EPJ Web of Conferences **175**, 09009 (2018) https://doi.org/10.1051/epjconf/201817509009
Lattice 2017

Automated lattice data generation

Venkitesh Ayyar[1], *Daniel C.* Hackett[1,⋆], *William I.* Jay[1], and *Ethan T.* Neil[1,2]

[1] *Department of Physics, University of Colorado, Boulder, Colorado 80309, USA*
[2] *RIKEN-BNL Research Center, Brookhaven National Laboratory, Upton, New York 11973, USA*
[3] *Raymond and Beverly Sackler School of Physics and Astronomy, Tel Aviv University, 69978 Tel Aviv, Israel*

Abstract. The process of generating ensembles of gauge configurations (and measuring various observables over them) can be tedious and error-prone when done "by hand". In practice, most of this procedure can be automated with the use of a workflow manager. We discuss how this automation can be accomplished using Taxi, a minimal Python-based workflow manager built for generating lattice data. We present a case study demonstrating this technology.

1 `taxi`: Production workflow management for lattice gauge theory

Workflow management in any large-scale calculation using high-performance computing may present a significant logistical challenge. Efficient use of computational resources is usually best accomplished by dividing the overall project into a number of smaller tasks, which can then be carried out in parallel by a pool of worker jobs. Since the tasks can have complicated inter-dependencies, coordination of the workers is essential when distributing tasks from the available pool.

For lattice gauge theory (LGT) in particular, most projects share a common task dependency structure which arises from the Markov-Chain Monte Carlo (MCMC) methods central to such calculations. First, an ensemble of configurations is generated in a Markov chain. Each configuration-generation task depends on the previous configuration-generation task in the chain. Second, physical observables are measured over the ensemble. To make a measurement on any given configuration, that configuration must already exist, so each measurement task depends on a configuration-generation task. This gives the overall workflow a tree-like structure: the "trunk" gauge ensemble must be completed serially, but once it is finished multiple other tasks can branch off from each configuration-generating task.

Given this structure, automating the workflow to generate a single LGT ensemble can be relatively straightforward; one can use a simple self-resubmitting script that will evolve the gauge configurations one-by-one, and then similar scripts which will run through the set of available configurations serially and compute any observables of interest. However, this approach is inflexible: with every worker specialized to a single task, each worker must exit when work for its specialization is exhausted, regardless of the state of the rest of the task pool. On high-performance computers with significant competition in the job queue, this can increase the wall time required to complete a set of tasks. Moreover, managing each ensemble with individual scripts can become cumbersome when the number of ensembles and/or observables becomes large, a common situation in modern LGT calculations.

⋆Speaker, e-mail: daniel.hackett@colorado.edu

These considerations motivate the use of a *workflow manager*, which can flexibly assign new tasks to general-purpose worker jobs as old tasks are completed. This allows a worker which exhausts a certain type of task, or which reaches the end of a branch of tasks, to pick up and continue running different types of tasks, or to start working on another branch. For projects with many ensembles and complicated sets of tasks, controlling and modifying a single centralized task repository is much simpler than interacting with a disparate set of scripts. The existence of a central "task plan" with information about dependencies and status also enables the automation of the overall workflow, including verification of task completion and error recovery.

Our collaboration became interested in automation and workflow management to enable our exploration of the thermodynamics of SU(4) gauge theory with fermions in multiple representations [1, 2]. In the course of our investigation, we ran $\sim 4 \times 10^5$ HMC trajectories over 10^3 ensembles, measuring both Wilson flow and spectroscopy for both representations of fermion. Generating this volume of data without significant automation proved logistically intractable, leading us to explore workflow managers.

