Monitoring Scientific Python Usage on a Supercomputer

Rollin Thomas‡*, Laurie Stephey‡*, Annette Greiner‡, Brandon Cook‡

Abstract—In 2021, more than 30% of users at the National Energy Research Scientific Computing Center (NERSC) used Python on the Cori supercomputer. To determine this we have developed and open-sourced a simple, minimally invasive monitoring framework that leverages standard Python features to capture Python imports and other job data via a package called "Customs". To analyze the data we collect via Customs, we have developed a Jupyter-based analysis framework designed to be interactive, shareable, extensible, and publishable via a dashboard. Our stack includes Papermill to execute parameterized notebooks, Dask-cuDF for multi-GPU processing, and Voila to render our notebooks as web-based dashboards. We report preliminary findings from Customs data collection and analysis. This work demonstrates that our monitoring framework can capture insightful and actionable data including top Python libraries, preferred user software stacks, and correlated libraries, leading to a better understanding of user behavior and affording us opportunity to make increasingly data-driven decisions regarding Python at NERSC.

Index Terms—HPC, Python monitoring, GPUs, dashboards, parallel, Jupyter

Introduction

The National Energy Research Scientific Computing Center (NERSC) is the primary scientific computing facility for the US Department of Energy's Office of Science. Some 8,000 scientists use NERSC to perform basic, non-classified research in predicting novel materials, modeling the Earth's climate, understanding the evolution of the Universe, analyzing experimental particle physics data, investigating protein structure, and more [OA20]. NERSC procures and operates supercomputers and massive storage systems under a strategy of balanced, timely introduction of new hardware and software technologies to benefit the broadest possible portion of this workload. While procuring new systems or supporting users of existing ones, NERSC relies on detailed analysis of its workload to help inform strategy.

Workload analysis is the process of collecting and marshaling data to build a picture of how applications and users really interact with and utilize systems. It is one part of a procurement strategy that also includes surveys of user and application requirements, emerging computer science research, developer or vendor roadmaps, and technology trends. Understanding our workload helps us engage in an informed way with stakeholders like funding agencies, vendors, developers, users, standards bodies, and other high-performance computing (HPC) centers. In particular, workload analysis informs non-recurring engineering contracts where NERSC partners with external software developers to address gaps in system programming environments. Actively monitoring the workload also enables us to identify suboptimal or potentially problematic user practices and address them through direct intervention, improving documentation, or simply making it easier for users to use the software better. Measuring the relative frequency of use of different software components can help us streamline delivery, retiring less-utilized packages, and promoting timely migration to newer versions. Detecting and analyzing trends in user behavior with software over time also helps us anticipate user needs and prepare accordingly. Comprehensive, quantitative workload analysis is a critical tool in keeping NERSC a productive supercomputer center for science.

With Python assuming a key role in scientific computing, it makes sense to apply workload analysis to Python in production settings like NERSC's Cray XC-40 supercomputer, Cori. Once viewed in HPC circles as merely a cleaner alternative to Perl or Shell scripting, Python has evolved into a robust platform for orchestrating simulations, running complex data processing pipelines, managing artificial intelligence workflows, visualizing massive data sets, and more. Adapting workload analysis practices to scientific Python gives its community the same data-driven leverage that other language communities at NERSC already enjoy.

This article documents NERSC's Python workload analysis efforts, part of an initiative called Monitoring of Data Services (MODS) [MODS], and what we have learned during this process. In the next section, we provide an overview of related work including existing tools for workload data collection, management, and analysis. In Methods, we describe an approach to Python-centric workload analysis that uses built-in Python features to capture usage data, and a Jupyter notebook-based workflow for exploring the data set and communicating what we discover. Our Results include high-level statements about what Python packages are used most often and at what scale on Cori, but also some interesting deeper dives into use of certain specific packages along with a few surprises. In the Discussion, we examine the implications of our results, share the strengths and weaknesses of our workflow and our lessons learned, and outline plans for improving the analysis to better fill out the picture of Python at NERSC. The Conclusion suggests some areas for future work.

* *Corresponding author: rcthomas@lbl.gov, lastephey@lbl.gov*
‡ *National Energy Research Scientific Computing Center, Lawrence Berkeley National Laboratory, 1 Cyclotron Road MS59-4010A, Berkeley, California, 94720*
* *Corresponding author: rcthomas@lbl.gov, lastephey@lbl.gov*

Related Work

The simplest approach used to get a sense of what applications run on a supercomputer is to scan submitted batch job scripts for executable names. In the case of Python applications, this is problematic since users often invoke Python scripts directly instead of as an argument to the `python` executable. This method also provides only a crude count of Python invocations and gives little insight into deeper questions about specific Python packages.

Software environment modules [Fur91] are a common way for HPC centers to deliver software to users. Environment modules operate primarily by setting, modifying, or deleting environment variables upon invocation of a module command (e.g. `module load`, `module swap`, or `module unload`) This provides an entrypoint for software usage monitoring. Staff can inject code into a module load operation to record the name of the module being loaded, its version, and other information about the user's environment. Lmod, a newer implementation of environment modules [Mcl11], provides documentation on how to configure it to use syslog and MySQL to collect module loads through a hook function. Counting module loads as a way to track Python usage has the virtue of simplicity. However, users often include module load commands in their shell resource files (e.g., *.bashrc*), meaning that user login or shell invocation may trigger a detection even if the user never actually uses the trigger module. Furthermore, capturing information at the package level using module load counts would also require that individual Python packages be installed as separate environment modules. Module load counts also miss Python usage from user-installed Python environments or in containers.

Tools like ALTD [Fah10] and XALT [Agr14] are commonly used in HPC contexts to track library usage in compiled applications. The approach is to introduce wrappers that intercept the linker and batch job launcher (e.g. `srun` in the case of Slurm used at NERSC). The linker wrapper can inject metadata into the executable header, take a census of libraries being linked in, and forward that information to a file or database for subsequent analysis. Information stored in the header at link time is dumped and forwarded later by the job launch wrapper. On systems where all user applications are linked and launched with instrumented wrappers, this approach yields a great deal of actionable information to HPC center staff. However, popular Python distributions such as Anaconda Python arrive on systems fully built, and can be installed by users without assistance from center staff. Later versions of XALT can address this through an `LD_PRELOAD` setting. This enables XALT to identify compiled extensions that are imported in Python programs using a non-instrumented Python, but pure Python libraries currently are not detected. XALT is an active project so this may be addressed in a future release.

