signac: Data Management and Workflows for Computational Researchers

Bradley D. Dice[‡†*], Brandon L. Butler[§†*], Vyas Ramasubramani[§], Alyssa Travitz[¶], Michael M. Henry[‖], Hardik Ojha[**], Kelly L. Wang[¶], Carl S. Adorf[§], Eric Jankowski[‖], Sharon C. Glotzer[‡§¶††]

Abstract—The **signac** data management framework (https://signac.io) helps researchers execute reproducible computational studies, scales workflows from laptops to supercomputers, and emphasizes portability and fast prototyping. With **signac**, users can track, search, and archive data and metadata for file-based workflows and automate workflow submission on high performance computing (HPC) clusters. We will discuss recent improvements to the software's feature set, scalability, scientific applications, usability, and community. Newly implemented synced data structures, features for generalized workflow execution, and performance optimizations will be covered, as well as recent research using the framework and changes to the project's outreach and governance as a response to its growth.

Index Terms—data management, data science, database, simulation, collaboration, workflow, HPC, reproducibility

Introduction

Scientific research addresses problems where questions often change rapidly, data models are always in flux, and compute infrastructure varies widely from project to project. The **signac** data management framework [ADRG18] is a tool designed by researchers, for researchers, to simplify the process of prototyping and then performing reproducible scientific computations. It forgoes encoding complex data files into a database in favor of working directly on file systems, providing fast indexing utilities for a set of directories. Using **signac**, a data space on the file system can be initialized, searched, and modified using either a Python or command-line interface. By its general-purpose design, **signac** is agnostic to data content and format. The companion package **signac-flow** interacts with the data space to generate and analyze data through reproducible workflows that scale from laptops to supercomputers. Arbitrary shell commands can be run by **signac-flow** as part of a workflow, making it as flexible as a script in any language of choice.

† These authors contributed equally.
∗ Corresponding author: bdice@umich.edu, butlerbr@umich.edu
‡ Department of Physics, University of Michigan, Ann Arbor
∗ Corresponding author: bdice@umich.edu, butlerbr@umich.edu
§ Department of Chemical Engineering, University of Michigan, Ann Arbor
¶ Macromolecular Science and Engineering Program, University of Michigan, Ann Arbor
‖ Micron School of Materials Science and Engineering, Boise State University
∗∗ Department of Chemical Engineering, Indian Institute of Technology Roorkee
†† Biointerfaces Institute, University of Michigan, Ann Arbor

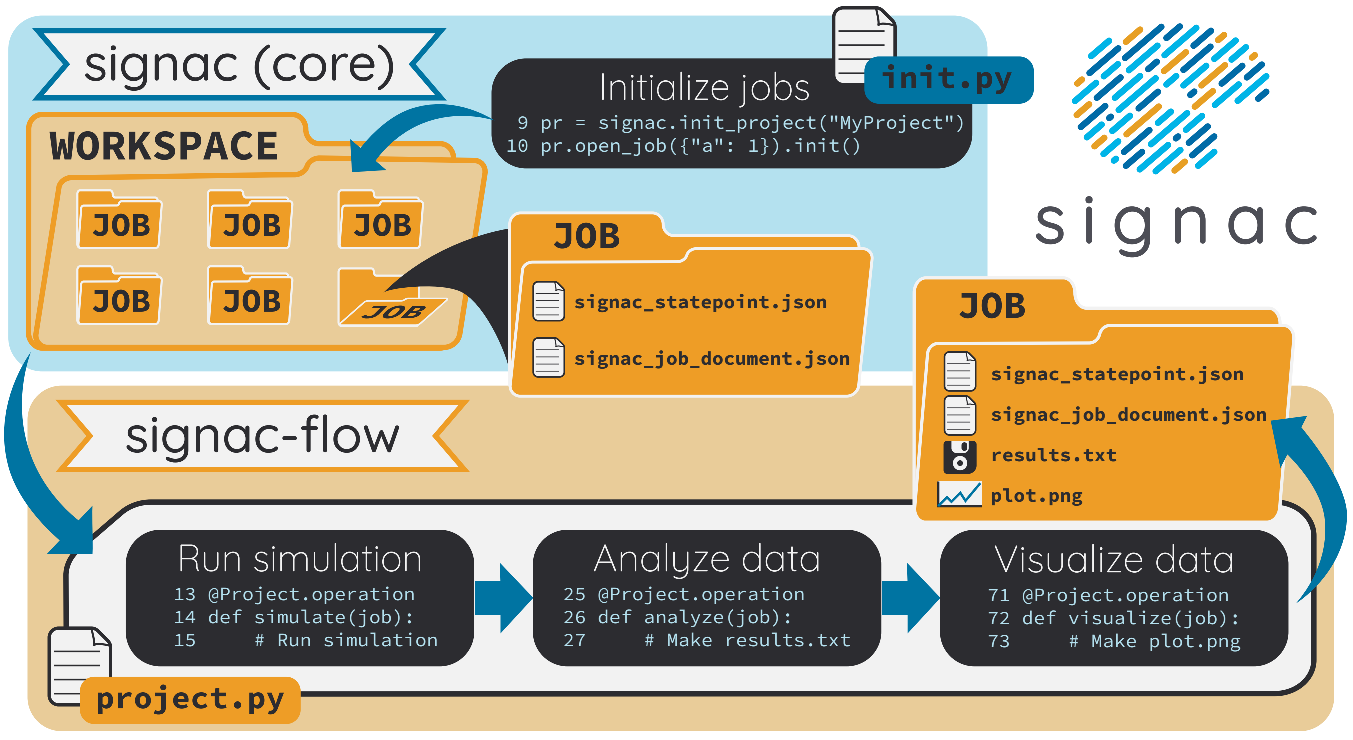
This paper will focus on developments to the **signac** framework over the last 3 years, during which features, flexibility, usability, and performance have been greatly improved. The core data structures in **signac** have been overhauled to provide a powerful and generic implementation of *synced collections*, that we will leverage in future versions of **signac** to enable more performant data indexing and flexible data layouts. In **signac-flow**, we have added support for submitting *groups* of operations with conditional dependencies, allowing for more efficient utilization of large HPC resources. Further developments allow for operations to act on arbitrary subsets of the data space via *aggregation*, rather than single jobs alone. Moving beyond code development, this paper will also discuss the scientific research these features have enabled and organizational developments supported through key partnerships. We will share our project's experience in continuously revising project governance to encourage sustained contributions, adding more entry points for learning about the project (Slack support, weekly public office hours), and participating in Google Summer of Code in 2020 as a NumFOCUS Affiliated Project. Much of the work has been carried out in conjunction with the Molecular Simulation Design Framework (MoSDeF) [CMI+21], a National Science Foundation Cyberinfrastructure for Sustained Scientific Innovation (CSSI) effort.

Structure and implementation

With **signac**, file-based data and metadata are organized in folders and JSON files, respectively (see Figure 1). A **signac** data space, or *workspace*, contains jobs, which are individual directories associated with a single primary key known as a *state point* stored in a file `signac_statepoint.json` in that directory. The JSON files allow **signac** to index the data space, providing a database-like interface to a collection of directories. Arbitrary user data may be stored in user-created files in these jobs, although **signac** also provides convenient facilities for storing simple lightweight data or array-like data via JSON (the "job document") and HDF5 (the "job data") utilities. Readers seeking more details about **signac** are referred to past **signac** papers [ADRG18], [RAD+18] as well as the **signac** website[1] and documentation[2].

