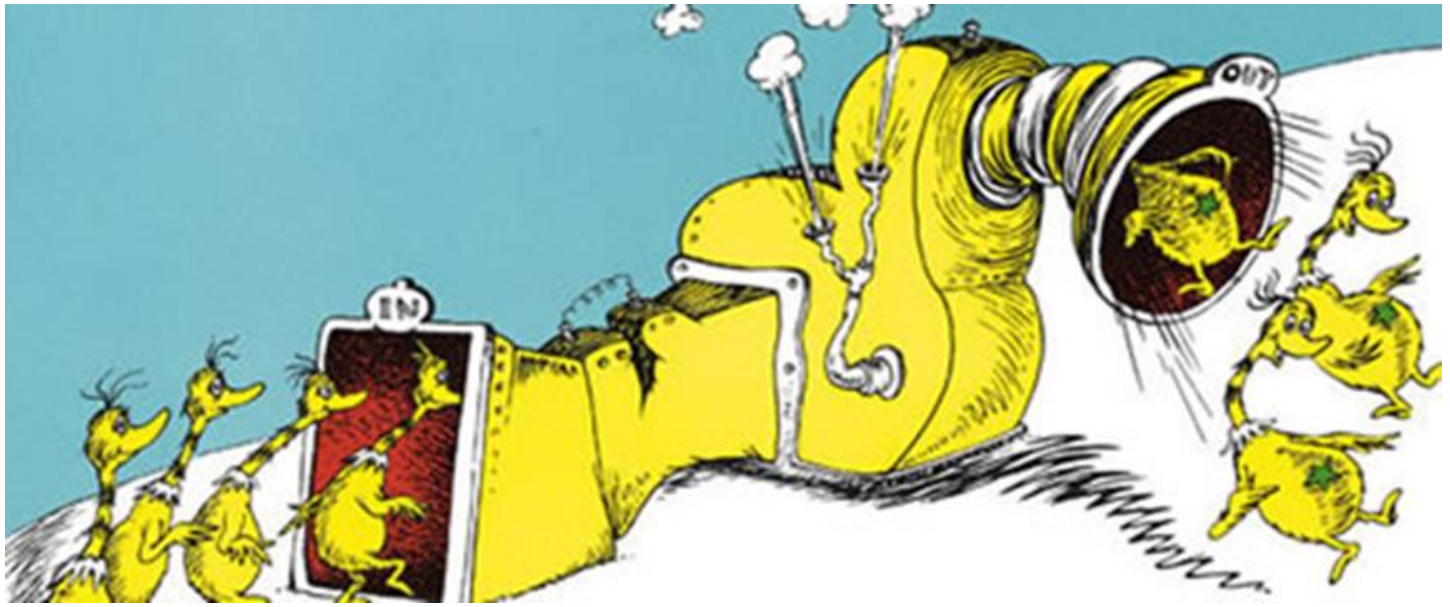
Automated Lattice Data Generation



Presented by Daniel Hackett

June 23, 2017

Lattice 2017

University of Colorado, Boulder

Venkitesh Ayyar

Daniel Hackett

William Jay

Ethan Neil *

* RIKEN-BNL Research Center

Outline

- Motivation
  - Backstory
  - How far can automation go?
- Problem statement
  - What are the constraints?
  - Graceful solution: taxi-dispatcher model
- taxi software package
  - Sketch simple use case: pure gauge data generation

Motivation/backstory: Multirep thermodynamics

Study of phase structure of SU(4) theory with dynamical fermions in 2 irreps

- 3D bare parameter space $(\beta, \kappa_4, \kappa_6)$ – Lots of room for novel phases to hide in
- Big dataset: $\sim$ 900 ensembles, $\sim$ 50000 gauge configs

Unusual case: small-volume thermo is effectively free, can run on local resources.