[Mac17] describes an approach to monitoring Python package use on Blue Waters using only built-in Python features: `sitecustomize` and `atexit`. During normal Python interpreter start-up, an attempt is made to import a module named `sitecustomize` that is intended to perform site-specific customizations. In this case, the injected code registers an exit handler through the `atexit` standard library module. This exit handler inspects `sys.modules`, a dictionary that normally describes all packages imported in the course of execution. On Blue Waters, `sitecustomize` was installed into the Python distribution installed and maintained by staff. Collected information was stored to plain text log files. An advantage of this approach is that

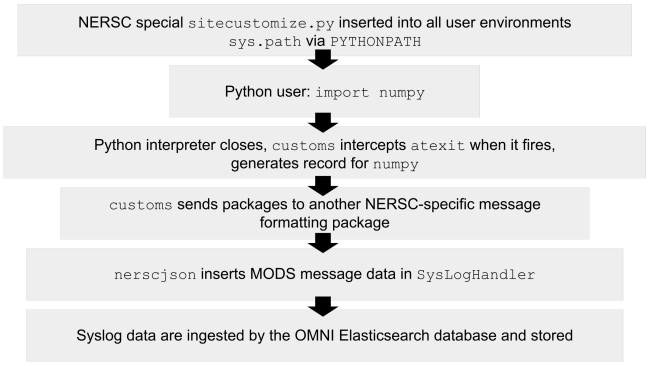

Fig. 1: *NERSC infrastructure for capturing Python usage data.*

`sitecustomize` failures are nonfatal, and placing the import reporting step into an exit hook (as opposed to instrumenting the import mechanism) means that it minimizes interference with normal operation of the host application. The major limitation of this strategy is that abnormal process terminations prevent the Python interpreter from proceeding through its normal exit sequence and package import data are not recorded.

Of course, much more information may be available through tools based on the extended Berkeley Packet Filter and the BPF compiler collection, similar to the `pythoncalls` utility that summarizes method calls in a running application. While eBPF overheads are very small, this approach requires special compilation flags for Python and libraries. Effort would be needed to make the monitoring more transparent to users and to marshal the generated data for subsequent analysis. This could be an interesting and fruitful approach to consider. Obviously, solutions that can overly impact application reliability or place an undue burden on system administrators and operations staff should be avoided. The fullest picture we currently can obtain comes from a combination of non-intrusive tooling and follow-up with users, using the story we can put together from the data we gather as a starting point for conversation.

Methods

Users have a number of options when it comes to how they use Python at NERSC. NERSC provides a "default" Python to its users through a software environment module, based on the Anaconda Python distribution with modifications. Users may load this module, initialize the Conda tool, and create their own custom Conda environments. Projects or collaborations may provide their users with shared Python environments, often as a Conda environment or as an independent installation altogether (e.g. using the Miniconda installer and building up). Cray provides a basic "Cray Python" module containing a few core scientific Python packages linked against Cray MPICH and LibSci libraries. Python packages are also installed by staff or users via Spack [Gam15], an HPC package manager. NERSC also provides Shifter [Jac16], a container runtime that enables users to run custom Docker containers that can contain Python built however the author desires. With a properly defined kernel-spec file, a user is able to use a Python environment based on any of the above options as a kernel in NERSC's Jupyter service. The goal is to gather data for workload analysis across all of these options.

Monitoring all of the above can be done quite easily by using the strategy outlined in [Mac17] with certain changes. Fig. 1 illustrates the infrastructure we have configured. As in [Mac17] a `sitecustomize` that registers the `atexit` handler is installed in a directory included into all users' Python `sys.path`. The `sitecustomize` module is installed directly on each compute node and not served over network, in order to avoid exacerbating poor performance of Python start-up at scale. We accomplish this by installing it and any associated Python modules into the node system images themselves, and configuring default user environments to include a `PYTHONPATH` setting that injects `sitecustomize` into `sys.path`. Shifter containers include the monitoring packages from the system image via runtime volume mount. Users can opt out of monitoring simply by unsetting or overwriting `PYTHONPATH`. We took the approach of provisioning a system-wide `PYTHONPATH` because we cast a much wider collection net (opt-out) than if we depend on users to install `sitecustomize` (opt-in). This also gives us a centrally managed source of truth for what is monitored at any given time.

Customs: Inspect and Report Packages

To organize `sitecustomize` logic we have created a Python package we call Customs, since it is for inspecting and reporting on Python package imports of particular interest. Customs can be understood in terms of three simple concepts. A **Check** is a simple object that represents a Python package by its name and a callable that is used to verify that the package (or even a specific module within a package) is present in a given dictionary. In production this dictionary should be `sys.modules` but during testing it can be a mock `sys.modules` dictionary. The **Inspector** is a container of Check objects, and is responsible for applying each Check to `sys.modules` (or mock) and returning the names of packages that are detected. Finally, the **Reporter** is an abstract class that takes some action given a list of detected package names. The Reporter action should be to record or transmit the list of detected packages, but exactly how this is done depends on implementation. Customs includes a few reference Reporter implementations and an example of a custom Customs Reporter.

Customs provides an entry point to use in `sitecustomize`, the function `register_exit_hook`. This function takes two arguments. The first is a list of strings or (string, callable) tuples that are converted into Checks. The second argument is the type of Reporter to be used. The exit hook can be registered multiple times with different package specification lists or Reporters if desired.

The intended workflow is that a staff member creates a list of package specifications they want to check for, selects or implements an appropriate Reporter, and passes these two objects to `register_exit_hook` within `sitecustomize.py`. Installing `sitecustomize` to system images generally involves packaging the software as an RPM to be installed into node system images and deployed by system administrators. When a user invokes Python, the exit hook will be registered using the `atexit` standard library module, the application proceeds as normal, and then at normal shutdown `sys.modules` is inspected and detected packages of interest are reported.