This filesystem-based approach has both advantages and disadvantages. Its key advantages lie in flexibility and portability. The serverless design removes the need for any external running server process, making it easy to operate on any filesystem. The

1. https://signac.io
2. https://docs.signac.io



```
pr = signac.init_project("MyProject")
pr.open_job({"a": 1}).init()
```

```
@Project.operation
def simulate(job):
    # Run simulation
```

```
@Project.operation
def analyze(job):
    # Make results.txt
```

```
@Project.operation
def visualize(job):
    # Make plot.png
```

Fig. 1: *Overview of the* ***signac*** *framework. Users first create a project, which initializes a workspace directory on disk. Users define state points which are dictionaries that uniquely identify a job. The workspace holds a directory for each job, containing JSON files that store the state point and job document. The job directory name is a hash of the state point's contents. Here, the* `init.py` *file initializes an empty project and adds one job with state point* `{"a": 1}`*. Next, users define a workflow using a subclass of* ***signac-flow****'s* `FlowProject`*. The workflow shown has three operations (simulate, analyze, visualize) that, when executed, produce two new files* `results.txt` *and* `plot.png` *in the job directory.*

design is also intrinsically distributed, making it well suited for highly parallel workflows where multiple processes concurrently read or write file-based data stored in job directories. Conversely, this distributed approach precludes the performance advantages of centralized data stores with persistent indexes in memory. Typically, the **signac** approach works very well for projects up to 100,000 jobs, while significantly larger projects may have wait times that constrain interactive usage. These limits are inherent to **signac**'s use of small files for each job's state point, but the framework has been aggressively optimized and uses extensive caching/buffering to maximize the achievable throughput within this model.

The framework is a strong choice for applications meeting one or more of the following criteria:

- input/output data is primarily file-based
- prototype research code where data schemas may change or evolve
- computations will use an HPC cluster
- the amount of computation per job is large
- parameter sweeps over a range of values (with values on a grid or dynamically determined by e.g. active learning)
- heterogeneous data (not all jobs have the same keys present in their state points)

For example, M. W. Thompson *et al.* in [TMS+] used 396 jobs/state points to execute computer simulations of room-temperature ionic liquids with GROMACS [PPS+], [LHvdS], [HKvdSL], [AMS+] simulations. The study investigated 18 compositions (by mass fraction) and 22 unique solvents from five chemical families (nitriles, alcohols, halocarbons, carbonyls, and glymes), with a state point for each pairing of mass fraction and solvent type.

Users working with large tabular data (e.g. flat files on disk or data from a SQL database) may prefer to use libraries like pandas [pdt20], [McK], Dask [Tea16], [Roc15], or RAPIDS [Tea18] that are specifically designed for those use cases. However, it is possible to create a **signac** project with state points corresponding to each row, which may be a good use of **signac** if there is file-based data affiliated with each row's parameters.

Code examples of features presented in this paper can be found online[3].

Applications of signac

The **signac** framework has been cited 54 times, according to Google Scholar, and has been used in a range of scientific fields with various types of computational workflows. Some of these studies include quantum calculations of small molecules [GG18], 4,480 simulations of epoxy curing (each containing millions of particles) [TAH+18], inverse design of pair potentials [AADG18], identifying photonic band gaps in 151,593 crystal structures [CADG21], benchmarking atom-density representations for use in machine learning [MVG+21], simulating fluid flow in polymer solutions [PHMTN19], design of optical metamaterials [HCVM20], and economic analysis of drought

3. https://github.com/glotzerlab/signac-examples

risk in agriculture [RD20]. To date, **signac** users have built workflows utilizing a wide range of software packages including simulation tools such as Cassandra and MoSDeF-Cassandra [SMRM+17], [DMD+21], foyer [KST+], GROMACS [PPS+], [LHvdS], [HKvdSL], [AMS+], HOOMD-blue [AGG], [GNA+], [BLBVRJAASCG20], mBuild [KSJ+], MIT Photonic Bands [JJ01], Quantum-ESPRESSO [GBB+09], Rigorous Coupled Wave Analysis (RCWA) [LF12], and VASP [KF96], machine learning libraries including Keras [C+15], scikit-learn [PVG+11], and TensorFlow [AAB+15], and analysis libraries for postprocessing data such as freud [RDH+20], librascal [MVG+21], MDAnalysis [MADWB11], MDTraj [MBH+15], and OVITO [Stu]. Much of the published research using **signac** comes from chemical engineering, materials science, or physics, the fields of many of **signac**'s core developers and thus fields where the project has had greatest exposure. Computational materials research commonly requires large HPC resources with shared file systems, a use case where **signac** excels. However, there are many other fields with similar hardware needs where **signac** can be applied. These include simulation-heavy HPC workloads such as fluid dynamics, atomic/nuclear physics, or genomics, data-intensive fields such as economics or machine learning, and applications needing fast, flexible prototypes for optimization and data analysis.

While there is no "typical" **signac** project, factors such as computational complexity and data sizes offer some rough guidelines for when **signac**'s database-on-the-filesystem is appropriate. For instance, the time to check the status of a workflow depends on the number of jobs, number of operations, and number of conditions to evaluate for those jobs. Typical **signac** projects have 100 to 10,000 jobs, with each job workspace containing arbitrarily large data sizes (the total file size of the job workspace has little effect on the speed of the **signac** framework). To give a rough idea of the limits of scalability, **signac** projects can contain up to around 100,000 jobs while keeping common tasks like checking workflow status in an "interactive" time scale of 1-2 minutes. Some users that primarily wish to leverage **signac-flow**'s workflows for execution and submission may have a very small number of jobs (< 10). One example of this would be executing a small number of expensive biomolecular simulations using different random seeds in each job's state point. Importantly, projects with a small number of jobs can be expanded at a later time, and make use of the same workflow defined for the initial set of jobs. The abilities to grow a project and change its schema on-the-fly catalyze the kind of exploration that is crucial to answering research questions.

The workflow submission features of **signac-flow** interoperates with popular HPC schedulers including SLURM, PBS/TORQUE, and LSF automating the generation and submission of scheduler batch scripts. Directives are set through Python decorators and define resource and execution requests for operations. Examples of directives include number of CPUs or GPUs, the walltime, and memory. The use of directives allows **signac-flow** workflows to be portable across HPC systems by generating resource requests that are specific to each machine's scheduler.

Overview of new features

The last three years of development of the **signac** framework have expanded its usability, feature set, user and developer documentation, and potential applications. Some of the largest architectural changes in the framework will be discussed in their own sections, namely extensions of the workflow model (support for executing groups of operations and aggregators that allow operations to act on multiple jobs) and a much more performant and flexible re-implementation of the core "data structure" classes that synchronize **signac**'s Python representation of state points and job documents with JSON-encoded dictionaries on disk.