Bottleneck is organization/automation, not computational power/cost

- "Human time vs. computer time"

What we learned

- **Trivial statement:** Computers can do sufficiently simple analyses without human intervention
- **Non-trivial statement:** "Sufficiently simple" may include much more than expected

Automated Data Pipeline

"I wonder what happens at $(\beta, \kappa_4, \kappa_6)$?"

Automated!

- Launch HMC stream
- Make measurements on new gauge configurations

This talk

- Parse data and sync to database
- Perform "bulk analysis" of data, e.g.:
  - Determine phases of ensembles (various diagnostics)
  - Compute meson masses, quark masses, decay constants
    - (Grid over various fit parameters, e.g.: t_{min}, t_{max})
- Compile summary/"digest" of data for all ensembles

Look at observables at $(\beta, \kappa_4, \kappa_6)$

Problem Statement

We want to generate lattice data. What are our typical constraints?

Problem: Queues

On most machines, jobs:

- Wait on queue before running
- Are not allowed to run indefinitely (24 hour limit)

Want to maximize data generation rate

- Running one task per submission is rarely optimal

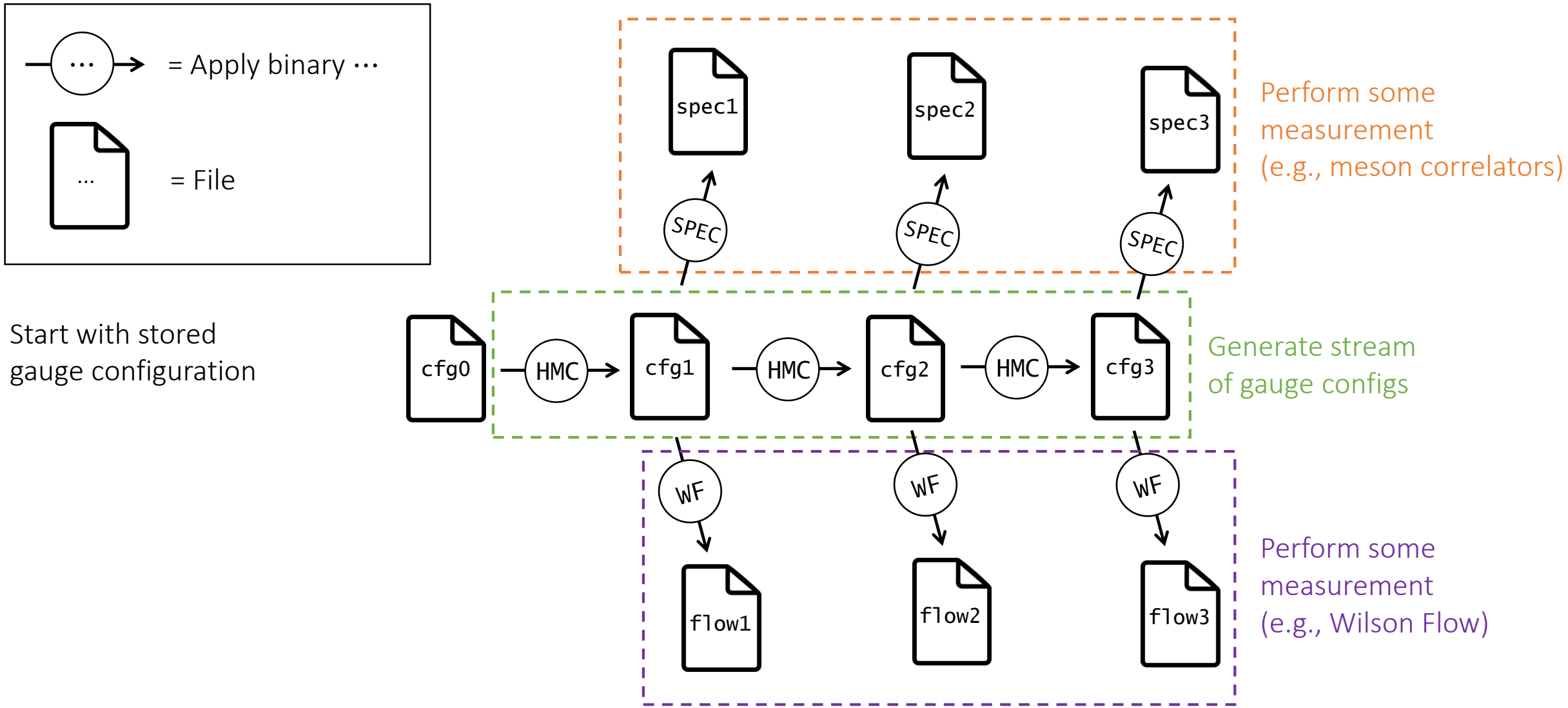
Solution (waiting): "Worker" scripts do more than one task per submission