Message Logging and Storage

NERSC has developed a lightweight abstraction layer for message logging called nerscjson. It is a simple Python package that consumes JSON messages and forwards them to an appropriate transport layer that connects to NERSC's Operations Monitoring and Notification Infrastructure (OMNI) [Bau19]. Currently this is done with Python's standard `SysLogHandler` from the logging library, modified to format time to satisfy RFC 3339. Downstream from these transport layers, a message key is used to identify the incoming messages, their JSON payloads are extracted, and then forwarded to the appropriate Elasticsearch index. The Customs Reporter used on Cori simply uses nerscjson.

Field	Description
`executable`	Path to Python executable used by this process
`is_compute`	True if the process ran on a compute node
`is_shifter`	True if the process ran in a Shifter container
`is_staff`	True if the user is a member of NERSC staff
`job_id`	Slurm job ID
`main`	Path to application, if any
`num_nodes`	Number of nodes in the job
`qos`	Batch queue of the job
`repo`	Batch job charge account
`subsystem`	System partition or cluster
`system`	System name
`username`	User handle

TABLE 1: *Additional monitoring metadata*

On Cori compute nodes, we use the Cray Lightweight Log Manager (LLM), configured to accept RFC 5424 protocol messages on service nodes. A random service node is chosen as the recipient in order to balance load. On other nodes besides compute nodes, such as login nodes or nodes running user-facing services, rsyslog is used for message transport. This abstraction layer allows us to maintain a stable interface for logging while using an appropriately scalable transport layer for the system. For instance, future systems will rely on Apache Kafka or the Lightweight Distributed Metrics Service [Age14].

Cori has 10,000 compute nodes running jobs at very high utilization, 24 hours a day for more than 340 days in a typical year. The volume of messages arriving from Python processes completing could be quite high, so we have taken a cautious approach of monitoring a list of about 50 Python packages instead of reporting the entire contents of each process's `sys.modules`. This introduces a potential source of bias that we return to in the Discussion, but we note here that Python 3.10 will include `sys.stdlib_module_names`, a frozenset of strings containing the names of standard library modules, that could be used in addition to `sys.builtin_module_names` to remove standard library and built-in modules from `sys.modules` easily. Ultimately we plan to capture all imports excluding standard and built-in packages, except for ones we consider particularly relevant to scientific Python workflows like `multiprocessing`.

To reduce excessive duplication of messages from MPI-parallel Python applications, we prevent reporting from processes with nonzero MPI rank or `SLURM_PROCID`. Other parallel applications using e.g. `multiprocessing` are harder to deduplicate. This moves deduplication downstream to the analysis phase. The key is to carry along enough additional information to enable the kinds of deduplication needed (e.g., by user, by job, by node, etc). Table 1 contains a partial list of metadata captured and forwarded along with package names and versions.

Fields that only make sense in a batch job context are set to a default (`num_nodes: 1`) or left empty (`repo: ""`). Basic job

quantities like node count help capture the most salient features of jobs being monitored. Downstream joins with other OMNI indexes or other databases containing Slurm job data (via `job_id`), identity (`username`), or banking (`repo`) enables broader insights.

In principle it is possible that messages may be dropped along the way to OMNI, since we are using UDP for transport. To control for this source of error, we submit scheduled "canary jobs" a few dozen times a day that run a Python script that imports libraries listed in `sitecustomize` and then exits normally. Matching up those job submissions with entries in Elastic enables us to quantify the message failure rate. Canary jobs began running in October of 2020 and from that time until now (May 2021), perhaps surprisingly, we actually have observed no message delivery failures.

Prototyping, Production, and Publication

OMNI has a Kibana visualization interface that NERSC staff use to visualize Elasticsearch-indexed data collected from NERSC systems, including data collected for MODS. The MODS team uses Kibana for creating plots of usage data, organizing these into attractive dashboard displays that communicate MODS high-level metrics. Kibana is very effective at providing a general picture of user behavior with the NERSC data stack, but the MODS team wanted deeper insights from the data and obtaining these through Kibana presented some difficulty, especially due to the complexities of deduplication we discussed in the previous section. Given that the MODS team is fluent in Python, and that NERSC provides users (including staff) with a productive Python ecosystem for data analytics, using Python tools for understanding the data was a natural choice. Using the same environment and tools that users have access to provides us a way to test how well those tools actually work.

Our first requirement was the ability to explore MODS Python data interactively. However, we also wanted to be able to record that process, document it, share it, and enable others to re-run or re-create the results. Jupyter Notebooks specifically target this problem, and NERSC already runs a user-facing JupyterHub service that enables access to Cori. Members of the MODS team can manage notebooks in a Gitlab instance run by NERSC, or share them with one another (and from Gitlab) using an NBViewer service running alongside NERSC's JupyterHub.

Iterative prototyping of data analysis pipelines often starts with testing hypotheses or algorithms against a small subset of the data and then scaling that analysis up to the entire data set. GPU-based tools with Python interfaces for filtering, analyzing, and distilling data can accelerate this scale-up using generally fewer compute nodes than with CPU-based tools. The entire MODS Python data set is currently about 260 GB in size, and while this could fit into one of Cori's CPU-based large-memory nodes, the processing power available there is insufficient to make interactive analysis feasible. With only CPUs, the main recourse is to scale out to more nodes and distribute the data. This is certainly possible, but being able to interact with the entire data set using a few GPUs, far fewer processes, and without inter-node communication is compelling.

