The Journal of Open Source Software

Jobflow: Computational Workflows Made Simple

Andrew S. Rosen [1,2], **Max Gallant** [1,2], **Janine George** [3,4], **Janosh Riebesell** [2,5], **Hrushikesh Sahasrabuddhe** [1,6], **Jimmy-Xuan Shen** [7], **Mingjian Wen** [8], **Matthew L. Evans** [9,10], **Guido Petretto**[9], **David Waroquiers** [9,10], **Gian-Marco Rignanese** [9,10,11], **Kristin A. Persson** [1,2,12], **Anubhav Jain** [6], **and Alex M. Ganose** [13]

1 Department of Materials Science and Engineering, University of California, Berkeley, Berkeley, CA, USA **2** Materials Science Division, Lawrence Berkeley National Laboratory, Berkeley, CA, USA **3** Federal Institute for Materials Research and Testing, Department Materials Chemistry, Berlin, Germany **4** Friedrich Schiller University Jena, Institute of Condensed Matter Theory and Solid-State Optics, Jena, Germany **5** Department of Physics, University of Cambridge, Cambridge, UK **6** Energy Storage and Distributed Resources Division, Lawrence Berkeley National Laboratory, Berkeley, CA, USA **7** Materials Science Division, Lawrence Livermore National Laboratory, Livermore, CA, USA **8** William A. Brookshire Department of Chemical and Biomolecular Engineering, University of Houston, Houston, TX, USA **9** Matgenix SRL, rue Armand Bury 185, 6534 Gozée, Belgium **10** Institut de la Matière Condensée et des Nanosciences, Université catholique de Louvain, Chemin des Étoiles 8, Louvain-la-Neuve 1348, Belgium **11** School of Materials Science and Engineering, Northwestern Polytechnical University, No. 127 Youyi West Road, Xi'an 710072 Shaanxi, PR China **12** Molecular Foundry, Lawrence Berkeley National Laboratory, Berkeley, CA, USA **13** Department of Chemistry, Imperial College London, London, UK

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor: Open Journals

Reviewers:

- @openjournals

Submitted: 01 January 1970
Published: unpublished

Summary

We present Jobflow, a domain-agnostic Python package for writing computational workflows tailored for high-throughput computing applications. With its simple decorator-based approach, functions and class methods can be transformed into compute jobs that can be stitched together into complex workflows. Jobflow fully supports dynamic workflows where the full acyclic graph of compute jobs is not known until runtime, such as compute jobs that launch other jobs based on the results of previous steps in the workflow. The results of all Jobflow compute jobs can be easily stored in a variety of filesystem- and cloud-based databases without the data storage process being part of the underlying workflow logic itself. Jobflow has been intentionally designed to be fully independent of the choice of workflow manager used to dispatch the calculations on remote computing resources. At the time of writing, Jobflow workflows can be executed either locally or across distributed compute environments via an adapter to the FireWorks package, and Jobflow fully supports the integration of additional workflow execution adapters in the future.

Statement of Need

The current era of big data and high-performance computing has emphasized the significant need for robust, flexible, and scalable workflow management solutions that can be used to efficiently orchestrate scientific calculations (Ben-Nun et al., 2020; Silva et al., 2023). To date, a wide variety of workflow systems have been developed, and it has become clear that there is no one-size-fits-all solution due to the diverse needs of the computational community (Al-Saadi et al., 2021; *Existing Workflow Systems*, n.d.). While several popular software packages in this space have emerged over the last decade, many of them require the user to tailor their domain-specific code with the underlying workflow management framework closely in mind. This can be a barrier to entry for many users and puts significant constraints on the portability of the underlying workflows.

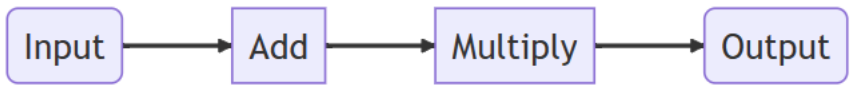
Here, we introduce Jobflow: a free, open-source Python library that makes it simple to transform collections of functions into complex workflows that can be executed either locally or across distributed computing environments. Jobflow has been intentionally designed to act as middleware between the user's domain-specific routines that they wish to execute and the workflow "manager" that ultimately orchestrates the calculations across different computing environments. Jobflow uses a simple decorator-based syntax that is similar to that of other recently developed workflow tools (Babuji et al., 2019; *Covalent*, 2023; *Prefect*, n.d.; *Redun*, 2023). This approach makes it possible to turn virtually any function into a Jobflow `Job` instance (i.e., a discrete unit of work) with minimal changes to the underlying code itself.