Data archival

The primary purpose of the core **signac** package is to simplify and accelerate data management. The **signac** command line interface is a common entry point for users, and provides subcommands for searching, reading, and modifying the data space. New commands for import and export simplify the process of archiving **signac** projects into a structure that is both human-readable and machine-readable for future access (with or without **signac**). Archival is an integral part of research data operations that is frequently overlooked. By using highly compatible and long-lived formats such as JSON for core data storage with simple name schemes, **signac** aims to preserve projects and make it easier for studies to be independently reproduced. This is aligned with the principles of TRUE (Transparent, Reproducible, Usable by others, and Extensible) simulations put forth by the MoSDeF collaboration [TGM+20].

Improved data storage, retrieval, and integrations

Data access via the shell: The `signac shell` command allows the user to quickly enter a Python interpreter that is pre-populated with variables for the current project or job (when in a project or job directory). This means that manipulating a job document or reading data can be done through a hybrid of bash/shell commands and Python commands that are fast to type.

```
~/project $ ls
signac.rc workspace
~/project $ cd workspace/42b7b4f2921788e.../
~/project/workspace/42b7b4f2921788e... $ signac shell
Python 3.8.3
signac 1.6.0

Project:         test
Job:             42b7b4f2921788ea14dac5566e6f06d0
Root:            ~/project
Workspace:       ~/project/workspace
Size:            1

Interact with the project interface using the
"project" or "pr" variable. Type "help(project)"
or "help(signac)" for more information.

>>> job.sp
{'a': 1}
```

HDF5 support for storing numerical data: Many applications used in research generate or consume large numerical arrays. For applications in Python, NumPy arrays are a de facto standard for in-memory representation and manipulation. However, saving these arrays to disk and handling data structures that mix dictionaries and numerical arrays can be cumbersome. The **signac** H5Store feature offers users a convenient wrapper around the h5py library [Col13] for loading and saving both hierarchical/key-value data and numerical array data in the widely-used HDF5 format [Gro21]. The `job.data` attribute is an instance of the `H5Store` class, and is a key-value store saved on disk as `signac_data.h5` in the job workspace. Users who prefer to split data across multiple files can use the `job.stores` API to save in multiple HDF5 files. Corresponding `project.data` and `project.stores` attributes exist, which save data files in the project root directory. Using an instance of `H5Store` as a context manager allows users to keep the HDF5 file open while reading large chunks of the data:

```
with job.data:
    # Copy array data from the file to memory
    # (which will persist after the HDF5 file is
    # closed) by indexing with an empty tuple:
    my_array = job.data["my_array"][()]
```

Advanced searching and filtering of the workspace: The `signac diff` command, available on both the command line and Python interfaces, returns the difference between two or more state points and allows for easily assessing subsets of the data space. By unifying state point and document queries, filtering, and searching workspaces can be more fine-grained and intuitive.

Data visualization and integrations

Integrating with the PyData ecosystem: Users can now summarize data from a **signac** project into a pandas DataFrame for analysis. The `project.to_dataframe()` feature exports state point and job document information to a pandas DataFrame in a consistent way that allows for quick analysis of all jobs' data. Support for Jupyter notebooks [KRKP+16] has also been added, enabling rich HTML representations of **signac** objects.

Dashboards: The companion package **signac-dashboard** allows users to quickly visualize data stored in a **signac** data space. The dashboard runs in a browser and allows users to display job state points, edit job documents, render images and videos, download any file from a job workspace, and search or browse through state points in their project. Dashboards can be hosted on remote servers and accessed via port forwarding, which makes it possible to review data generated on a remote HPC system without needing to copy it back to a local system for inspection. Users can quickly save notes into the job document and then search those notes, which is useful for high throughput studies that require some manual investigation (e.g. reviewing plots).

Performance enhancements

In early 2021, a significant portion of the codebase was profiled and refactored to improve performance and these improvements were released in **signac** 1.6.0 and **signac-flow** 0.12.0. As a result of these changes, large **signac** projects saw 4-7x speedups for operations such as iterating over the jobs in a project compared to the 1.5.0 release of **signac**. Similarly, performance of a sample workflow that checks status, runs, and submits a FlowProject with 1,000 jobs, 3 operations, and 2 label functions improved roughly 4x compared to **signac-flow** 0.11.0. These improvements allow **signac** to scale to ~100,000 jobs.

In **signac**, the core of the `Project` and `Job` classes were refactored to support lazy attribute access and delayed initialization, which greatly reduces the total amount of disk I/O by waiting until data is actually requested by the user. Other improvements include early exits in functions, reducing the number of required system calls with smarter usage of the `os` library, and switching to algorithms that operate in constant time, $O(1)$, instead of linear time, $O(N_{jobs})$. Optimizations were identified by profiling the performance of common operations on small and large real-world projects with cProfile and visualized with snakeviz [Dav].

Similarly, performance enhancements were also made in the **signac-flow** package. Some of the optimizations identified include lazy evaluation of run commands and directives, and caching of job status information. In addition, the improvements in **signac** such as faster iteration over large **signac** projects used in **signac-flow** made **signac-flow**'s primary functions — checking project status, executing operations, and submitting operations to a cluster — significantly faster.

Improved user output

Workflow graph detection: The preconditions and postconditions of operations in a **signac-flow** `FlowProject` implicitly define a graph. For example, if the operation "analyze" depends on the operation "simulate" via the precondition `@FlowProject.pre.after(simulate)`, then there is a directed edge from "simulate" to "analyze." This graph can now be detected from the workflow conditions and returned in a NetworkX [HSS08] compatible format for display or inspection.

Templated status output: Querying the status of a **signac-flow** project now has many options controlling the information displayed and has been templated to allow for plain text, Markdown, or HTML output. In doing so, the output has also become cleaner and compatible with external tools.

Enhanced workflows

Directives: Execution directives (or *directives* for short) provide a way to specify required resources on HPC schedulers such as number of CPUs/GPUs, MPI ranks, OpenMP threads, walltime, memory, and others. Directives can be a function of the job as well as the operation, allowing for great flexibility. In addition, directives work seamlessly with operation groups, job aggregation, and submission bundling (all of which are described in the following section).

Dynamic workspaces: The **signac-flow** package can now handle workspaces where jobs are created as the result of operations on other jobs. This is crucial for optimization workflows and iteratively sampling parameter spaces, and allows projects to become more automated with some data points only run if a prior condition on another data point is reached.

Executing complex workflows via groups and aggregation

Two new concepts in **signac-flow** provide users with significantly more power to implement complex workflows: *groups* and *aggregation*. A related third concept – *bundling* – which is not new, also provides flexibility to users in their workflows, but exclusively affects scheduler submission, not workflow definition. Figure 2 show a graphical illustration of the three concepts.