To do interactive analysis, prototyping, or data exploration we use Dask-cudf and cuDF, typically using 4 NVIDIA Volta V100 GPUs coordinated by a Dask-CUDA cluster. The Jupyter notebook itself is started from NERSC's JupyterHub using BatchSpawner (i.e., the Hub submits a batch job to run the notebook on the GPU cluster). The input data, in compressed Parquet format, are read using Dask-cuDF directly into GPU memory. These data are

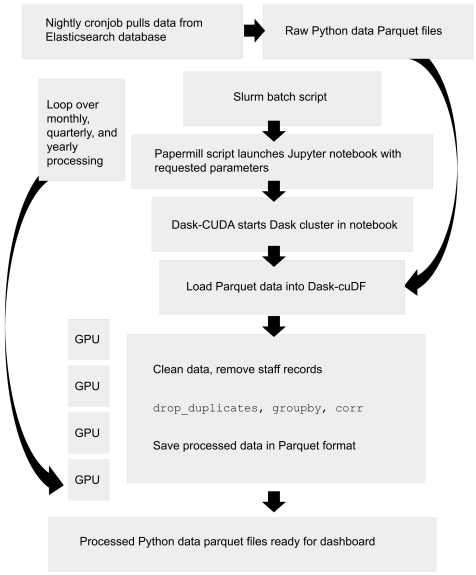

Fig. 2: *Workflow for processing and analyzing Python usage data.*

periodically gathered from OMNI using the Python Elasticsearch API and converted to Parquet. Reduced data products are stored in new Parquet files, again using direct GPU I/O.

As prototype analysis code in notebooks evolves into something resembling a production analysis pipeline, data scientists face the choice of whether to convert their notebooks into scripts or try to stretch their notebook to serve as a production tool. The latter approach has the appeal that production notebooks can be re-run interactively when needed with all the familiar Jupyter notebook benefits. We decided to experiment with using Papermill to parameterize notebook execution over months, quarters, and years of data and submit these notebooks as batch jobs. In each Jupyter notebook, a Dask-CUDA cluster is spun up and then shutdown at the end for memory/worker cleanup. Processing all data for all permutations currently takes about 2 hours on 4 V100 GPUs on the Cori GPU cluster. Fig. 2 illustrates the workflow.

Members of the MODS team can share Jupyter notebooks with one another, but this format may not make for the best way to present data to other stakeholders, in particular center management, DOE program managers, vendors, or users. Voilà is a tool that uses a Jupyter notebook to power a stand-alone, interactive dashboard-style web application. We decided to experiment with Voilà for this project to evaluate best practices for its use at NERSC. To run our dashboards we use NERSC's Docker container-as-a-service platform, called Spin, where staff and users can run persistent web services. Spin is external to NERSC's HPC resources and has no nodes with GPUs, but mounts the NERSC Global Filesystem.

Creating a notebook using a GPU cluster and then using the same notebook to power a dashboard running on a system without GPUs presents a few challenges. We found ourselves adopting a pattern where the first part of the notebook used a Dask cluster and GPU-enabled tools for processing the data, and the second part of the notebook used reduced data using CPUs to power the dashboard visualizations. We used cell metadata tags to direct Voilà to simply skip the first set of cells and pick up dashboard rendering with the reduced data. This process was a little clumsy, and we found it easy to make the mistake of adding a cell and then

forgetting to update its metadata. Easier ways of managing cell metadata tags would improve this process. Another side-effect of this approach is that packages may appear to be imported multiple times in a notebook.

We found that even reduced data sets could be large enough to make loading a Voilà dashboard slow, but we found ways to hide this by lazily loading the data. Using Pandas DataFrames to prepare even reduced data sets for rendering, especially histograms, resulted in substantial latency when interacting with the dashboard. Vaex [vaex] provided for a more responsive user experience, owing to multi-threaded CPU parallelism. We did use some of Vaex's native plotting functionality (in particular `viz.histogram`), but we primarily used Seaborn for plotting with Vaex objects "underneath" which we found to be a fast and friendly way to generate appealing visualizations. Sometimes Matplotlib was used when Seaborn could not meet our needs (to create a stacked barplot, for example).

Finally, we note that the Python environment used for both data exploration and reduction on the GPU cluster, and for running the Voilà dashboard in Spin, is managed using a single Docker image (Shifter runtime on GPU, Kubernetes in Spin).

Results

Our data collection framework yields a rich data set to examine and our workflow enables us to interactively explore the data and translate the results of our exploration into dashboards for monitoring Python. Results presented come from data collected between January and May 2021. Unless otherwise noted, all results exclude Python usage by members of NERSC staff (`is_staff==False`) and include only results collected from batch jobs (`is_compute==True`). All figures are extracted from the Jupyter notebook/Voilà dashboard.

During the period of observation there were 2448 users running jobs that used Python on Cori, equivalent to just over 30% of all NERSC users. 84% of jobs using Python ran on Cori's Haswell-based partition, 14% used Cori-KNL, and 2% used Cori's GPU cluster. 63% of Python users use the NERSC-provided Python module directly (including on login nodes and Jupyter nodes) but only 5% of jobs using Python use the module: Most use a user-built Python environment, namely Conda environments. Anaconda Python provides scientific Python libraries linked against the Intel Math Kernel Library (MKL), but we observe that only about 17% of MKL-eligible jobs (ones using NumPy, SciPy, NumExpr, or scikit-learn) are using MKL. We consider this finding in more detail in Discussion.

Fig. 3 displays the top 20 Python packages in use determined from unique user imports (i.e. how many users ever use a given package) across the system, including login nodes and Jupyter nodes. These top libraries are similar to previous observations reported from Blue Waters and TACC [Mcl11], [Eva15], but the relative prominence of `multiprocessing` is striking. We also note that Joblib, a package for lightweight pipelining and easy parallelism, ranks higher than both mpi4py and Dask.

The relatively low rankings for TensorFlow and PyTorch are probably due to the current lack of GPU resources, as Cori provides access to only 18 GPU nodes mainly for application readiness activities in support of Perlmutter, the next (GPU-based) system being deployed. Additionally, some users that are training deep learning models submit a chain of jobs that may not be expected to finish within the requested walltime; the result is

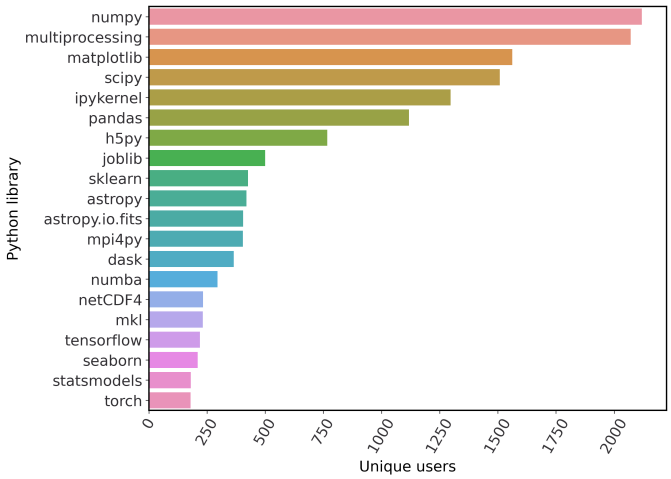

Fig. 3: *Top 20 tracked Python libraries at NERSC, deduplicated by user, across our system.*

