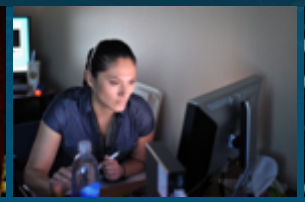
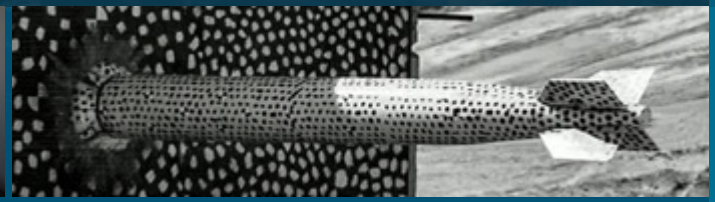
Managing execution of Dakota evaluations with Parsl





October 7, 2020

J. Adam Stephens

SAND2020-10549 C

What is Dakota?

An open-source (LGPL) **software package** developed at Sandia National Laboratories.

Provides scientists and engineers (analysts, designers, decision makers) greater perspective on their models' predictions:

- **Enhances understanding of risk** by quantifying margins/uncertainties
- **Improves products** through simulation-based design, calibration
- **Assesses simulation credibility** as part of a verification and validation process/workflow

A set of algorithms that can be applied to computational simulations in black box fashion for the purposes of **uncertainty quantification**, **optimization**, **calibration,** and **sensitivity analysis**.

How does it work?

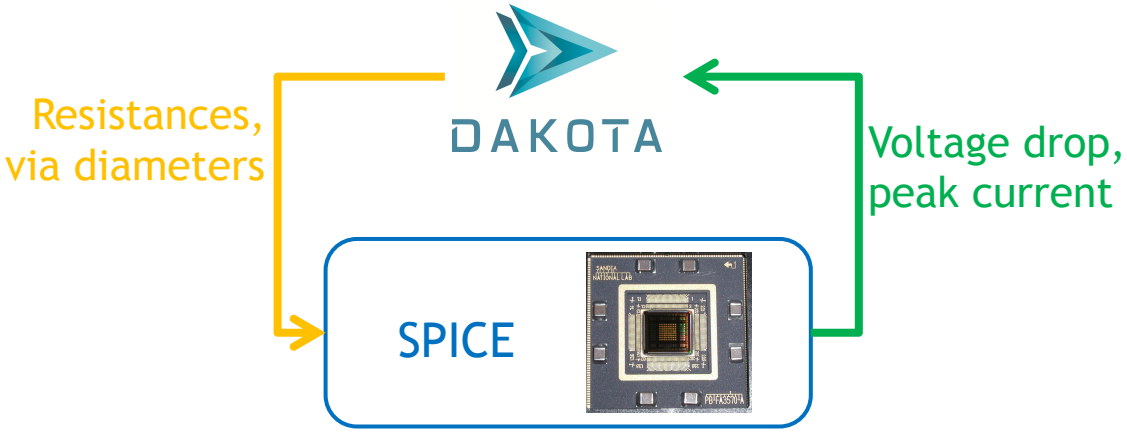
Dakota runs ensembles of simulations

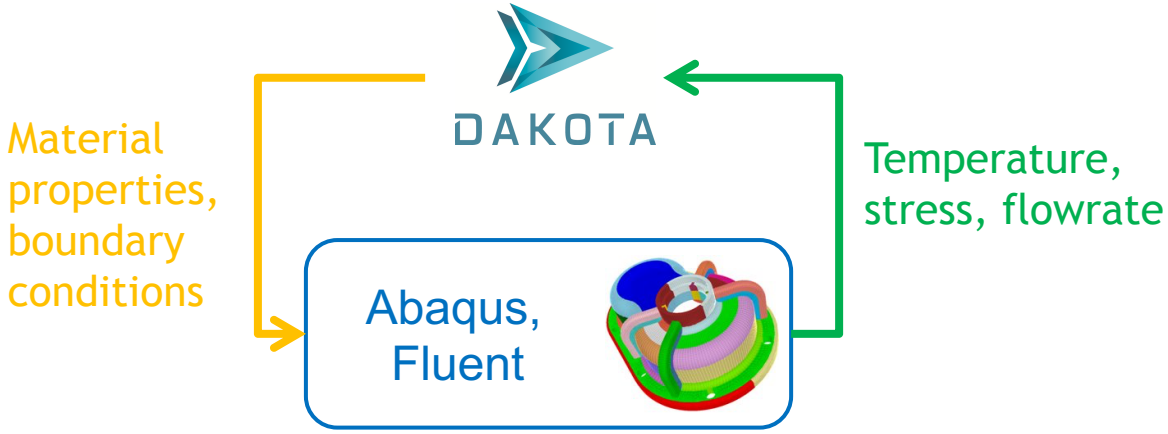
For each, it provides strategically selected inputs and collects outputs—communicates using file I/O

Each run of the simulation (parameter-response mapping) is an ***evaluation***

Dakota runs simulations using a user-developed driver

Nearly always used locally (workstation, laptop) or on HPCs





Common Evaluation Strategies and their Challenges

Massively Serial

Many serial evaluations

Dakota must be run in parallel to manage work across nodes

Evaluation Tiling

MPI parallelized evaluations are subscheduled in one HPC job

- Dakota provides no robust tools to help users tile evaluations
- Evaluations within one Dakota study may have unequal resource requirements (duration, memory, cores)

Evaluation Submission

Each evaluation is submitted as a separate HPC job

Job management

Demonstrations

Three examples to explore using Parsl for **Massively Serial** and **Evaluation Tiling** strategies

Property	Textbook	Rosenbrock	Parallel Textbook
Description	Scalable (inputs and responses) test problem	Optimization test problem with two inputs and one response	MPI parallelized version of textbook
Workflow	Driver only	Preprocessing, driver, postprocessing	Driver only
Launcher	`SingleNodeLauncher`	`SingleNodeLauncher`	`SimpleLauncher`
App Types	`bash_app`	`python_apps` for pre- and postprocessing, `bash_app` for driver	`bash_app`

Massively Serial

Evaluation Tiling

Can Parsl Help?

Massively Serial	Clear Win: Parsl does exactly what we need, and is intuitive to boot!
Evaluation Tiling	Useful, with a couple of caveats: • No way to express Apps' resource requirements • Support for landing MPI-parallelized tasks
Evaluation Submission	Could be used, but overkill

For all three use cases, task sandboxing is a highly desirable feature