There are a number of general workflow management systems available already, such as Pegasus [3], Makeflow [4], Apache Taverna [5], and Kepler [6]. We chose to create a new tool, `taxi`, for two main reasons, both related to ease of use for our particular problem. First, `taxi` is specialized to the MCMC-focused workflow of lattice gauge theory; this is a less general approach, but as a trade-off requires less code to be written for different MCMC calculations. Our second motivation was the requirement, typical in large-scale lattice calculations,of running the same project on a number of different remote machines, with heterogenous software environments, network access restrictions, etc. Setting up one of the more general workflow managers listed above (and all of their dependencies) on many different systems where user access is limited can present a significant (or even impassable) logistical obstacle. `taxi` is designed to be lightweight and flexible, requiring only Python 2.6.6 (included with most Linux distributions) as a minimal dependency and modularized to work with different queueing systems interchangeably.

In the next section, we give an overview of `taxi`'s design philosophy for task management in general and for structure MCMC workflow plans in particular. Section 3 gives an example of basic usage for a toy LGT workflow. Finally, in section 4 we give an overview of the current status and near-future plans for developing `taxi`.

2 System architecture

2.1 Task management model

The `taxi` workflow system has two types of actor: Taxis and the Dispatcher. A Taxi is a general-purpose worker which executes computing tasks assigned by the Dispatcher. The Dispatcher contains the planned workflow in full, including priorities, dependencies, and completion status for all tasks. Whenever a Taxi completes a task, it queries the Dispatcher to receive the next available task of highest priority; the Dispatcher provides the appropriate task and updates its records.

A key feature of this "Taxi-Dispatcher" model is that the Dispatcher is only active when it is queried by a Taxi. This allows the Dispatcher in `taxi` to be structured as a passive repository, typically built on top of some form of SQL database, with Dispatcher-specific logic built in to the interface to the database seen by the Taxis. This removes the need for any additional monitor program to be run on the remote machine, instead using the workers to carry out organizational tasks. Contrast this with the more typical "overlord-minion" model, in which central coordination is provided by an active monitor program which tracks the workers and assigns new tasks when it detectors that workers are idle. In the latter model, the "overlord" program must be active or accessible on the remote machine at all

times, which can be difficult to maintain reliably (dropped connections, etc.), potentially wasteful of computational resources (if run on compute nodes), and/or irritating to administrators (if a resource-intensive monitor is run on an access or compile node).

In practice, we divide the passive central repository in to the Dispatcher, which contains information about tasks only, and the Pool, which tracks the status of the Taxis in the queue as well as their resource limits (maximum time and number of nodes). This separation allows for easier adjustment of the desired number of workers and resource limits, independent of the set of available tasks, and for easier coordination when running multiple projects simultaneously on the same queue.

2.2 MCMC workflow structure

As discussed in section 1, there is a common structure to MCMC workflows, regardless of the exact theory under investigation or software suite being used. `taxi` takes advantage of this structure to minimize the work necessary to specify data generation tasks (i.e., to specify a "run") and to adapt the software to a new application. This comprises most of the user-facing component of `taxi`, as most or all of workflow planning and task management for typical workflows is taken care of transparently and automatically, without any additional input from the user.

`taxi` includes two abstract superclasses for running MCMC tasks. Instances of `ConfigGenerator` run binaries that advance the Markov chain and generate new configurations: in LGT this is usually a Hybrid Monte Carlo (HMC) binary. Instances of `ConfigMeasurement` run binaries that perform measurements on existing configurations: some LGT examples include measuring correlation functions (spectroscopy) or computing various observables as the configuration is evolved under Wilson flow. Adapting `taxi` to a new application or lattice software suite (e.g., a variant of MILC) amounts to implementing subclasses of these two abstract classes. This process requires the user to implement only a few methods in each subclass: a constructor that preprocesses and stores the parameters needed to run the binary; a `build_input` method that generates the input string or input file to be fed to the binary, using the parameters stored in each object; and a `verify_output` method that checks that the output of the binary is present, well-formed, and complete. To give a sense of scale, the five subclasses in our MILC application are each $100 - 300$ lines of low-density Python code.