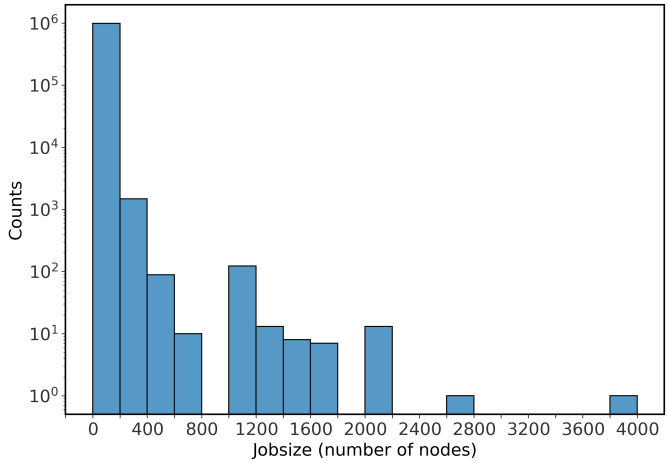

Fig. 4: *Distribution of job size for batch jobs that use Python.*

that the job may end before Customs can capture data from the `atexit`, resulting in under-reporting.

Fig. 4 shows the distribution of job size (node count) for jobs that invoked Python and imported one or more of the packages we monitor. Most of these jobs are small, but the distribution tracks the overall distribution of job size at NERSC.

Breaking down the Python workload further, Fig. 5 contains a 2D histogram of Python package counts as a function of job size. Package popularity in this figure has a different meaning than in Fig. 3: The data are deduplicated by `job_id` and package name to account for jobs where users invoke the same executable repeatedly or invoke multiple applications using the same libraries. The marginal axes summarize the total package counts and total job size counts as a function of `job_id`. Most Python libraries we track do not appear to use more than 200 nodes. Perhaps predictably, `mpi4py` and NumPy are observed at the largest node counts. Dask jobs are observed at 500 nodes and fewer, so it appears that Dask is not being used to scale as large as `mpi4py` is. Workflow managers FireWorks [Jai15] and Parsl [Bab19] are observed scaling to 1000 nodes. PyTorch (`torch`) appears at larger scales than TensorFlow and Keras, which suggests users may find it easier to scale PyTorch on Cori.

While it is obvious that packages that depend on or are

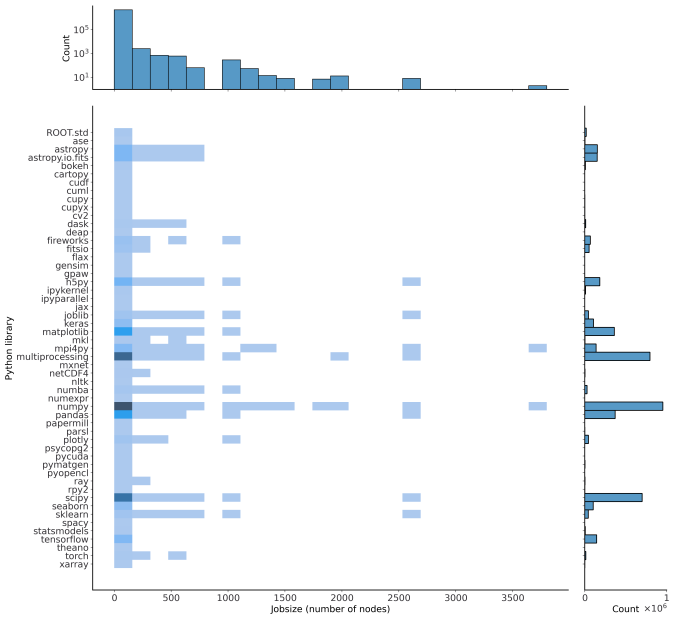

Fig. 5: *2D histogram of Python package counts versus job size. The marginal x-axis (right) shows the total package counts. The marginal y-axis (top) shows the total job counts displayed on a log scale. Here we measure number of unique packages used within a job rather than number of jobs, so these data are not directly comparable to Fig. 3 nor to Fig. 4.*

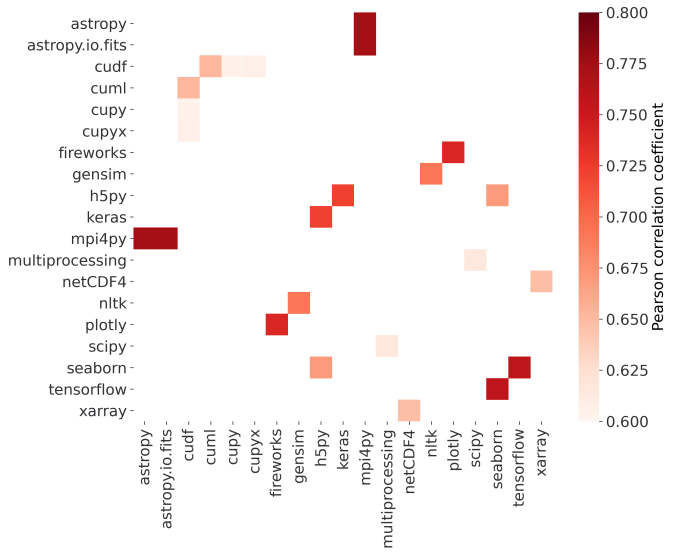

Fig. 6: *Pearson correlation coefficients for tracked Python libraries within the same job. Libraries were only counted once per job. Here we display correlation coefficient values between 0.6 and 0.8 in an effort to highlight a regime in which packages have a strong relationship but no explicit dependencies.*