- **minimal e.g.:** bash script with a for loop

Solution (time limits): Worker resubmission

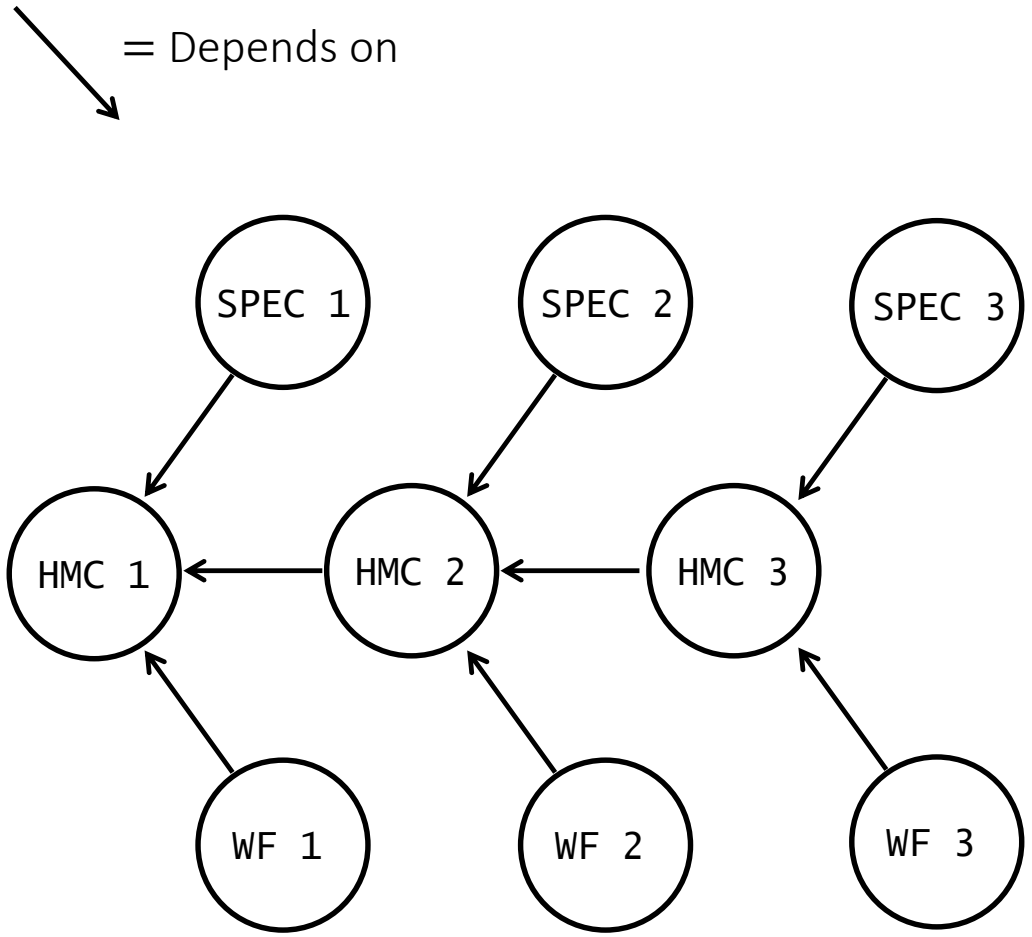
- **Centralized:** Monitor program launches replacement workers
- **Decentralized:** Workers coordinate, launch new workers
  - Limiting case: Self-resubmitting job scripts