Most of the rest of the necessary functionality to generate MCMC data is built in to the abstract superclasses. For example: each sequence of gauge configurations in a run will start with either a fresh start (i.e., a random configuration or a unit configuration), from a configuration generated by a previous run and stored in the filesystem, or by branching off from another sequence of configurations. The `ConfigGenerator` superconstructor takes a single `starter` parameter that detects and handles each of these cases appropriately. Similarly, the `ConfigMeasurement` constructor takes a `measure_on` parameter that may be used to specify either a stored gauge configuration file or a `ConfigGenerator` task instance from the same run. If `starter` or `measure_on` is used to specify a `ConfigGenerator` from the same run, the object will read relevant physical parameters from the specified `ConfigGenerator` instance, obviating the need to specify them by hand (and removing the opportunity to specify mismatched parameters). If `starter` or `measure_on` is instead used to specify a stored file, `taxi` uses modularized file naming conventions (easily specified by the user and automatically detected by the software) to read physical parameters from the filename, similarly simplifying the process of making measurements (any necessary parameters not present in the filename must still be specified by the user). This behavior is helpful in specifying long chains of `ConfigGenerator` tasks and even more useful in the case of measurements: for example, to measure correlation functions on some configuration, one need only specify the parameters specific to spectroscopy (e.g., smearing

radius and boundary conditions) versus having to carefully provide all the matching parameters used to generate the configuration.

3 Usage example

Figure 1 shows the code for a working toy example which sets up and launches two jobs (i.e., two worker taxis) which coordinate to generate two ensembles of pure gauge data on 4^4 lattices and apply the Wilson flow to those ensembles. This example illustrates a number of convenience functions which further simplify run specification for the user.

In the first block of code in the `__main__` section of the script, the `make_config_generator_stream` convenience function is used to construct two chains of instances of `PureGaugeORATask`, a subclass of `ConfigGenerator` that runs a pure-gauge MCMC binary. The first set, `seed_stream`, will generate an ensemble of 10 configurations at $\beta = 7.75$ from a fresh start, with each stored configuration separated by 100 trajectories. The second set, `fork_stream`, will generate a second ensemble of 5 configurations at $\beta = 7.76$. To cut down on equilibration time versus a fresh start, `fork_stream` forks off from `seed_stream` after the fifth configuration. Integrator parameters (such as the number of overrelaxation steps or number of quantum heatbath steps to run per trajectory) are specified by default arguments to the PureGaugeORATask constructor.

In the second block, all fifteen `PureGaugeORATask`s are pooled in to `cg_pool`, and the `measure_on_config_generators` convenience function is used to specify the measurement of Wilson Flow across both ensembles. This convenience function creates instances of `config_measurement_class` applied to all `ConfigGenerator` instances supplied to the argument `measure_on`. Each instance of `config_measurement_class`, in this case `FlowTask`, steals parameters from the `ConfigGenerator` instances they are associated with and so only the flow integrator step size `epsilon` and the maximum time to flow `tmax` need to be provided to perform the measurement. The parameter `start_at_traj` specifies that the first two configurations (i.e. 200 trajectories) are for equilibration and thus not to be measured on; consequently, `flow_pool` contains only eleven instances of `FlowTask`.

The remainder of the data generation process is set in to motion by the final two blocks. In the third block, the run-specification script initializes a `Dispatcher`, which compiles the task pool in to a SQLite dispatch database to be stored in the provided location. The fourth block of code generates two taxis, each of which are to run on one node; instantiates a `Pool` object for the taxis to coordinate among themselves; and registers those taxis with both the `Dispatcher` and `Pool`. The final line launches the appropriate number of taxis to start the job working: in this case, one taxi will be launched, as there is one "trunk" job available to work on at the beginning (the first task in `seed_stream`). Once launched, the taxis will automatically maintain the optimal number of running taxis (equal to the number of active sequences of `ConfigGenerator`s, or "trunk number" of the task pool): so, after the fifth task in `seed_stream` completes, a second taxi will be launched to work on `fork_stream` in parallel. After launching the first taxi, the run-specification script exits and (barring task failures) no further user intervention is required.