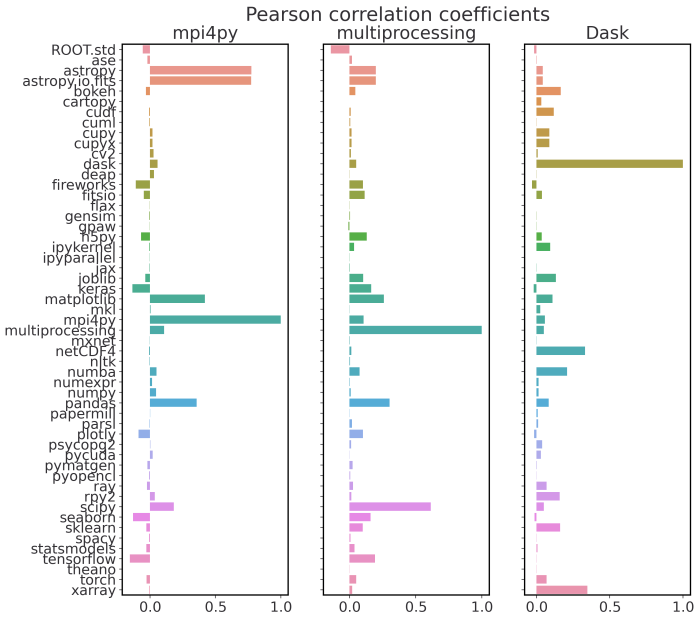

Fig. 7: *Pearson correlation coefficient values for* `mpi4py` *(left),* `multiprocessing` *(center), and Dask (right), with all other Python libraries we currently track.*

dependencies of other packages will be correlated within jobs, it is still interesting to examine the co-occurrence of certain packages within jobs. A simple way of looking at this is to determine Pearson correlation coefficients for each tracked library with all others, assigning a 1 to jobs in which a certain package was used and 0 otherwise. Fig. 6 shows an example package correlation heatmap. The heatmap includes only package correlations above 0.6 to omit less interesting relationships and less than 0.8 as a simple way to filter out interdependencies. Notable relationships between non-dependent packages include `mpi4py` and AstroPy, Seaborn and TensorFlow, FireWorks and Plotly.

We used this correlation information as a starting point for examining package use alongside `mpi4py`, `multiprocessing`, and Dask, all of which we are especially interested in because they enable parallelism within batch jobs. We omit Joblib, noting that a number of packages depend on Joblib and Joblib itself uses `multiprocessing`. Fig. 7 presents the correlations of each of these packages with all other tracked packages.

The strongest correlations observed for `mpi4py` (Fig. 7, left) is the domain-specific package AstroPy and its submodule `astropy.io.fits`. This suggests that users of AstroPy have been able to scale associated applications using `mpi4py` and that AstroPy developers may want to consider engaging with `mpi4py` users regarding their experiences. Following up with users generally reveals that using `mpi4py` for "embarrassingly parallel" calculations is very common: "My go-to approach is to broadcast data using `mpi4py`, split up input hyperparameters/settings/etc. across ranks, have each rank perform some number of computations, and then gather all the results (which are almost always NumPy arrays) using `mpi4py`." Very few users report more intricate communication patterns.

Next we consider `multiprocessing`. The `conda` tool uses `multiprocessing` but even after filtering out those cases, it remains one of the most popular Python libraries in use on Cori. In Fig. 7 (center), we do not see any particularly strong

relationships as we did with `mpi4py`. The primary correlation visible here is with SciPy, which has some built-in support for inter-operating with `multiprocessing`, for instance through `scipy.optimize`. To learn more we followed up with several of the top `multiprocessing` users. One reported: "I'm using and testing many bioinformatics Python-based packages, some of them probably using Python `multiprocessing`. But I'm not specifically writing myself scripts with `multiprocessing`." Another reported: "The calculations are executing a workflow for computing the binding energies of ligands in metal complexes. Since each job is independent, `multiprocessing` is used to start workflows on each available processor." As a package that users directly interact with, and as a dependency of other packages in scientific Python, `multiprocessing` is a workhorse.

Finally we consider Dask, a Python package for task-based parallelism and analytics at scale. Users are increasingly interested in cluster runtimes where they queue up work, submit the work to the scheduler as a task graph, and the scheduler handles dependencies and farms out the tasks to workers. Dask also inter-operates with GPU analytics libraries from NVIDIA as part of RAPIDS, so we are naturally interested in its potential for our next system based in part on GPUs. As noted, large jobs using Dask are generally smaller than those using `mpi4py` (500 nodes versus 3000+ nodes), which may indicate a potential gap in scalability on Cori. The correlation data shown in Fig. 7 (right) indicate an affinity with the weather and climate community, where `netCDF4` and `xarray` seem particularly important. We reached out to several Dask users to learn more. One responded: "I don't remember having any Python Dask-related jobs running in the past 3 months." After some additional discussion and analysis, we discovered the user was using `xarray` which we believe was using Dask unbeknownst to the user. This kind of response from "Dask users" was not uncommon.

Discussion

Our results demonstrate that we are able to collect useful data on Python package use on Cori, tag it with additional metadata useful for filtering during analysis, and conduct exploratory analysis of the data that can easily evolve to production and publication. The results themselves confirm many of our expectations about Python use on Cori, but also reveal some surprises that suggest next actions for various stakeholders. Such surprises suggest new opportunities for engagements between NERSC, users, vendors, and developers of scientific Python infrastructure.

We observe that Python jobs on Cori mostly come from environments that users themselves have provisioned, and not directly from the Python software environment module that NERSC provides. Our expectation was that the fraction of jobs running from such environments would be high since we knew through interacting with our users that custom Conda environments were very popular. A major driver behind this popularity is that users often want versions of packages that are newer than they what can get from a centrally-managed Python environment. But rather than take that as a cue that we should be updating the NERSC-provided Python environment more often, finding new ways to empower users to manage their own software better has become our priority instead. This currently includes continuing to provide easy access to Conda environments, locations for centralized installations (i.e. shared by a collaboration), and improved support for containerized environments. However we are constantly reevaluating how best to support the needs of our users.