As the names of both groups and aggregation imply, the features enable the "grouping" or "aggregating" of existing concepts: operations in the case of groups and jobs in the case of aggregates. The conceptual model of **signac-flow** builds on **signac**'s notions of the `Project` and `Job` (the unit of the data space) through a `FlowProject` class that adds the ability to define and execute operations (the unit of a workflow) that act on jobs. Operations are Python functions or shell commands that act on a job within the data space, and are defined using Python decorator syntax.

```
# project.py
from flow import FlowProject

@FlowProject.operation
@Flowproject.post.true("initialized")
def initialize(job):
    # perform necessary initialize steps
    # for simulation
    job.doc.initialized == True
```

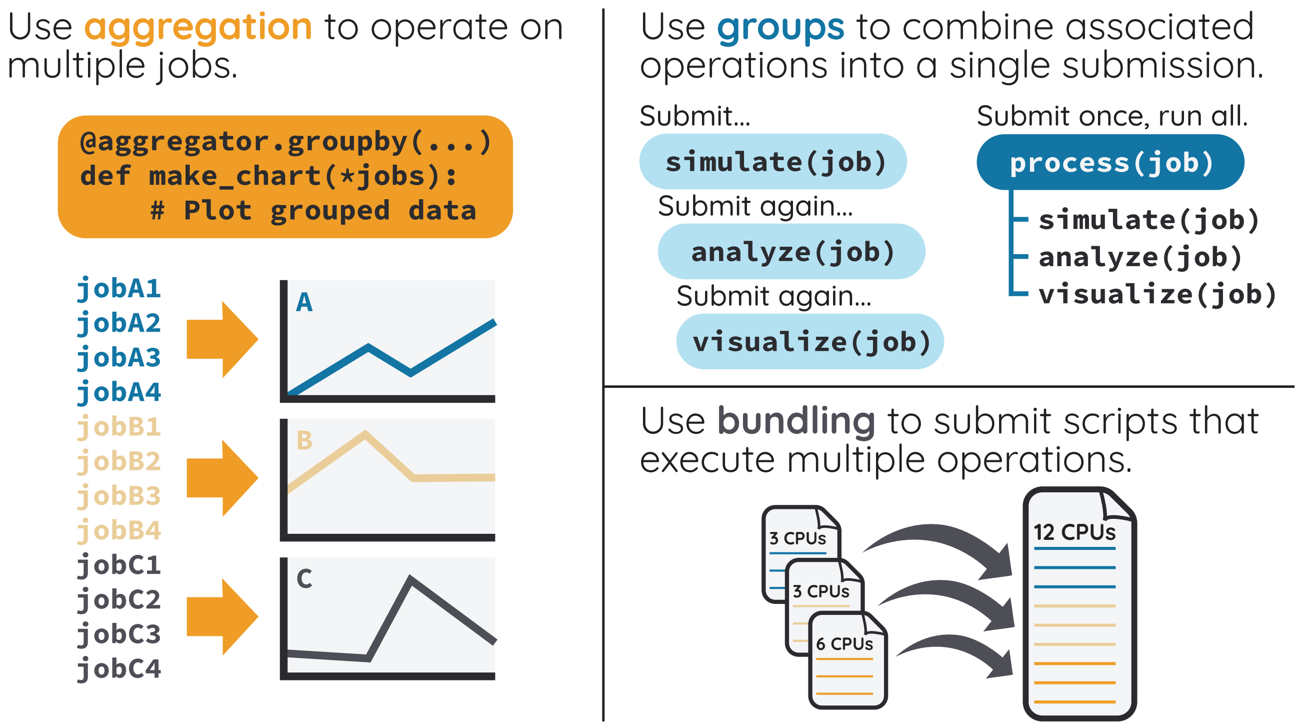


Fig. 2: Aggregation, groups, and bundling allow users to build complex workflows. The features are orthogonal, and can be used in any combination. Aggregation enables one operation or group to act on multiple jobs. Groups allow users to combine multiple operations into one, with dependencies among operations resolved at run time. Bundling helps users efficiently leverage HPC schedulers by submitting multiple commands in the same script, to be executed in serial or parallel.

```python
if __name__ == "__main__":
    FlowProject().main()
```

When this project is run using **signac-flow**'s command line API (`python project.py run`), the current state point is prepared for simulation. Operations can have preconditions and postconditions that define their eligibility. All preconditions must be met in order for a operation to be eligible for a given job. If all postconditions are met, that indicates an operation is complete (and thus ineligible). Examples of such conditions include the existence of an input file in a job's workspace or a key in the job document (as shown in the above snippet). However, this type of conditional workflow can be inefficient when sequential workflows are coupled with an HPC scheduler interface, because the user must log on to the HPC and submit the next operation after the previous operation is complete. The desire to submit large and long-running jobs to HPC schedulers encourages users to write large operation functions which are not modular and do not accurately represent the individual units of the workflow, thereby limiting **signac-flow**'s utility and reducing the readability of the workflow.

Groups

Groups, implemented by the `FlowGroup` class and `FlowProject.make_group` method, allows users to combine multiple operations into a single entity that can be run or submitted. Submitting a group allows **signac-flow** to dynamically resolve preconditions and postconditions of operations as each operation is executed, making it possible to combine separate operations (e.g. for simulation and analysis and plotting) into a single submission script that will execute eligible operations in sequence. This allows users to write smaller, modular functions, which may require a specific order of execution, without sacrificing the ability to submit large, long-running jobs on HPCs. Furthermore, groups are aware of directives and can properly combine the directives of their constituent operations to specify resources and quantities like walltime whether executing in parallel or serial.

```python
from flow import FlowProject

example_group = FlowProject.make_group(
    name="example_group")

@example_group.with_directives(
    {"ngpu": 2,
     "walltime": lambda job: job.doc.hours_to_run})
@FlowProject.post.true("simulated")
@FlowProject.operation
def simulate(job):
    # run simulation
    job.doc.simulated = True

@example_group
@FlowProject.pre.after(simulate)
@FlowProject.post.true("analyzed")
@FlowProject.operation
def analyze(job):
    # analyze simulation results
    job.doc.analyzed = True
```

Groups also allow for specifying multiple machine specific resources (CPU or GPU) with the same operation. An operation can have unique directives for each distinct group to which it belongs. By associating an operation's directives with respect to a specific group, groups can represent distinct compute environments, such as a local workstation or a remote supercomputing cluster. The below snippet shows an `expensive_simulate` operation which can be executed with three different directives depending on how it is written. If executed through `cpu_group` the operation will request 48 cores, if `gpu_group` 4 GPUs, if neither then it will request 4 cores. This represents the real use case where a user may want to run an operation locally (in this case without a group), or on a CPU or GPU focused HPC/workstation.

```
from flow import FlowProject

cpu_group = FlowProject.make_group(name="cpu")
gpu_group = FlowProject.make_group(name="gpu")

@cpu_group.with_directives({"np": 48})
@gpu_group.with_directives({"ngpu": 4})
@FlowProject.operation.with_directives({"np": 4})
def expensive_simulate(job):
    # expensive simulation run on CPUs or GPUs
    pass
```

Aggregation

Users also frequently work with multiple jobs when performing tasks such as plotting data from all jobs in the same figure. Though the **signac** package has methods like `Project.groupby`, which can generate subsets of the project that are grouped by a state point key, there has been no way to use these "aggregation" features in **signac-flow** for defining workflows. The concept of aggregation provides a straightforward way for users to write and submit operations that act on arbitrary subsets of jobs in a **signac** data space through functions analogous to `Project.groupby`. Just as the groups feature acts as an abstraction over operations, aggregation can be viewed as an abstraction over jobs. When decorated with an aggregator, operations can accept multiple job instances as positional arguments through Python's argument unpacking. Decorators are used to define aggregates, encompassed in the `@aggregator` decorator for single operations and in the argument `aggregator_function` to `FlowProject.make_group` for groups of operations.

```
from flow import FlowProject

@aggregator
@FlowProject.operation
def plot_enzyme_activity(*jobs):
    import matplotlib.pyplot as plt
    import numpy as np

    x = [job.sp.temperature for job in jobs]
    y = [job.doc.activity for job in jobs]
    fig, ax = plt.subplots()
    ax.scatter(x, y)
    ax.set_title(
        "Enzymatic Activity Across Temperature")
    fig.savefig("enzyme-activity.png")
```

Like groups, there are many reasons why a user might wish to use aggregation. For example, a **signac** data space that describes weather data for multiple cities in multiple years might want to plot or analyze data that uses `@aggregator.groupby("city")` to show changes over time for each city in the data space. Similarly, aggregating over replicas (e.g. the same simulation with different random seeds) facilitates computing averaged quantities and error bars. Another example is submitting aggregates with a fixed number of jobs in each aggregate to enable massive parallelization by breaking a large MPI communicator into a smaller communicator for each independent job, which is necessary for efficient utilization of leadership-class supercomputers like OLCF Summit.

