean climate modeling. *arXiv preprint arXiv:2104.09355* (2021).

[39] Robert M. Patton, J. Travis Johnston, Steven R. Young, Catherine D. Schuman, Don D. March, Thomas E. Potok, Derek C. Rose, Seung-Hwan Lim, Thomas P. Karnowski, Maxim A. Ziatdinov, and Sergei V. Kalinin. 2018. 167-PFlops Deep Learning for Electron Microscopy: From Learning Physics to Atomic Manipulation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Dallas, Texas) *(SC '18)*. IEEE Press, Piscataway, NJ, USA, Article 50, 11 pages. http://dl.acm.org/citation.cfm?id=3291656.3291723

[40] Robert M. Patton, J. Travis Johnston, Steven R. Young, Catherine D. Schuman, Thomas E. Potok, Derek C. Rose, Seung-Hwan Lim, Junghoon Chae, Le Hou, Shahira Abousamra, Dimitris Samaras, and Joel Saltz. 2019. Exascale Deep Learning to Accelerate Cancer Research. In *2019 IEEE International Conference on Big Data (Big Data)*. 1488–1496. https://doi.org/10.1109/BigData47090.2019.9006467

[41] Iman Pouya, Sander Pronk, Magnus Lundborg, and Erik Lindahl. 2017. Copernicus, a hybrid dataflow and peer-to-peer scientific computing platform for efficient large-scale ensemble sampling. *Future Generation Computer Systems* 71 (2017), 18–31.

[42] Sander Pronk, Iman Pouya, Magnus Lundborg, Grant Rotskoff, Björn Wesén, Peter M Kasson, and Erik Lindahl. 2015. Molecular simulation workflows as parallel algorithms: The execution engine of Copernicus, a distributed high-performance computing platform. *Journal of Chemical Theory and Computation* 11, 6 (2015), 2600–2608.

[43] Matthew Rocklin. 2015. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th python in science conference*, Vol. 130. Citeseer, 136.

[44] Brian Sanders, Suman Pohkrel, Audrey Labbe, Irimpan Mathews, Connor Cooper, Russell Davidson, Gwyndalyn Phillips, Kevin Weiss, Qiu Zhang, Hugh O'Neill, Manat Kaur, Lori Ferrins, Jurgen Schmidt, Walter Reichard, Surekha Surendranathan, Desigan Kumaran, Babak Andi, Gyorgy Babnigg, Nigel Moriarty, Paul Adams, Andrzej Joachimiak, Colleen Jonsson, Soichi Wakatsuki, Stephanie Galanie, Martha Head, and Jerry Parks. 2021. Potent and Selective Covalent Inhibitors of the Papain-like Protease from SARS-CoV-2. *Research square* (oct 2021). https://doi.org/10.21203/rs.3.rs-906621/v1

[45] Eric O Scott and Kenneth A De Jong. 2015. Understanding simple asynchronous evolutionary algorithms. In *Proceedings of the 2015 ACM Conference on Foundations of Genetic Algorithms XIII*. 85–98.

[46] Renan Souza, Vitor Silva, Daniel Oliveira, Patrick Valduriez, Alexandre AB Lima, and Marta Mattoso. 2015. Parallel execution of workflows driven by a distributed database management system. In *ACM/IEEE Conference on Supercomputing, Poster*.

[47] Chuan Tian, Koushik Kasavajhala, Kellon A.A. Belfon, Lauren Raguette, He Huang, Angela N. Migues, John Bickel, Yuzhang Wang, Jorge Pincay, Qin Wu, and Carlos Simmerling. 2020. Ff19SB: Amino-Acid-Specific Protein Backbone Parameters Trained against Quantum Mechanics Energy Surfaces in Solution. *Journal of Chemical Theory and Computation* 16, 1 (jan 2020), 528–552. https://doi.org/10.1021/acs.jctc.9b00591

[48] Matteo Turilli, Mark Santcroos, and Shantenu Jha. 2018. A comprehensive perspective on pilot-job systems. *ACM Computing Surveys (CSUR)* 51, 2 (2018), 43.

[49] E Weinan, Weiqing Ren, and Eric Vanden-Eijnden. 2002. String method for the study of rare events. *Physical Review B* 66, 5 (2002), 052301.

[50] Katherine Wolstencroft, Robert Haines, Donal Fellows, Alan Williams, David Withers, Stuart Owen, Stian Soiland-Reyes, Ian Dunlop, Aleksandra Nenadic, Paul Fisher, et al. 2013. The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud. *Nucleic Acids Research* 41, W1 (2013), W557–W561.

[51] Thomas B Woolf and Benoit Roux. 1994. Conformational flexibility of o-phosphorylcholine and o-phosphorylethanolamine: a molecular dynamics study of solvation effects. *Journal of the American Chemical Society* 116, 13 (1994), 5916–5926.

[52] Justin M Wozniak, Timothy G Armstrong, Michael Wilde, Daniel S Katz, Ewing Lusk, and Ian T Foster. 2013. Swift/T: Large-scale application composition via distributed-memory dataflow processing. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. IEEE, 95–102.

[53] Steven R Young, Derek C Rose, Travis Johnston, William T Heller, Thomas P Karnowski, Thomas E Potok, Robert M Patton, Gabriel Perdue, and Jonathan Miller. 2017. Evolving Deep Networks Using HPC. In *Proceedings of the Machine Learning on HPC Environments*. ACM, 7.

[54] Li Zhong, Dennis Hoppe, Naweiluo Zhou, and Oleksandr Shcherbakov. 2021. Hybrid workflow of Simulation and Deep Learning on HPC: A Case Study for Material Behavior Determination. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 698–704.