4 Current status and future plans

An experimental release of the taxi software package is available on GitHub [7]. The GitHub repository includes an example application (a suite of runners for our custom multi-representation variant of the MILC binary suite, which should be easily adaptable to any MILC derivative), and a number

```
import taxi
import taxi.mcmc
from taxi.pool import SQLitePool
from taxi.dispatcher import SQLiteDispatcher
from taxi.apps.milc.pure_gauge_ora import PureGaugeORATask
from taxi.apps.milc.flow import FlowTask

PureGaugeORATask.binary = './pg_ora'
FlowTask.binary = './flow'

if __name__ == '__main__':
    seed_stream = taxi.mcmc.make_config_generator_stream(
        config_generator_class=PureGaugeORATask,
        starter=None,
        req_time=240, streamseed=1, N=10,
        Ns=4, Nt=4, beta=7.75, n_traj=100
    )
    fork_stream = taxi.mcmc.make_config_generator_stream(
        config_generator_class=PureGaugeORATask,
        starter=seed_stream[4],
        req_time=240, streamseed=2, N=5,
        Ns=4, Nt=4, beta=7.76, n_traj=100
    )

    cg_pool = seed_stream + fork_stream
    flow_pool = taxi.mcmc.measure_on_config_generators(
        config_measurement_class=FlowTask,
        measure_on=cg_pool,
        req_time=60, start_at_traj=200,
        tmax=1, epsilon=.03
    )

    job_pool = cg_pool + flow_pool
    my_disp = SQLiteDispatcher(db_path="./dispatch.sqlite")
    my_disp.initialize_new_job_pool(job_pool)

    taxi_list = [taxi.Taxi(time_limit=10*60, nodes=1)) for i in range(2)]
    my_pool = SQLitePool(db_path="./pool.sqlite",
        work_dir="./work/", log_dir="./log/")
    for my_taxi in taxi_list:
        my_pool.register_taxi(my_taxi)
        my_disp.register_taxi(my_taxi, my_pool)

    my_pool.spawn_idle_taxis(dispatcher=my_disp)
```

Figure 1. Example run-specification script. This script specifies and launches a run that will generate two ensembles of pure-gauge data, including a measurement of the Wilson flow.

of examples/templates for common use cases, such as generating new ensembles of configurations while performing standard measurements, or performing measurements on an existing set of stored configuration files. These examples should be sufficient to illustrate basic use. In the near future, we hope to provide detailed documentation that elaborates on how to adapt `taxi` to new applications, how to use `taxi` for common uses cases, and how to adapt `taxi` to run on new clusters or machines. Also in development is a suite of toy applications and examples, so that users won't need to first adapt `taxi` to their binary suite to play with example workflows.

The driving design requirement of `taxi` has been ease-of-use. For ease of installation, the software is designed to work transparently with Python's `virtualenv` virtual environment system, which allows users to install Python software painlessly on remote machines without administrator rights. In the context of getting `taxi` up and running, `virtualenv` provides an application-sufficient subset of the capabilities of container systems like Docker with the benefit of being nearly trivial to set up and install without enhanced user privileges or help from administrators. Localizations (i.e., the framework required to run on different supercomputers or different queueing systems) in `taxi` are modular, meaning that `taxi` can be adapted to run on new machines with a minimum of coding. The current distribution of taxi includes localizations for an SGE-based queueing system and for the USQCD machines at Fermilab, which use TORQUE PBS. Coming soon are localizations for machines that use SLURM, as well as for the Redis queueing system, a local queue to run on workstations and laptops. Another possible queueing backend is METAQ [8, 9], which would allow `taxi` to manage tasks inside a larger bundled job.

The package includes a set of command-line tools to monitor job progress and to recover from common failure cases. For example, it is often the case that HMC jobs are initially run with overly-optimistic integrator parameters or estimated walltime requirements, resulting in task failure or job cancellation. The package includes tools to adjust task parameters in situ and to roll back failed tasks. For more sophisticated monitoring and modification of active runs, `taxi` and its tool suite have been designed with an eye towards using the Jupyter notebook as a dashboard interface. Examples of this use case are forthcoming.