Jobflow has grown out of a need to carry out high-throughput computational materials science workflows at scale as part of the Materials Project (Jain et al., 2013). As the kinds of calculations — from *ab initio* to semi-empirical to those based on machine learning — continue to evolve and the resulting data streams continue to diversify, it was necessary to rethink how we managed an increasingly diverse range of computational workflows. Going forward, Jobflow will become the computational backbone of the Materials Project, which we hope will inspire additional confidence in the readiness of Jobflow for production-quality scientific computing applications.

Features and Implementation

Overview

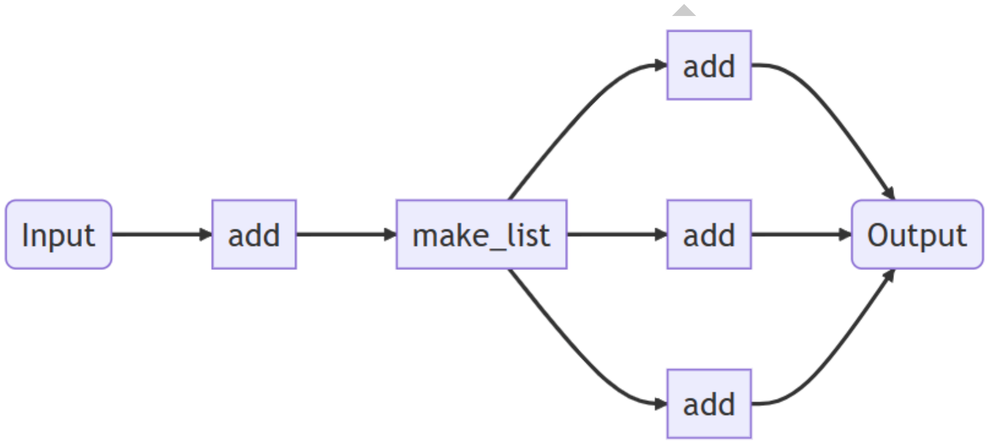
As a simple demonstration, the example below shows how one can construct a simple `Flow` (i.e., a graph of interdependent `Job`s) composed of two sequential `Job`s: the first `Job` adds two numbers together (`1 + 2`), and the second `Job` multiplies the result by another number (`3 * 3`). For simplicity, this `Flow` is executed locally, but it can be easily dispatched to a remote computing environment via the selection of a different workflow manager with no modifications to the underlying function definitions. While trivial, this example demonstrates the simplicity of the Jobflow syntax and how it can be used to construct complex workflows from user-defined functions. Note that each `Job` object is not run when instantiated; rather, an `OutputReference` associated with the `Job` (presented to the user via a Universally Unique Identifier, or UUID) is returned, which is resolved when the workflow is ultimately executed.



```python
from jobflow import Flow, job, run_locally

@job
def add(a, b):
    return a + b

@job
def multiply(a, b):
    return a * b

job1 = add(1, 2)  # 1 + 2 = 3
job2 = multiply(job1.output, 3)  # 3 * 3 = 9
flow = Flow([job1, job2])

responses = run_locally(flow)
```

Dynamic Workflows

Beyond the typical acyclic graph of jobs, Jobflow fully supports dynamic workflows where the precise number of jobs is unknown until runtime. This is a particularly common requirement in chemistry and materials science workflows and is made possible through the use of a `Response` object that controls the flow execution. For instance, the example below is a `Flow` that will add two numbers (`1 + 2`), construct a list of random length containing the prior result (e.g. `[3, 3, 3]`), and then add an integer to each element of the list (`[3 + 10, 3 + 10, 3 + 10]`). The `Response(replace=Flow(jobs))` syntax tells Jobflow to replace the current `Job` with a (sub)workflow after the `Job` completes.



```python
from random import randint
from jobflow import Flow, Response, job, run_locally

@job
def add(a, b):
    return a + b

@job
def make_list(val):
    return [val] * randint(2, 6)

@job
def add_distributed(vals, c):
    jobs = [add(val, c) for val in vals]
    return Response(replace=Flow(jobs))

job1 = add(1, 2)  # 1 + 2 = 3
job2 = make_list(job1.output)  # e.g., [3, 3, 3]
job3 = add_distributed(job2.output, 10)  # [3 + 10, 3 + 10, 3 + 10]
flow = Flow([job1, job2, job3])

responses = run_locally(flow)
```

Data Management

Jobflow has first-class support for a variety of data stores through an interface with the `maggma` Python package (*Maggma*, 2023). This makes it possible to easily store the results of workflows in a manner that is independent of the choice of storage medium and that is entirely decoupled from the workflow logic itself. Additionally, it is possible within Jobflow to specify multiple types of data stores for specific Python objects (e.g., primitive types vs. large binary blobs) created by a given workflow, which is often useful for storing a combination of metadata (e.g., in a NoSQL database like MongoDB or file-system based store like MontyDB (MontyDB, 2023)') and raw data (e.g., in a cloud object store like Amazon S3 or Microsoft Azure).

