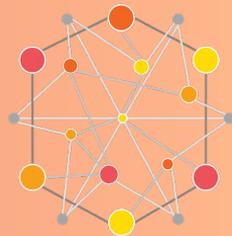


An update on the ExaWorks Project

Parsl & FuncX Fest '22

Daniel Laney

LLNL-PRES-835634



The Team



Dan Laney

Lawrence Livermore National Laboratory



Aymen Alsaadi

Brookhaven National Laboratory



Zeke Morton

Lawrence Livermore National Laboratory



Kyle Chard

Argonne National Laboratory



Ben Clifford

Argonne National Laboratory



Mikhail Titov

Brookhaven National Laboratory



Shantenu Jha

Brookhaven National Laboratory



James Corbett

Lawrence Livermore National Laboratory



Matteo Turilli

Brookhaven National Laboratory



Rafael Ferreira da Silva

Oak Ridge National Laboratory



Mihael Hategan

Argonne National Laboratory



Andreas Wilke

Argonne National Laboratory



Dong H. Ahn

Lawrence Livermore National Laboratory



Ketan Maheshwari

Oak Ridge National Laboratory



Justin M. Wozniak

Argonne National Laboratory



Todd Munson

Argonne National Laboratory



Andre Merzky

Brookhaven National Laboratory

Previous Contributors



Stephen Herbein

Lawrence Livermore National Laboratory



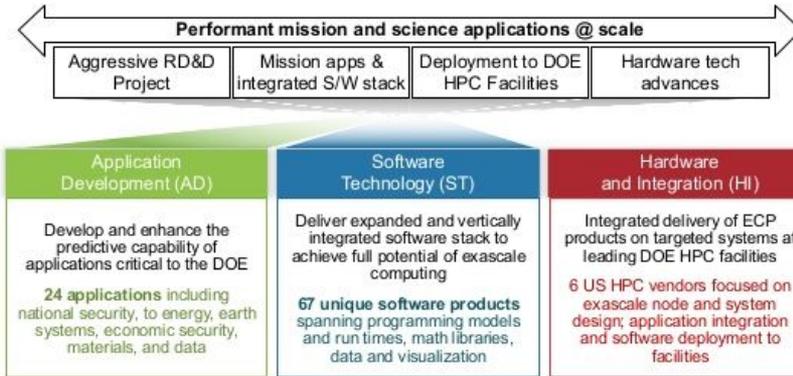
Yadu Babuji

Argonne National Laboratory

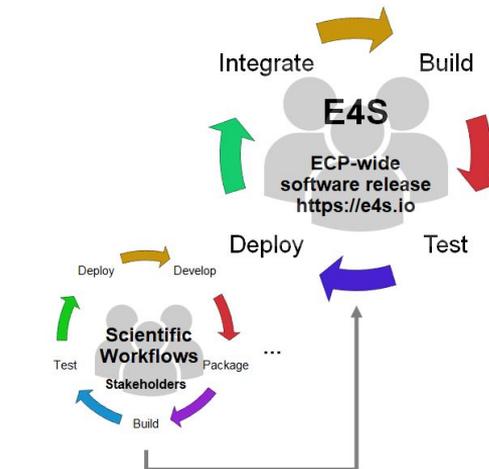
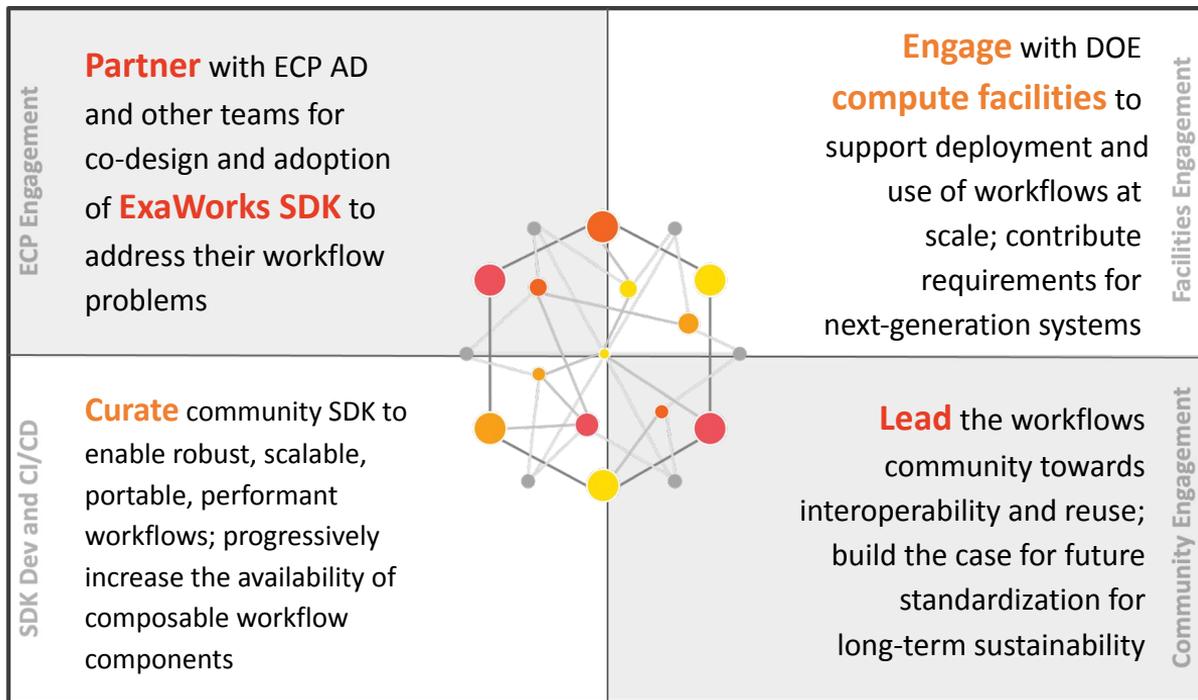
Exascale Computing Project (ECP)