Natural Structure of Lattice Data Generation



Natural Structure of Lattice Data Generation

"Integrate out the files," get a task dependency tree



To generate data, must resolve dependency tree

Problem: Different binaries perform each task, but want measurements before HMC is done

Solution: Flexible workers can perform any task
"Specialization is for insects." –Robert Heinlein

(**Buzzword:** Scientific Workflow Management. For a good discussion in the context of another workflow management system than taxi, see Pegasus docs)

Problem: Remote computing resources

Need to run on remote machines

(Unless you have a very nice laptop)

Awkward solution: Active central monitor/task-manager

Monitor must stay running – robustness issues!

If run on local machine: must keep local machine on, connection to remote machine open

If run on access node: admins might not be happy

If run on compute node: wasteful

More graceful solution: Passive, decentralized monitor/task-manager

Taxi-Dispatcher Model of Task Management

Idea: Passive central control

Workflow:

- Taxi runs some task
- Upon finishing task, taxi:
  - Tells Dispatcher that task is complete
  - Requests a new task to work on
- Dispatcher then:
  - Updates task states
  - Decides what task should be run next
    - (Factoring in dependencies, task priority, time remaining for taxi job, etc.)
  - Tells taxi new task to run

Taxi: general-purpose worker script
- Runs on compute nodes, performs tasks
- Submits new/replacement taxis

Dispatcher: passive task manager
- Keeps track of task states
- Decides which tasks to run

Dispatcher only needs to “think” when called by taxis

⇒ No separate monitor program to keep running!

taxi software package

Implementation of Taxi-Dispatcher model
- In practice: Dispatcher “thinking” is distributed across Taxis, “memory” is shared SQLite DB
- Task objects stored in SQLite DB as JSON payloads

Minimal/lightweight workflow management system for lattice/MCMC data generation

Written in Python 2.6.6 (distributed with Linux) for painless setup

Modular
- Easily adapted to new machines/queue systems (e.g.: Fermilab PBS, SLURM, …)
- Easily adapted to new binary suites/applications (e.g.: MILC)

Lots of common structure between MCMC applications:

Abstract superclasses for basic MCMC tasks capture most structure
- **ConfigGenerator** – Task that outputs a gauge config
- **ConfigMeasurement** – Task that processes a gauge config

To run some binary, write subclasses

Example: Pure gauge theory and Wilson Flow

taxi example: pure gauge (Task class)

Subclass of ConfigGenerator to run a pure gauge binary

[PureGauge inherits from ConfigGenerator]

```
class PureGauge(ConfigGenerator):
    def __init__(self, Ns, Nt, beta, other_pg_params=defaults, **more_general_params):
        # Superconstructor handles most non-app-specific structure
        super(PureGauge, self).__init__(**more_general_params)

        self.binary = "/path/to/pure_gauge_binary" # Where is the binary?

        # Store params in self
        self.Ns, self.Nt, self.beta = Ns, Nt, beta
        self.other_pg_params = other_pg_params

    def build_input_string(self):
        return "string to feed to binary"
        # Can provide heredoc, or generate input file and provide path

    def verify_output(self):
        # Looks at files self.fout and self.saveg
        # Returns: Outputs are (present and complete and well-formatted)
```

taxi example: pure gauge (Run specification)