Other results indicate that this may need to be done carefully. As mentioned in the Results, only about 17% of jobs that use NumPy, SciPy, scikit-learn, or NumExpr are using versions of those packages that rely on OpenMP-threaded, optimized Intel MKL. Given that Cori's CPU architectures come from Intel, we might expect the best performance to come from libraries optimized for that architecture. We caution that there are a number of hypotheses to consider behind this observation, as it is a question of how well-educated users are on the potential benefits of such libraries. The surprising reliance of our users on `multiprocessing` and the tendency of users to use `mpi4py` for embarrassing parallelism suggest that users may find it easier to manage process parallelism than OpenMP thread parallelism in scientific Python. Another consideration is that users value ease in software installation rather than performance. Many Conda users rely heavily on the `conda-forge` channel, which does have a much greater diversity of packages as compared to the `defaults` channel, and will install libraries based on OpenBLAS. Users may be willing or able to tolerate some performance loss in favor of being able to easily install and update their software stack. (There are no easy answers or quick fixes to this problem of facilitating both easy installation and good performance, but this is a major goal of our efforts to support Python at NERSC.) Finally, many users install complex packages designed for use on a wide range of systems; many of these packages such as GPAW may use OpenBLAS rather than MKL. Having seen that MKL adoption is low, our goal is to try to better understand the factors leading to this and ensure that users who can benefit from MKL make good choices about how they build their Python environments through documentation, training, and direct recommendation.

While some discoveries suggest next actions and user engagement for NERSC staff, others suggest opportunities for broader stakeholder action. The importance of `multiprocessing` to users on nodes with large core count suggests an opportunity for developers and system vendors. Returning to the observation that jobs using AstroPy have an tendency to also use `mpi4py`, we conclude that users of AstroPy have been able to scale their AstroPy-based applications using MPI and that AstroPy developers may want to consider engaging with our users to make that interaction better. Examining the jobs further we find that these users tend to be members of large cosmology experiments like Dark Energy Survey [Abb18], Dark Energy Spectroscopic Instrument [DESI], the Dark Energy Science Collaboration [DESC], and CMB-S4 [Aba16]. The pattern appears over many users in several experiments. We also note that the use of `astropy.io.fits` in MPI-enabled Python jobs by astronomers suggests that issues related to FITS I/O performance in AstroPy on HPC systems may be another area of focus.

While the results are interesting, making support decisions based on data alone has its pitfalls. There are limitations to the data set, its analysis, and statements we can make based on the data, some of which can be addressed easily and others not. First and foremost, we address the limitation that we are tracking a prescribed list of packages, an obvious source of potential bias. The reason for prescribing a list is technical: Large bursts of messages from jobs running on Cori at one time caused issues for OMNI infrastructure and we were asked to find ways to limit the rate of messages or prevent such kinds of bursts. Since then, OMNI has evolved and may be able to handle a higher data rate, making it possible to simply report all entries in `sys.modules` excluding built-in and standard modules (but not entirely, as

`multiprocessing` would go undetected). One strategy may be to forward `sys.modules` to OMNI on a very small random subset of jobs (say 1%) and use that control data set to estimate bias in the tracked list. It also helps us control for a major concern, that of missing out on data on emerging new packages.

Another source of bias is user opt-out. Sets of users who opt out tend to do so in groups, in particular collaborations or experiments who manage their own software stacks: Opting out is not a random error source, it is another source of systematic error. A common practice is for such collaborations to provide scripts that help a user "activate" their environment and may unset or rewrite `PYTHONPATH`. This can cause undercounts in key packages, but we have very little enthusiasm for removing the opt-out capability. Rather, we believe we should make a positive case for users to remain opted in, based on the benefits it delivers to them. Indeed, that is a major motivation for this paper.

A different systematic undercount may occur for applications that habitually run into their allocated batch job wallclock limit. As mentioned with TensorFlow, we confirmed with users a particular pattern of submitting chains of dozens of training jobs that each pick up where the previous job left off. If all these jobs hit the wallclock limit, we will not collect any data. Counting the importance of a package by the number of jobs that use it is dubious; we favor understanding the impact of a package from the breadth of the user community that uses it. This further supports the idea that multiple approaches to understanding Python package use are needed to build a complete picture; each has its own shortcomings that may be complemented by others.

Part of the power of scientific Python is that it enables its developers to build upon the work of others, so when a user imports a package it may import several other dependencies. All of these libraries "matter" in some sense, but we find that often users are importing those packages without even being aware they are being used. For instance, when we contacted users who appeared to be running Dask jobs at a node count of 100 or greater, we received several responses like "I'm a bit curious as to why I got this email. I'm not aware to have used Dask in the past, but perhaps I did it without realizing it." More generally, large-scale jobs may use Python only incidentally for housekeeping operations. Importing a package is not the same as actual use, and use of a package in a job running at scale is not the same as that package actually being used at scale.

Turning to what we learned from the process of building our data analysis pipeline, we found that the framework gave us ways to follow up on initial clues and then further productionize the resulting exploratory analysis. Putting all the steps in the analysis (extraction, aggregation, indexing, selecting, plotting) into one narrative improves communication, reasoning, iteration, and reproducibility. One of our objectives was to manage as much of the data analysis as we could using one notebook for exploratory analysis with Jupyter, parameterized calculations in production with Papermill, and shared visualization as a Voilà dashboard. Using cell metadata helped us to manage both the computationally-intensive "upstream" part of the notebook and the less expensive "downstream" dashboard within a single file. One disadvantage of this approach is that it is very easy to remove or forget to apply cell tags. This could be addressed by making cell metadata easier to apply and manage. The Voilà JupyterLab extension helps with this problem by providing a preview of a dashboard rendering before it is published to the web. Another issue with the single-notebook pattern is that some code may be repeated for different purposes. This is not a source of error necessarily, but it can cause confusion. All of these issues disappear if the same hardware could be used to run the notebook in exploratory analysis, pipelined production, and dashboard phases, but these functions are simply not available in a single system at NERSC today.

Conclusion

We have taken our first small steps in understanding the Python workload at NERSC in detail. Instrumenting Python to record how frequently key scientific Python packages are being imported in batch jobs on Cori confirmed many of our assumptions but yielded a few surprises. The next step is acting on the information we have gathered, and of course, monitoring the impact those actions have.