Seven-year, \$1.8B project that aims to accelerate R&D, acquisition, and deployment of **exascale** computing capability to DOE

Six core national laboratories are focused on software, applications, hardware, system engineering and testbed platforms



Our approach will ensure exascale readiness of a wide range of ECP workflows and improve their long-term sustainability



The ExaWorks SDK is packaged, deployed, and tested using **E4S** and **ECP CI infrastructure**

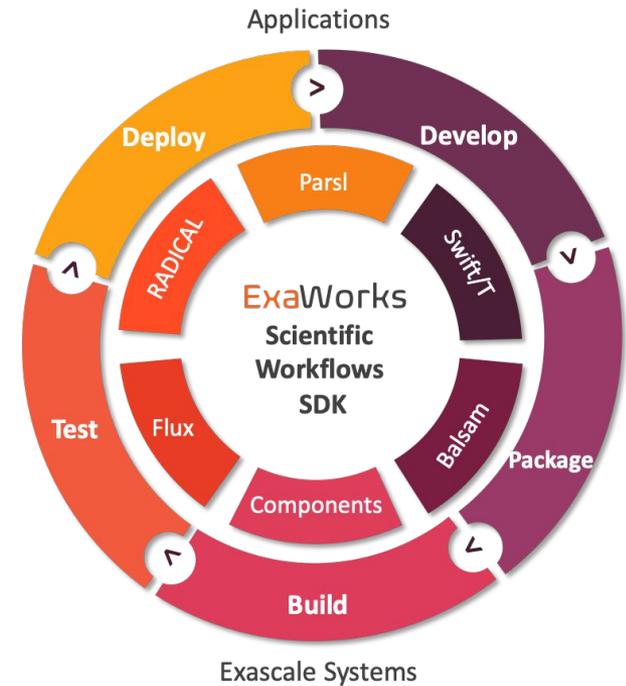
ExaWorks is *not* funded to build another workflow system

We are funded to provide a **production-grade Software Development Kit (SDK)** for exascale workflows

Our SDK democratizes access to hardened, scalable, and interoperable workflow management technologies and components

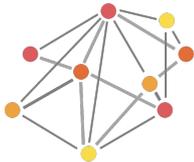
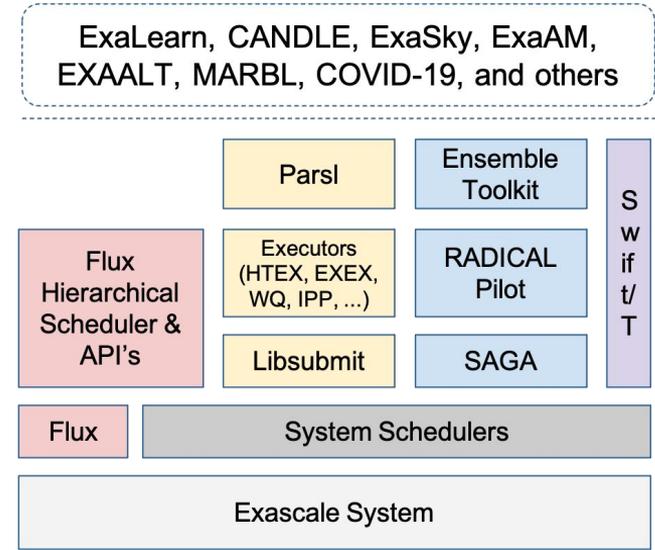
Approach