Bundling

Finally, bundling is another way to use workflows in conjunction with an HPC scheduling system. Whereas aggregates are concerned with jobs and groups operations, bundling is concerned with combining executable units into a single submission script. This distinction means that bundling is not part of the workflow definition, but is a means of tailoring batch scripts for different HPC systems. Bundles allow users to leverage scheduler resources effectively and minimize queue time, and can be run in serial (the default) or parallel. Users enable bundling by passing the command line argument `--bundle`, optionally with another argument `--parallel` to run each command in the bundle in parallel (the Python API has corresponding options as well). The simplest case of a bundle is a submission script with the same operation being executed for multiple jobs. Bundling is what allows the submission script to contain multiple jobs executing the same operation. By storing information about the generated bundles during submission, **signac-flow** prevents accidental resubmission just as in the unbundled case. While the example mentioned above does not use either groups or aggregation, bundles works seamlessly with both.

Cluster templates

The **signac-flow** software includes automatic detection and script support for SLURM, PBS/TORQUE, and LSF schedulers. However, effective HPC utilization frequently relies on specific information such as numbers of cores per compute node or designated partitions for GPU or large memory applications. To this end, **signac-flow** includes templates for a number of HPC clusters including OLCF Summit and Andes, XSEDE [TCD+14] clusters such as PSC Bridges-2, SDSC Comet, and TACC Stampede2, and university clusters such as the University of Michigan's Great Lakes and University of Minnesota's Mangi. These cluster templates change frequently as HPC systems are brought online and later decommissioned. Users can create their own templates to contribute to the package or use locally.

Synced collections: backend-agnostic, persistent, mutable data structures

Motivation

At its core, **signac** is a tool for organizing and working with data on the filesystem, presenting a Pythonic interface for tasks like creating directories and modifying files. In particular, **signac** makes modifying the JSON files used to store a job's state points and documents as easy as working with Python dictionaries. Despite heavy optimization, when seeking to scale **signac** to ever-larger data spaces, we quickly realized that the most significant performance barrier was the overhead of parsing and modifying large numbers of text files. Unfortunately, the usage of JSON files in this manner was deeply embedded in our data model, which made switching to a more performant backend without breaking APIs or severely complicating our data model a daunting task.

While attempting to separate the **signac** data model from its original backend implementation (manipulating JSON files on disk), we identified a common pattern: providing a dictionary-like interface for an underlying resource. Several well-known Python packages such as h5py [Col13] and zarr [MjD+20] also use dictionary-like interfaces to make working with complex resources feel natural to Python users. Most such packages implement this layer directly for their particular use case, but the nature of the problem suggested to us the possibility of developing a more generic representation of this interface. Indeed, the purpose of the Python standard library's `collections.abc` module to make it easy to define objects that "look like" standard Python objects while having completely customizable behavior under the hood. As such, we saw an opportunity to specialize this pattern for a specific use case: the transparent synchronization of a Python object with an underlying resource.

The *synced collections* framework represents the culmination of our efforts in this direction, providing a generic framework in which interfaces of any abstract data type can be mapped to arbitrary underlying synchronization protocols. In **signac**, this framework allows us to hide the details of a particular file storage medium (like JSON) behind a dictionary-like interface, but it can just as easily be used for tasks such as creating a new, list-like interface that automatically saves all its data in a plain-text CSV format. This section will offer a high-level overview of the synced collections framework and our plans for its use within **signac**, with an eye to potential users in other domains as well.

Summary of features

We designed synced collections to be flexible, easily extensible, and independent of **signac**'s data model. Most practical use cases for this framework involve an underlying resource that may be modified by any number of associated in-memory objects that behave like standard Python collections, such as dictionaries or lists. Therefore, all normal operations must be preceded by loading from this resource and updating the in-memory store, and they must be succeeded by saving to that resource. The central idea behind synced collections is to decouple this process into two distinct groups of tasks: the saving and loading of data from a particular resource backend, and the synchronization of two in-memory objects of a given type. This delineation allows us to, for instance, encapsulate all logic for JSON files into a single `JSONCollection` class and then combine it with dictionary- or list-like `SyncedDict`/`SyncedList` classes via inheritance to create fully functional JSON-backed dictionaries or lists. Such synchronization significantly lowers performance, so the framework also exposes an API to implement buffering protocols to collect operations into a single transaction before submitting them to the underlying resource.

Previously, **signac** contained a single `JSONDict` class as part of its API, along with a separately implemented internal-facing `JSONList` that could only be used as a member of a `JSONDict`. With the new framework, users can create fully-functional, arbitrarily nested `JSONDict` and `JSONList` objects that share the same logic for reading from and writing to JSON files. Just as importantly, **signac** can now combine these data structures with a different backend, allowing us to swap in different storage mechanisms for improved performance and flexibility with no change in our APIs. Since different types of resources may have different approaches to batching transactions — for example, a SQLite backend may want to exploit true SQL transactions, while a Redis backend might simply collect all changes in memory and delay sending memory to the server — synced collections also support customizable buffering protocols, again via class inheritance.

Applications of synced collections

The new synced collections promise to substantially simplify both feature and performance enhancements to the **signac** framework. Performance improvements in the form of Redis-based storage are already possible with synced collections, and as expected they show substantial speedups over the current JSON-based approach. We have also exploited the new and more flexible buffering protocol to implement and test alternatives to the previous approach. In certain cases, our new buffering techniques improve performance of buffered operations by 1-2 orders of magnitude. Some of these performance improvements are drop-in replacements that require no changes to our existing data models, and we plan to enable these in upcoming versions of **signac**.

The generality of synced collections makes them broadly useful even outside the **signac** framework. Adding Pythonic APIs to collection-like objects can be challenging, particularly when those objects should support arbitrary nesting, but synced collections enable nesting as a core feature to dramatically simplify this process. Moreover, while the framework was originally conceived to support synchronization of an in-memory data structure with a resource on disk, it can also be used to synchronize with another in-memory resource. A powerful example of this would be wrapping a C or C++ extension type, for instance by creating a `SyncedList` that synchronizes with a C++ `std::vector`, such that changes to either object would be transparently reflected in the other. With synced collections, creating this class just requires defining a conversion between a `std::vector` and a raw Python list, a trivial task using standard tools for exposing extension types such as pybind or Cython.