Using Python itself as a platform for analyzing the Python workload poses a few challenges mostly related to supporting infrastructure and tooling. With a few tricks, we find that the same Jupyter notebooks can be used for both exploratory and production data analysis, and also to communicate high-level results through dashboards. We initiated this project not only to perform Python workload analysis but to test the supposition that users could assemble all the pieces they needed for a Python-based data science pipeline at NERSC. Along the way, we identified shortcomings in our ecosystem, and this motivated us to develop tools for users that fill those gaps, and gave us direct experience with the very same tools our users use to do real science.

In the near future, we will expand Python workload analysis to Perlmutter, a new system with CPU+GPU and CPU-only nodes to identify users of the CPU nodes who might be able to take advantage of GPUs. Other future plans include examining Python use within the context of specific science areas by linking our data with user account and allocation data, and using natural language processing and machine learning to proactively identify issues that users have with Python on our systems. Another interesting avenue to pursue is whether the monitoring data we gather may be of use to users as an aid for reproducible computational science. If users are able to access Python usage data we collect from their jobs, they could use it to verify what Python packages and package versions were used and obtain some degree of software provenance for reproducing and verifying their results.

We anticipate that developers of scientific Python software may find the information we gather to be informative. Readers can view the public MODS Python dashboard at https://mods.nersc.gov/public/

Acknowledgments

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231. We thank our colleagues Brian Austin, Tiffany Connors, Aditya Kavalur, and Colin MacLean for discussions on workload analysis, process monitoring, and Python. We also thank the Vaex developers for their help and advice, and the Dask-cuDF and cuDF developers for their responsiveness to issues and advice on effective use of Dask-cuDF and cuDF. Finally we thank our users who were kind enough to provide feedback to us and allow us to use their quotes about how they are using Python at NERSC.

REFERENCES

[Aba16] K. N. Abazajian, et al., *CMB-S4 Science Book, First Edition*, 2016. <https://arxiv.org/abs/1610.02743>

[Abb18] T. M. C. Abbott, et al., *Dark Energy Survey year 1 results: Cosmological constraints from galaxy clustering and weak lensing* Physical Review D, 98, 043526, 2018. <https://doi.org/10.1103/PhysRevD.98.043526>

[Age14] A. Agelastos, et al., *Lightweight Distributed Metric Service: A Scalable Infrastructure for Continuous Monitoring of Large Scale Computing Systems and Applications,* Proc. IEEE/ACM International Conference for High Performance Storage, Networking, and Analysis, SC14, New Orleans, LA, 2014. <https://doi.org/10.1109/SC.2014.18>

[Agr14] K. Agrawal, et al., *User Environment Tracking and Problem Detection with XALT,* Proceedings of the First International Workshop on HPC User Support Tools, Piscataway, NJ, 2014. <http://doi.org/10.1109/HUST.2014.6>

[Bab19] Y. Babuji, et al., *Parsl: Pervasive Parallel Programming in Python,* 28th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC), Phoenix, AZ, 2019. <https://doi.org/10.1145/3307681.3325400>

[Bau19] E. Bautista, et al., *Collecting, Monitoring, and Analyzing Facility and Systems Data at the National Energy Research Scientific Computing Center,* 48th International Conference on Parallel Processing: Workshops (ICPP 2019), Kyoto, Japan, 2019. <https://doi.org/10.1145/3339186.3339213>

[DESI] The DESI Collaboration, *The DESI Experiment Part I: Science, Targeting, and Survey Design,* Science Final Design Report, <https://arxiv.org/abs/1611.00036>

[Eva15] T. Evans, A. Gomez-Iglesias, and C. Proctor, *PyTACC: HPC Python at the Texas Advanced Computing Center,* Proceedings of the 5th Workshop on Python for High-Performance and Scientific Computing, SC15, Austin, TX, 2015 <https://doi.org/10.1145/2835857.2835861>

[Fah10] M. Fahey, N Jones, and B. Hadri, *The Automatic Library Tracking Database,* Proceedings of the Cray User Group, Edinburgh, United Kingdom, 2010. <https://doi.org/10.1145/1838574.1838582>

[Fur91] J. L. Furlani, *Modules: Providing a Flexible User Environment,* Proceedings of the Fifth Large Installation Systems Administration Conference (LISA V), San Diego, CA, 1991.

[Gam15] T. Gamblin, et al., *The Spack Package Manager: Bringing Order to HPC Software Chaos,* in Supercomputing 2015, SC15, Austin, TX, 2015. <https://doi.org/10.1145/2807591.2807623>

[Jac16] D. M. Jacobsen and R. S. Canon, *Shifter: Containers for HPC,* in Cray Users Group Conference (CUG16), London, United Kingdom, 2016

[Jai15] Jain, A., et al., *FireWorks: a dynamic workflow system designed for high-throughput applications.* Concurrency Computat.: Pract. Exper., 27: 5037–5059, 2015. <https://doi.org/10.1002/cpe.3505>

[DESC] LSST Dark Energy Science Collaboration, *Large Synoptic Survey Telescope: Dark Energy Science Collaboration,* White Paper, 2012. <https://arxiv.org/abs/1211.0310>

[Mac17] C. MacLean. *Python Usage Metrics on Blue Waters* Proceedings of the Cray User Group, Redmond, WA, 2017.

[vaex] A. Maarten. and J. V. Breddels, *Vaex: big data exploration in the era of Gaia,* Astronomy & Astrophysics, 618, A13, 2018. <https://arxiv.org/abs/1801.02638v1>

[Mcl11] R. McLay, K. W. Schulz, W. L. Barth, and T. Minyard, *Best practices for the deployment and management of production HPC clusters* in State of the Practice Reports, SC11, Seattle, WA, 2011. <https://doi.acm.org/10.1145/2063348.2063360>

[MODS] NERSC 2017 Annual Report. pg 31. <https://www.nersc.gov/assets/Uploads/2017NERSC-AnnualReport.pdf>

[OA20] NERSC Operational Assessment. In press, 2020.