- Community policies for software quality (based on E4S)
- Open community-based design and implementation process
- Ensure scalability of components on **Exascale Systems**
- Standard packaging and testing
- Work toward shared capabilities in the SDK



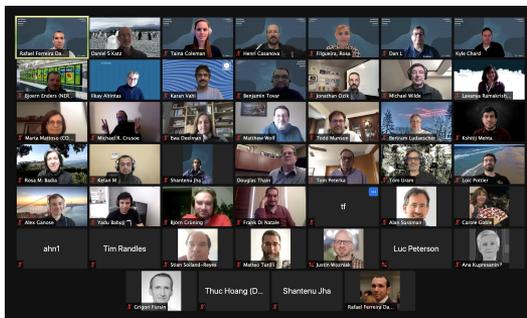
ExaWorks SDK brings together five seed technologies currently impacting ECP applications

- Scientific workflows SDK includes four seed technologies
 - **Flux** – hierarchical resource and job management software
 - **Parsl** – flexible and scalable parallel programming library for Python
 - **RADICAL** – component-based workflow middleware
 - **Swift/T** – high performance dataflow computing



We are engaging Workflow Communities and Computing Facilities

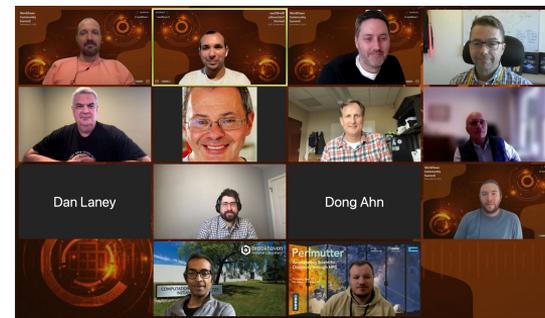
- **Workflows Community Summit: Researchers**
 - Brought together workflows leaders to develop a vision for community activities
 - <https://doi.org/10.5281/zenodo.4606958>
- **Workflows Community Summit: Developers**
 - Explored technical approaches for realizing the community vision
 - <https://doi.org/10.5281/zenodo.4915801>
- **Workflows Community Summit: Facilities**
 - Small group of facility representatives discussing facilities perspectives, challenges, and opportunities
- Invited Paper summarizing community roadmap: <https://arxiv.org/abs/2110.02168>



First Workflows Community Summit:
45 participants, 27+ workflow systems



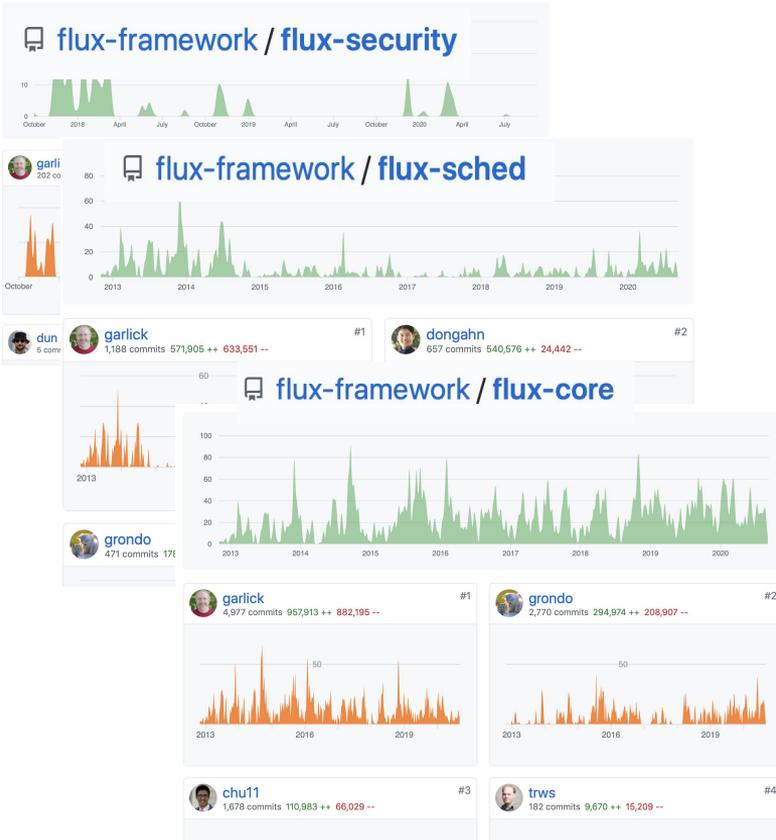
Second Workflows Community Summit:
75 participants