Promoting Code Reuse

Unlike most workflow solutions that rely on the use of functional programming, Jobflow fully supports and encourages the use of compute jobs that involve class-based inheritance to reduce duplication of code. While subtle, this oft-overlooked feature is particularly useful for scientific workflows where very similar calculations need to be carried out but with slightly different parameters or implementation details.

In particular, Jobflow has an abstract class called a `Maker` that makes it convenient to define a class that can return a `Job` to be executed. This makes it possible to take advantage of the benefits of object-oriented programming while still being able to use the straightforward Jobflow decorator syntax. The support for classes also avoids the need for each workflow to accept a large number of keyword arguments in order to give the user freedom to modify the behavior of any constituent `Job` in the `Flow`. Instead, the class variables of the `Maker` can be updated directly, and these changes will be reflected in the `Job` at runtime, as demonstrated in the example below. Inheriting from the `Maker` class also enables updating the parameters of specific `Jobs` in a `Flow` through a convenient `Flow.update_maker_kwargs(...)` function, which allows for easy customization of workflows even after the `Jobs` have been defined.

```python
from dataclasses import dataclass
from jobflow import job, Flow, Maker
from jobflow.managers.local import run_locally

@dataclass
class ExponentiateMaker(Maker):
    name: str = "Exponentiate"
    exponent: int = 2

    @job
    def make(self, a):
        return a**self.exponent

job1 = ExponentiateMaker().make(a=2)  # 2**2 = 4
job2 = ExponentiateMaker(exponent=3).make(job1.output)  # 4**3 = 64
flow = Flow([job1, job2])

responses = run_locally(flow)
```

Workflow Execution

One of the major benefits of Jobflow is that it decouples the details related to workflow execution from the workflow definitions themselves. The simplest way to execute a workflow is to run it directly on the machine where the workflow is defined using the `run_locally(...)` function, as shown in the examples above. This makes it possible to quickly test even complex workflows without the need to rely on a database or configuring remote resources.

When deploying production calculations, workflows often need to be dispatched to large supercomputers through a remote execution engine. Jobflow has an interface with the FireWorks package (Jain et al., 2015) via a one-line command to convert a `Flow` and its underlying `Job` objects into the analogous FireWorks `Workflow` and `Firework` objects that enable execution on high-performance computing machines. The logic behind the `Job` and `Flow` objects are not tied to FireWorks in any direct way, such that the two packages are fully decoupled.

Additionally, a remote mode of execution built solely around Jobflow is currently under active development. With this approach, workflows can be executed across multiple "workers" (e.g., a simple computer, a supercomputer, or a cloud-based service) and managed through a modern command-line interface without relying on an external workflow execution engine. The forthcoming Jobflow remote mode of execution has been designed such that no inbound connection from the workers to the database of jobs and results is needed, thus ensuring data and network security for professional usage.

More generally, it is possible for users to develop custom "adapter" interfaces to their personal workflow execution engine of choice. As a result, Jobflow fills a niche in the broader workflow community and can help make the same workflow definition interoperable across multiple workflow execution engines.

Testing and Documentation

Jobflow has been designed with robustness in mind. The Jobflow codebase has 100% test coverage at the time of writing and is fully documented. The detailed testing suite, along with continuous integration pipelines on GitHub, makes it easy for users to write their own workflows with confidence that they will continue to work as expected for the foreseeable future. Furthermore, the ability to run Jobflow `Flow` objects locally makes it simple to write unit tests when designing a new Python package built around Jobflow without the need for complex monkey-patching or spinning up a test server.

Usage To-Date

While domain-agnostic, Jobflow has been used in several materials science Python packages to date, including but not limited to:

- Atomate2 (*Atomate2*, 2023), Quacc (*Quacc – the Quantum Accelerator*, 2023): Libraries of computational chemistry and materials science workflows.
- NanoParticleTools (*NanoParticleTools*, 2023): Workflows for Monte Carlo simulations of nanoparticles.
- Reaction Network (McDermott et al., 2021; *Reaction Network*, 2023): Workflows for constructing and analyzing inorganic chemical reaction networks.
- WFacer (*WFacer*, 2023): Workflows for modeling the statistical thermodynamics of solids via automated cluster expansion.