At a higher level, synced collections represent an important step in improving both the scalability and flexibility of **signac**. By abstracting away details of persistent file storage from the rest of **signac**, they make it much easier for the rest of **signac** to focus on offering flexible data models. One of the most common use cases of **signac** is creating data spaces with homogeneous schemas that fit naturally into tabular data structures. In future iterations of **signac**, we plan to allow users to opt into homogeneous schemas, which would enable us to replace file-based indexes with SQL-backed databases that would offer orders of magnitude in performance improvements. Using this flexibility, we could also move away from our currently rigid workspace model to allow more general data layouts on disk for cases where users may benefit from more general folder structures. As such, synced collections are a stepping stone to creating a more general and powerful version of **signac**.

Project evolution

The **signac** project has evolved from being an open-source project mostly developed and managed by the Glotzer Group at the University of Michigan, to being supported by over 30 contributors and 8 committers/maintainers on 3 continents and with over 55 citations from academic and government research labs and 12 talks at large scientific, Python, and data science conferences. The growth in involvement with **signac** results from our focus on developing features based on user needs, as well as our efforts to transition **signac** users into **signac** contributors, through many initiatives in the past few years. Through encouraging users to become contributors, we ensure that **signac** addresses real users' needs. Early on, we identified that the framework had the potential to be used by a wide community of researchers and that its philosophy was aligned with other projects in the scientific Python ecosystem. We have expanded **signac**'s contributor base beyond the University of Michigan through research collaborations such as the MoSDeF CSSI with other universities, sharing the framework at conferences, and through the Google Summer of Code (GSoC) program, which we applied to under the NumFOCUS organization. Working with and mentoring students through GSoC led to a new committer and significant work on the synced collections and aggregation projects presented above. We provide active support and open discussion for the contributor and user community through Slack. In addition, we have started hosting weekly "office hours" for in-person (virtual) introduction and guided contributions to the code base. By pairing new contributors with experienced **signac** developers, we significantly reduce the knowledge barrier to joining a new project. Close interactions between developers and users during office hours has led to more features and documentation born directly out of user need. Contributing to documentation has been a productive starting point for new users-turned-contributors, both for the users and the project, since it improves the users' familiarity with the API as well as addresses weak spots in the documentation that are more obvious to new users.

In our growth with increasing contributors and users, we recognized a need to change our governance structure to make contributing easier and provide a clear organizational structure to the community. We based our new model on the Meritocratic Governance Model and our manager roles on Numba [LPS] Czars. We decided on a four category system with maintainers, committers, contributors, and users. Code review and pull request merge responsibilities are granted to maintainers and committers, who are (self-) nominated and accepted by a vote of the project maintainers. Maintainers are additionally responsible for the strategic direction of the project and administrative duties. Contributors consist of all members of the community who have contributed in some way to the framework, which includes adding or refactoring code as well as filing issues and improving documentation. Finally, users refer to all those who use **signac** in any capacity.

In addition, to avoid overloading our committers and maintainers, we added three rotating manager roles to our governance model that ensure project management goes smoothly: triage, community, and release. These managers have specific rotation policies based on time (or release cycles). The triage manager role rotates weekly and looks at new issues or pull requests and handles cleanup of outdated issues. The community manager role rotates monthly and is in charge of meeting planning and outreach. Lastly, the release manager rotates with each release cycle and is the primary decision maker for the timeline and feature scope of package releases. This prevents burnout among our senior developers and provides a sense of ownership to a greater number of people, instead of relying on a "benevolent dictator/oligarchy for life" mode of project leadership.

Conclusions

From the birth of the **signac** framework in 2015 to now, **signac** has grown in usability, performance, and use. In the last three years, we have added exciting new features such as groups, aggregation, and synced collections, while learning how to manage outreach and establish sustainable project governance in a burgeoning scientific open-source project. We hope to continue expanding the framework through user-oriented development, reach users in research fields beyond materials science that routinely have projects suited for **signac**, and welcome new contributors with diverse backgrounds and skills to the project.

Installing signac

The **signac** framework is tested for Python 3.6+ and is compatible with Linux, macOS, and Windows. The software is available under the BSD-3 Clause license. To install, execute

```
conda install -c conda-forge signac \
signac-flow signac-dashboard
```

or

```
pip install signac signac-flow signac-dashboard
```

Source code is available on GitHub[4][5] and documentation is hosted online by ReadTheDocs[6].

Acknowledgments

We would also like to thank NumFOCUS for providing helpful advice on open-source governance, project sustainability, and community outreach, as well as funding for the design of the **signac** project logo.

This work was supported by the National Science Foundation, Office of Advanced Cyberinfrastructure Awards OAC 1835612 and OAC 1835593. B.D. and B.B. acknowledge fellowship support from the National Science Foundation under ACI 1547580, S212: Impl: The Molecular Sciences Software Institute [WDC18], [KWB$^+$18]. B.D. was also supported by a National Science Foundation Graduate Research Fellowship Grant DGE 1256260 (2016–2019). V.R. acknowledges the 2019-2020 J. Robert Beyster Computational Innovation Graduate Fellowship at the University of Michigan. A.T. is supported by the National Science Foundation under DMR 1707640. Software was deployed and validated and benchmarked on the Extreme Science and Engineering Discovery Environment (XSEDE) [TCD$^+$14], which is supported by National Science Foundation Grant No. ACI-1053575 (XSEDE award DMR 140129) and on resources of the Oak Ridge Leadership Computing Facility which is a DOE Office of Science User Facility supported under Contract No. DE-AC05-00OR22725.

Author contributions

Conceptualization, B.D.D., B.L.B., V.R., A.T., M.M.H., H.O., and C.S.A.; data curation, B.D.D., B.L.B., V.R., A.T., M.M.H., H.O., and C.S.A.; funding acquisition, E.J. and S.C.G.; methodology, B.D.D., B.L.B., V.R., A.T., M.M.H., H.O., and C.S.A.; project administration, B.D.D., B.L.B., V.R., A.T., M.M.H., H.O., and C.S.A.; software, B.D.D., B.L.B., V.R., A.T., M.M.H., H.O., and C.S.A.; supervision, S.C.G.; visualization, B.D.D., B.L.B., A.T., and K.W.; writing – original draft, B.D.D., B.L.B., V.R., A.T., and H.O.; writing – review & editing, B.D.D., B.L.B., V.R., A.T., M.M.H., H.O., K.W., C.S.A., and S.C.G. All authors have read and agreed to the published version of the manuscript.

References

[AAB$^+$15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon

4. https://github.com/glotzerlab/signac
5. https://github.com/glotzerlab/signac-flow
6. https://docs.signac.io/

Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems. 2015. Software available from tensorflow.org. URL: https://www.tensorflow.org/.

[AADG18] Carl S. Adorf, James Antonaglia, Julia Dshemuchadse, and Sharon C. Glotzer. Inverse design of simple pair potentials for the self-assembly of complex structures. *The Journal of Chemical Physics*, 149(20):204102–204102, November 2018. doi:10.1063/1.5063802.