Third Workflows Community Summit:
9 participants, 8 facilities/centers (ALCF, OLCF, NERSC, LC, BNL, PSC, NREL, NCSA)



A portable, flexible next gen job scheduler for emerging workflows



- Open-source project in active development at flux-framework GitHub organization with ~15 team members
- Single-user and multi-user (a.k.a. system instance) modes
 - Single-user mode has been used in production for ~3 years
 - Multi-user mode is having its debut on LLNL's Linux clusters
- Plan of record for **LLNL's El Capitan exascale** system
- Designed with heterogeneous systems and advanced workflows in mind
- Rich Python and C/C++ API's

Parsl: a parallel programming library for Python

Apps define opportunities for **parallelism**
Python apps call Python functions
Bash apps call external applications

Apps return “futures”: a proxy for a result that might not yet be available

Apps run concurrently respecting dataflow dependencies. Natural parallel programming!

Parsl scripts are independent of where they run. Write once run anywhere!

Parsl scales to 100,000s of tasks on the largest HPC systems

```
pip install parsl
```

```
@python_app
def hello():
    return 'Hello World!'

print(hello().result())
```



Hello World!

```
@bash_app
def echo_hello(stdout='echo-hello.stdout'):
    return 'echo "Hello World!"'

echo_hello().result()

with open('echo-hello.stdout', 'r') as f:
    print(f.read())
```



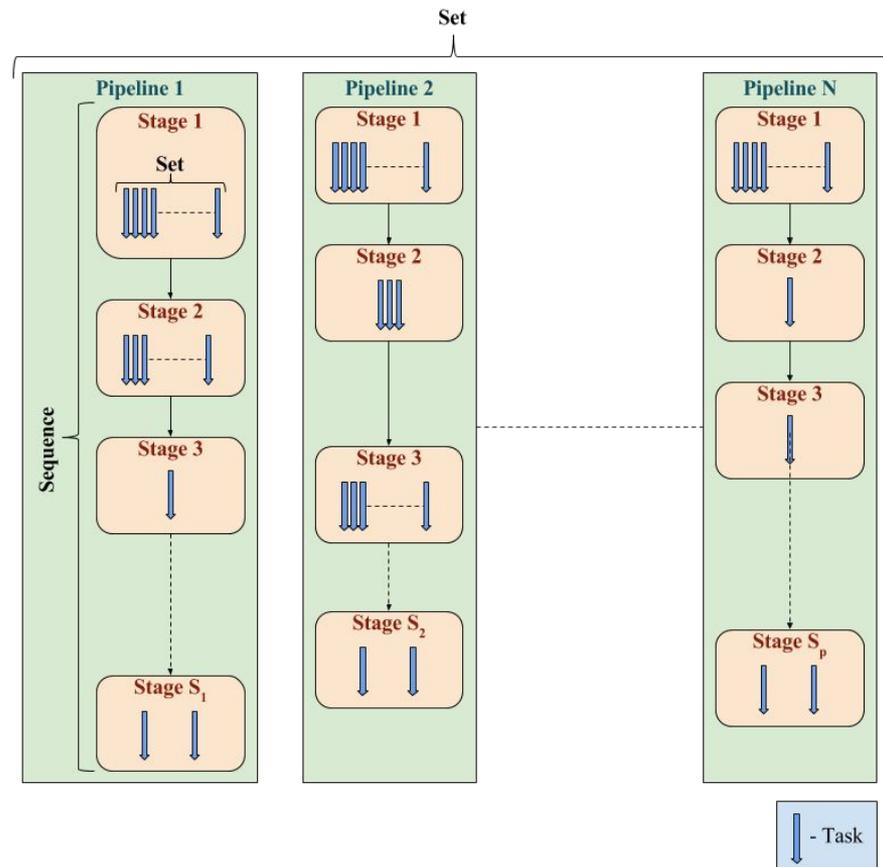
Hello World!