Currently, Dispatcher and Pool are implemented in SQLite, a local file-based implementation of SQL. While SQLite is adequate for running on single machines, a central networked database will be necessary to coordinate runs across multiple machines (as well as providing other benefits). To this end, we are looking in to adapting `taxi` to other flavors of SQL, particularly the open-source PostgreSQL. Because much of the Dispatcher logic is implemented abstractly and modularly, adaptation to other types of database would be equally straightforward.

To push forward the automation horizon, we have been considering what is possible when automated analysis of automatically-generated data is used to specify further data to generate, thus "closing the loop" on data generation. This idea leads straightforwardly to the complete automation of several tedious tasks traditionally performed by humans. Among these tasks are scaling tests on supercomputers, tuning of HMC integrator parameters, and running data until some threshold amount of statistics has been achieved. An even more sophisticated and physically interesting case is in thermodynamics: it should be possible to completely automate the exploration of bare-parameter phase diagrams. In such a case, the user need only specify the ranges of parameters to explore (including a bounding set of conditionals, e.g. "the AWI quark mass is less than a certain value"). In this closed pipeline, `taxi` generates data; automated analysis software measures observables on the data as it is produced (e.g., the thermodynamic phase, how long it took an ensemble to equilibrate, the AWI quark mass); finally, a further piece of automated-run-specification software examines the analyzed data and adds new tasks to the ongoing run to continue the exploration. Such a closed pipeline can be used to precisely (to whatever specified threshold) pin down the location of finite-temperature transitions

or κ_c lines with only minimum human intervention. The only piece of this loop that we have not yet implemented in practice is the automated-run-specification component.

Acknowledgements

Our research was supported in part by the U.S. Department of Energy under grant number DE-SC0010005. Brookhaven National Laboratory is supported by the U. S. Department of Energy under contract DE-SC0012704.

References

[1] V. Ayyar, T. DeGrand, D.C. Hackett, W.I. Jay, E.T. Neil, Y. Shamir, B. Svetitsky, *Chiral Transition of SU(4) Gauge Theory with Fermions in Multiple Representations*, in *Proceedings, 35th International Symposium on Lattice Field Theory (Lattice2017): Granada, Spain*, to appear in EPJ Web Conf., 1709.06190

[2] V. Ayyar, D.C. Hackett, W.I. Jay, E.T. Neil, *Confinement study of an SU(4) gauge theory with fermions in multiple representations*, in *Proceedings, 35th International Symposium on Lattice Field Theory (Lattice2017): Granada, Spain*, to appear in EPJ Web Conf., 1710.03257

[3] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P.J. Maechling, R. Mayani, W. Chen, R. Ferreira da Silva, M. Livny et al., Future Generation Computer Systems **46**, 17 (2015), funding Acknowledgements: NSF ACI SDCI 0722019, NSF ACI SI2-SSI 1148515 and NSF OCI-1053575

[4] M. Albrecht, P. Donnelly, P. Bui, D. Thain, *Makeflow: A Portable Abstraction for Data Intensive Computing on Clusters, Clouds, and Grids*, in *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies* (ACM, New York, NY, USA, 2012), SWEET '12, pp. 1:1–1:13, ISBN 978-1-4503-1876-1, http://doi.acm.org/10.1145/2443416.2443417

[5] Apache Taverna, https://taverna.incubator.apache.org/

[6] Kepler Project, https://kepler-project.org

[7] D.C. Hackett, E.T. Neil, *Taxi: Production workflow management for lattice gauge theory*, https://github.com/dchackett/taxi (2017)

[8] E. Berkowitz (2017), 1702.06122

[9] E. Berkowitz, G.R. Jansen, K. McElvain, A. Walker-Loud, *Job Management and Task Bundling* (2017), 1710.01986, https://inspirehep.net/record/1628820/files/arXiv:1710.01986.pdf