[ADRG18] Carl S. Adorf, Paul M. Dodd, Vyas Ramasubramani, and Sharon C. Glotzer. Simple data and workflow management with the signac framework. *Comput. Mater. Sci.*, 146(C):220–229, 2018. doi:10.1016/j.commatsci.2018.01.035.

[AGG] Joshua A. Anderson, Jens Glaser, and Sharon C. Glotzer. HOOMD-blue: A Python package for high-performance molecular dynamics and hard particle Monte Carlo simulations. 173:109363. doi:10.1016/j.commatsci.2019.109363.

[AMS+] Mark James Abraham, Teemu Murtola, Roland Schulz, Szilárd Páll, Jeremy C. Smith, Berk Hess, and Erik Lindahl. GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. 1-2:19–25. doi:10.1016/j.softx.2015.06.001.

[BLBVRJAASCG20] Brandon L. Butler, Vyas Ramasubramani, Joshua A. Anderson, and Sharon C. Glotzer. HOOMD-blue version 3.0 A Modern, Extensible, Flexible, Object-Oriented API for Molecular Simulations. In Meghann Agarwal, Chris Calloway, Dillon Niederhut, and David Shupe, editors, *Proceedings of the 19th Python in Science Conference*, pages 24–31, 2020. doi:10.25080/Majora-342d178e-004.

[C+15] François Chollet et al. Keras, 2015. URL: https://keras.io.

[CADG21] Rose K. Cersonsky, James Antonaglia, Bradley D. Dice, and Sharon C. Glotzer. The diversity of three-dimensional photonic crystals. *Nature Communications*, 12(1):2543, May 2021. doi:10.1038/s41467-021-22809-6.

[CMI+21] Peter T. Cummings, Clare McCabe, Christopher R. Iacovella, Akos Ledeczi, Eric Jankowski, Arthi Jayaraman, Jeremy C. Palmer, Edward J. Maginn, Sharon C. Glotzer, Joshua A. Anderson, J. Ilja Siepmann, Jeffrey Potoff, Ray A. Matsumoto, Justin B. Gilmer, Ryan S. DeFever, Ramanish Singh, and Brad Crawford. Open-source molecular modeling software in chemical engineering focusing on the Molecular Simulation Design Framework. *AIChE Journal*, 67(3):e17206, 2021. doi:10.1002/aic.17206.

[Col13] Andrew Collette. *Python and HDF5*. O'Reilly, 2013.

[Dav] Matt Davis. snakeviz. URL: https://jiffyclub.github.io/snakeviz/.

[DMD+21] Ryan S DeFever, Ray A Matsumoto, Alexander W Dowling, Peter T Cummings, and Edward J Maginn. Mosdef cassandra: A complete python interface for the cassandra monte carlo software. *Journal of Computational Chemistry*, 2021. doi:10.1002/jcc.24807.

[GBB+09] Paolo Giannozzi, Stefano Baroni, Nicola Bonini, Matteo Calandra, Roberto Car, Carlo Cavazzoni, Davide Ceresoli, Guido L Chiarotti, Matteo Cococcioni, Ismaila Dabo, Andrea Dal Corso, Stefano de Gironcoli, Stefano Fabris, Guido Fratesi, Ralph Gebauer, Uwe Gerstmann, Christos Gougoussis, Anton Kokalj, Michele Lazzeri, Layla Martin-Samos, Nicola Marzari, Francesco Mauri, Riccardo Mazzarello, Stefano Paolini, Alfredo Pasquarello, Lorenzo Paulatto, Carlo Sbraccia, Sandro Scandolo, Gabriele Sclauzero, Ari P Seitsonen, Alexander Smogunov, Paolo Umari, and Renata M Wentzcovitch. QUANTUM ESPRESSO: a modular and open-source software project for quantum simulations of materials. *Journal of Physics: Condensed Matter*, 21(39):395502, sep 2009. doi:10.1088/0953-8984/21/39/395502.

[GG18] Marco Govoni and Giulia Galli. Gw100: Comparison of methods and accuracy of results obtained with the west code. *Journal of Chemical Theory and Computation*, 14(4):1895–1909, 2018. doi:10.1021/acs.jctc.7b00952.

[GNA+] Jens Glaser, Trung Dac Nguyen, Joshua A. Anderson, Pak Lui, Filippo Spiga, Jaime A. Millan, David C. Morse, and Sharon C. Glotzer. Strong scaling of general-purpose molecular dynamics simulations on GPUs. 192:97–107. doi:10.1016/j.cpc.2015.02.028.

[Gro21] The HDF Group. Hierarchical data format, version 5, 1997-2021. URL: https://www.hdfgroup.org/HDF5/.

[HCVM20] Eric S. Harper, Eleanor J. Coyle, Jonathan P. Vernon, and Matthew S. Mills. Inverse design of broadband highly reflective metasurfaces using neural networks. *Physical Review B*, 101(19):195104, May 2020. doi:10.1103/PhysRevB.101.195104.

[HKvdSL] Berk Hess, Carsten Kutzner, David van der Spoel, and Erik Lindahl. GROMACS 4: Algorithms for Highly Efficient, Load-Balanced, and Scalable Molecular Simulation. 4(3):435–447. doi:10.1021/ct700301q.

[HSS08] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11–15, Pasadena, CA USA, 2008.

[JJ01] Steven G. Johnson and J. D. Joannopoulos. Block-iterative frequency-domain methods for maxwell's equations in a planewave basis. *Opt. Express*, 8(3):173–190, Jan 2001. doi:10.1364/OE.8.000173.

[KF96] G. Kresse and J. Furthmüller. Efficient iterative schemes for ab initio total-energy calculations using a plane-wave basis set. *Phys. Rev. B*, 54:11169–11186, Oct 1996. doi:10.1103/PhysRevB.54.11169.

[KRKP+16] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. Jupyter notebooks – a publishing format for reproducible computational workflows. In F. Loizides and B. Schmidt, editors, *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87–90. IOS Press, 2016.

[KSJ+] Christoph Klein, János Sallai, Trevor J. Jones, Christopher R. Iacovella, Clare McCabe, and Peter T. Cummings. A Hierarchical, Component Based Approach to Screening Properties of Soft Matter. In Randall Q Snurr, Claire S. Adjiman, and David A. Kofke, editors, *Foundations of Molecular Modeling and Simulation: Select Papers from FOMMS 2015*, Molecular Modeling and Simulation, pages 79–92. Springer. doi:10.1007/978-981-10-1128-3_5.

[KST+] Christoph Klein, Andrew Z. Summers, Matthew W. Thompson, Justin B. Gilmer, Clare McCabe, Peter T. Cummings, Janos Sallai, and Christopher R. Iacovella. Formalizing atom-typing and the dissemination of force fields with foyer. 167:215–227. doi:10.1016/j.commatsci.2019.05.026.

[KWB+18] Anna Krylov, Theresa L. Windus, Taylor Barnes, Eliseo Marin-Rimoldi, Jessica A. Nash, Benjamin Pritchard, Daniel G. A. Smith, Doaa Altarawy, Paul Saxe, Cecilia Clementi, T. Daniel Crawford, Robert J. Harrison, Shantenu Jha, Vijay S. Pande, and Teresa Head-Gordon. Perspective: Computational chemistry software and its advancement as illustrated through three grand challenge cases for molecular science. *The Journal of Chemical Physics*, 149(18):180901, 2018. doi:10.1063/1.5052551.