Additional Details

Naturally, the summary presented in this article constitutes only a small subset of the features that Jobflow has to offer. For additional details along with helpful tutorials ranging from basic applications to examples specifically targeting the computational materials science community, we refer the reader to the Jobflow documentation. Suggestions, contributions, and bug reports are always welcome.

Acknowledgements

This work was primarily funded and intellectually led by the Materials Project, which is funded by the U.S. Department of Energy, Office of Science, Office of Basic Energy Sciences, Materials Sciences and Engineering Division, under Contract no. DE-AC02-05-CH11231: Materials Project program KC23MP. A.S.R. acknowledges support via a Miller Research Fellowship from the Miller Institute for Basic Research in Science, University of California, Berkeley. J.G would like to acknowledge the Gauss Centre for Supercomputing e.V. (www.gauss-centre.eu) for funding workflow-related developments by providing generous computing time on the

GCS Supercomputer SuperMUC-NG at Leibniz Supercomputing Centre (www.lrz.de) (Project pn73da). J.R. acknowledges support from the German Academic Scholarship Foundation (Studienstiftung). M.L.E. thanks the BEWARE scheme of the Wallonia-Brussels Federation for funding under the European Commission's Marie Curie-Skłodowska Action (COFUND 847587). G.P. and D.W. acknowledge Umicore for the financial support in developing the remote execution mode of jobflow. D.W. and G.M.R. acknowledge funding from the European Union's Horizon 2020 research and innovation program under the grant agreement No 951786 (NOMAD CoE). A.M.G. is supported by EPSRC Fellowship EP/T033231/1.

References

Al-Saadi, A., Ahn, D. H., Babuji, Y., Chard, K., Corbett, J., Hategan, M., Herbein, S., Jha, S., Laney, D., Merzky, A., & others. (2021). Exaworks: Workflows for exascale. *2021 IEEE Workshop on Workflows in Support of Large-Scale Science (WORKS)*, 50–57. https://doi.org/10.1109/works54523.2021.00012

Atomate2. (2023). https://github.com/materialsproject/atomate2

Babuji, Y., Woodard, A., Li, Z., Katz, D. S., Clifford, B., Kumar, R., Lacinski, L., Chard, R., Wozniak, J. M., Foster, I., & others. (2019). Parsl: Pervasive parallel programming in python. *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, 25–36.

Ben-Nun, T., Gamblin, T., Hollman, D. S., Krishnan, H., & Newburn, C. J. (2020). Workflows are the new applications: Challenges in performance, portability, and productivity. *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 57–69. https://doi.org/10.1109/p3hpc51967.2020.00011

Covalent. (2023). https://doi.org/10.5281/zenodo.5903364

Existing workflow systems. (n.d.). https://s.apache.org/existing-workflow-systems

Jain, A., Ong, S. P., Chen, W., Medasani, B., Qu, X., Kocher, M., Brafman, M., Petretto, G., Rignanese, G.-M., Hautier, G., & others. (2015). FireWorks: A dynamic workflow system designed for high-throughput applications. *Concurrency and Computation: Practice and Experience*, *27*(17), 5037–5059. https://doi.org/10.1002/cpe.3505

Jain, A., Ong, S. P., Hautier, G., Chen, W., Richards, W. D., Dacek, S., Cholia, S., Gunter, D., Skinner, D., Ceder, G., & others. (2013). Commentary: The materials project: A materials genome approach to accelerating materials innovation. *APL Materials*, *1*(1). https://doi.org/10.1063/1.4812323

Maggma. (2023). https://github.com/materialsproject/maggma

McDermott, M. J., Dwaraknath, S. S., & Persson, K. A. (2021). A graph-based network for predicting chemical reaction pathways in solid-state materials synthesis. *Nature Communications*, *12*(1), 3097. https://doi.org/10.1038/s41467-021-23339-x

MontyDB. (2023). https://github.com/davidlatwe/montydb

NanoParticleTools. (2023). https://github.com/BlauGroup/NanoParticleTools

Prefect. (n.d.). https://github.com/PrefectHQ/prefect

Quacc – the quantum accelerator. (2023). https://doi.org/10.5281/zenodo.7720998

Reaction network. (2023). https://github.com/materialsproject/reaction-network

Redun. (2023). https://github.com/insitro/redun

Silva, R. F. da, Badia, R. M., Bala, V., Bard, D., Bremer, P.-T., Buckley, I., Caino-Lores, S., Chard, K., Goble, C., Jha, S., & others. (2023). Workflows community summit 2022: A roadmap revolution. *arXiv Preprint arXiv:2304.00019*.

WFacer. (2023). https://github.com/CederGroupHub/WFacer