Script to run off a stream of 100 pure gauge configurations

```
## BOILERPLATE SETUP CODE (Some machine dependence)

cfg_pool = make_config_generator_stream(
    config_generator_class=PureGauge,  # From last slide
    Ns=12, Nt=6, beta=4,               # Pure gauge parameters

    # Other pure gauge params are taken care of by defaults

    N = 100,                           # Run binary 100 times
    req_time = 300,                    # Each binary call takes <5 minutes
    starter = "/path/to/seed_config",  # Start from an existing config
    stream seed = 42,                  # Metaseed to generate seeds for each binary
    nodes = 1                          # Number of nodes to run on
)

job_pool = cfg_pool

## BOILERPLATE LAUNCHING CODE (Some machine dependence)
```

taxi example: Wilson Flow (Task class)

All fits on one slide!

```
import os
from taxi.mcmc import ConfigMeasurement
import taxi.local.local_taxi as local_taxi
from taxi._utility import sanitized_path
flow_input_template = """
prompt 0

nx {Ns}
ny {Ns}
nz {Ns}
nt {Nt}

epsilon {epsilon}
tmax {tmax}
minE {minE}
mindE {mindE}

reload_serial {loadg}
forget

EOF
"""
```

```
class Flow(ConfigMeasurement):
    def __init__(self, tmax, req_time=600, minE=0, mindE=0.0, epsilon=0.1,
                 Ns=None, Nt=None, **kwargs):
        super(FlowJob, self).__init__(req_time=req_time, **kwargs)
        self.tmax = tmax
        self.minE = minE
        self.mindE = mindE
        self.epsilon = epsilon
        if Ns is not None:
            self.Ns = Ns
        if Nt is not None:
            self.Nt = Nt
        if tmax == 0:
            assert minE != 0 or mindE != 0, \
                "If tmax is 0, must set adaptive flow parameters or flow will be trivial"
        self.binary = local_taxi.flow_binary
        self.loadg = sanitized_path(self.loadg)
        self.generate_outfilename()
    def generate_outfilename(self):
        self.fout = "flow_" + self.ensemble_name + "_{}".format(self.traj)
    def parse_params_from_loadg(self):
        words = os.path.basename(self.loadg).split('_')
        return {'Ns' : int(words[1]), 'Nt' : int(words[2]),
                'traj': int(words[-1]), 'ensemble_name' : '_'.join(words[1:-1])}
    def build_input_string(self):
        input_dict = self.to_dict()
        return flow_input_template.format(**input_dict)
```

taxi example: pure gauge (Run specification)

Convenience functions make use of common MCMC structure

Small addition to script: run Wilson Flow on gauge configs as they come out:

```
## BOILERPLATE CODE (Some machine dependence)

cfg_pool = ...CODE FROM LAST SLIDE...

flow_pool = measure_on_config_generators(
    config_measurement_class=Flow, # Subclass of ConfigMeasurement, just as easy to make
                                   # as subclassing PureGauge from ConfigGenerator
    tmax = 1, epsilon=.01          # Flow integrator parameters
    config_generators = cfg_pool,  # List of ConfigGenerators, run flow on their output
    start_at_traj = 1000,          # Let the stream equilibrate
    req_time = 60,                 # Takes <1 minute to run
)

job_pool = cfg_pool + flow_pool

## BOILERPLATE LAUNCHING CODE (Some machine dependence)
```

Future: “Close the loop” on pipeline

Use processed/analyzed data coming out of pipeline to inform/automate how to run new data

Some progress on this already: automatic choice of which ensemble to get starting lattice from, instead of having to specify a gauge file manually

In development:

Automated phase diagram exploration: Trace out thermal transition, κ_c, etc.

Auto-statistics: If ensemble needs longer to equilibrate, or a fit is failing, run more configurations for this ensemble

HMC auto-tuning: Automatically find ideal integrator parameters

Summary

Can automate almost all of the data generation process

Decentralized workflow management is a graceful fit to typical lattice constraints

- Taxi-Dispatcher Model

taxi software package

- Lightweight Python workflow manager specialized to lattice/MCMC

Is the package available yet?

- *“Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law.”*
- Public release version in progress, coming soon
- Follow https://github.com/CULattice and/or watch for preprint -- arXiv:170?.????(?)