[LF12] Victor Liu and Shanhui Fan. S4 : A free electromagnetic solver for layered periodic structures. *Computer Physics Communications*, 183(10):2233–2244, 2012. doi:10.1016/j.cpc.2012.04.026.

[LHvdS] Erik Lindahl, Berk Hess, and David van der Spoel.

GROMACS 3.0: A package for molecular simulation and trajectory analysis. 7(8):306–317. doi:10.1007/s008940100045.

[LPS] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, pages 1–6. Association for Computing Machinery. doi:10.1145/2833157.2833162.

[MADWB11] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. Mdanalysis: A toolkit for the analysis of molecular dynamics simulations. *Journal of Computational Chemistry*, 32(10):2319–2327, 7 2011. doi:10.1002/jcc.21787.

[MBH+15] Robert T. McGibbon, Kyle A. Beauchamp, Matthew P. Harrigan, Christoph Klein, Jason M. Swails, Carlos X. Hernández, Christian R. Schwantes, Lee-Ping Wang, Thomas J. Lane, and Vijay S. Pande. Mdtraj: A modern open library for the analysis of molecular dynamics trajectories. *Biophysical Journal*, 109(8):1528–1532, 2015. doi:10.1016/j.bpj.2015.08.015.

[McK] Wes McKinney. Data Structures for Statistical Computing in Python. pages 56–61. doi:10.25080/Majora-92bf1922-00a.

[MjD+20] Alistair Miles, jakirkham, Martin Durant, Matthias Bussonnier, James Bourbeau, Tarik Onalan, Joe Hamman, Zain Patel, Matthew Rocklin, shikharsg, Ryan Abernathey, Josh Moore, Vincent Schut, raphael dussin, Elliott Sales de Andrade, Charles Noyes, Aleksandar Jelenak, Anderson Banihirwe, Chris Barnes, George Sakkis, Jan Funke, Jerome Kelleher, Joe Jevnik, Justin Swaney, Poruri Sai Rahul, Stephan Saalfeld, john, Tommy Tran, pyup.io bot, and sbalmer. zarr-developers/zarr-python: v2.5.0, October 2020. doi:10.5281/zenodo.4069231.

[MVG+21] Félix Musil, Max Veit, Alexander Goscinski, Guillaume Fraux, Michael J. Willatt, Markus Stricker, Till Junge, and Michele Ceriotti. Efficient implementation of atom-density representations. *The Journal of Chemical Physics*, 154(11):114109, March 2021. doi:10.1063/5.0044689.

[pdt20] The pandas development team. pandas-dev/pandas: Pandas, February 2020. doi:10.5281/zenodo.3509134.

[PHMTN19] Michael P. Howard, Thomas M. Truskett, and Arash Nikoubashman. Cross-stream migration of a Brownian droplet in a polymer solution under Poiseuille flow. *Soft Matter*, 15(15):3168–3178, 2019. doi:10.1039/C8SM02552E.

[PPS+] Sander Pronk, Szilárd Páll, Roland Schulz, Per Larsson, Pär Bjelkmar, Rossen Apostolov, Michael R. Shirts, Jeremy C. Smith, Peter M. Kasson, David van der Spoel, Berk Hess, and Erik Lindahl. GROMACS 4.5: A high-throughput and highly parallel open source molecular simulation toolkit. 29(7):845–854. doi:10.1093/bioinformatics/btt055.

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[RAD+18] Vyas Ramasubramani, Carl S. Adorf, Paul M. Dodd, Bradley D. Dice, and Sharon C. Glotzer. signac: A python framework for data and workflow management. pages 152–159, 2018. doi:10.25080/Majora-4af1f417-016.

[RD20] David Rodziewicz and Jacob Dice. Drought Risk to the Agriculture Sector. *The Federal Reserve Bank of Kansas City Economic Review*, December 2020. doi:10.18651/ER/v105n2RodziewiczDice.

[RDH+20] Vyas Ramasubramani, Bradley D. Dice, Eric S. Harper, Matthew P. Spellings, Joshua A. Anderson, and Sharon C. Glotzer. freud: A software suite for high throughput analysis of particle simulation data. *Computer Physics Communications*, 254:107275, 2020. doi:10.1016/j.cpc.2020.107275.

[Roc15] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In Kathryn Huff and James Bergstra, editors, *Proceedings of the 14th Python in Science Conference*, pages 130–136, 2015. doi:10.25080/Majora-7b98e3ed-013.

[SMRM+17] Jindal K Shah, Eliseo Marin-Rimoldi, Ryan Gotchy Mullen, Brian P Keene, Sandip Khan, Andrew S Paluch, Neeraj Rai, Lucienne L Romanielo, Thomas W Rosch, Brian Yoo, et al. Cassandra: An open source monte carlo package for molecular simulation, 2017. doi:10.1002/jcc.26544.

[Stu] Alexander Stukowski. Visualization and analysis of atomistic simulation data with OVITO–the Open Visualization Tool. 18(1):015012. doi:10.1088/0965-0393/18/1/015012.

[TAH+18] Stephen Thomas, Monet Alberts, Michael M Henry, Carla E Estridge, and Eric Jankowski. Routine million-particle simulations of epoxy curing with dissipative particle dynamics. *Journal of Theoretical and Computational Chemistry*, 17(03):1840005, April 2018. doi:10.1142/S0219633618400059.

[TCD+14] John Towns, Timothy Cockerill, Maytal Dahan, Ian Foster, Kelly Gaither, Andrew Grimshaw, Victor Hazlewood, Scott Lathrop, Dave Lifka, Gregory D. Peterson, Ralph Roskies, J. Ray Scott, and Nancy Wilkins-Diehr. Xsede: Accelerating scientific discovery. *Computing in Science Engineering*, 16(5):62–74, 2014. doi:10.1109/MCSE.2014.80.

[Tea16] Dask Development Team. *Dask: Library for dynamic task scheduling*, 2016. URL: https://dask.org.

[Tea18] RAPIDS Development Team. *RAPIDS: Collection of Libraries for End to End GPU Data Science*, 2018. URL: https://rapids.ai.

[TGM+20] Matthew W. Thompson, Justin B. Gilmer, Ray A. Matsumoto, Co D. Quach, Parashara Shamaprasad, Alexander H. Yang, Christopher R. Iacovella, Clare McCabe, and Peter T. Cummings. Towards molecular simulations that are transparent, reproducible, usable by others, and extensible (TRUE). *Molecular Physics*, 118(9-10):e1742938, June 2020. doi:10.1080/00268976.2020.1742938.

[TMS+] Matthew W. Thompson, Ray Matsumoto, Robert L. Sacci, Nicolette C. Sanders, and Peter T. Cummings. Scalable Screening of Soft Matter: A Case Study of Mixtures of Ionic Liquids and Organic Solvents. 123(6):1340–1347. doi:10.1021/acs.jpcb.8b11527.

[WDC18] Nancy Wilkins-Diehr and T. Daniel Crawford. Nsf's inaugural software institutes: The science gateways community institute and the molecular sciences software institute. *Computing in Science Engineering*, 20(5):26–38, 2018. doi:10.1109/MCSE.2018.05329813.