RADICAL Cybertools: scalable Python abstractions for workflows

RADICAL EnTk represents an ensemble application as a set of Pipelines.

Two (pythonic) collections of objects:

- **Set:** contains objects that have no relative order with each other
- **Sequence/List:** contains objects that have a linear order, i.e. object 'i' depends on object 'i-1'
- Pipelines can thus represent general DAG structures
- Pipelines can coordinate and communicate



Swift/T: Enabling high-performance scripted workflows

Write site-independent scripts, translates to MPI

Automatic task parallelization and data movement

Invoke native code, script fragments
in Python and R

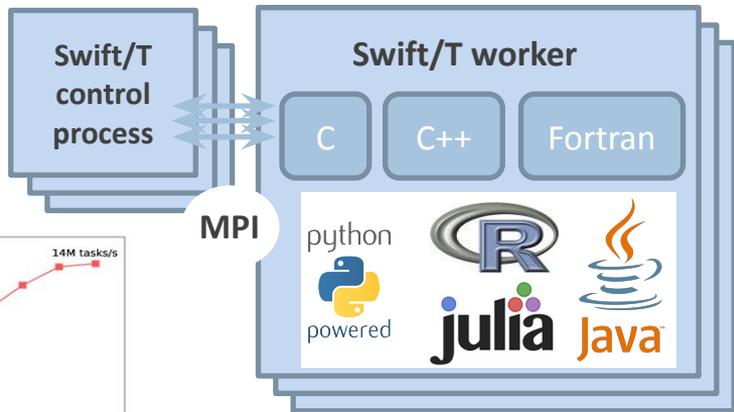
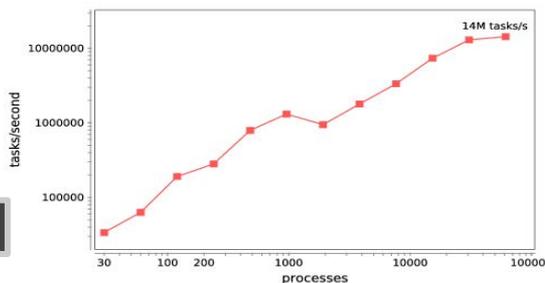
Rapidly subdivide large
partitions for MPI jobs
in multiple ways (MPI 3.0)



```
$ spack install stc
```

```
$ conda install -c lightsource2-tag swift-t
```

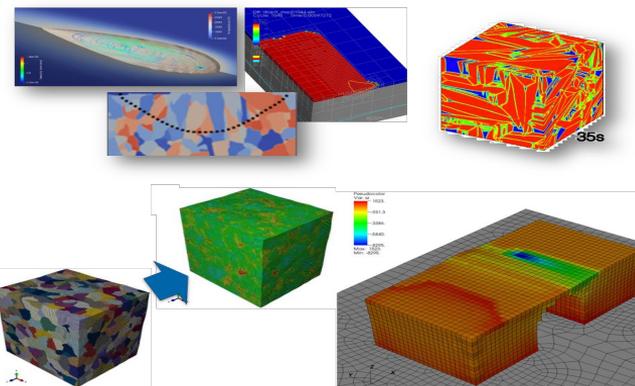
64K cores of Blue Waters
2 billion Python tasks
14 million Pythons/s



Swift/T: Scalable data flow programming for distributed-memory task-parallel applications
Proc. CCGrid 2013.

We are working closely with ECP Applications to impact deliverables

- **Approach:** Continuous engagement with ECP applications to address their workflow challenges and implement best practices, scalable, and performant workflows using the ExaWorks SDK.
- **ExaAM's** complex workflow simulates laser melt-pool additive manufacturing processes.
- **Colmena (ExaLearn):** open-source Python framework for ML-steering of simulation campaigns at scale.
- **CANDLE:** Relies on Swift/T for rapid development, scalability, and portability of DL-oriented cancer application suite on DOE systems
- **COVID:** National Virtual Biotechnology Lab used billions of core hours harnessed rapidly and effectively for heterogeneous workflows
- **Gordon Bell Prizes:** 3 of the 4 finalists used ExaWorks technologies



Gordon Bell submission	Description	ExaWorks Technologies Used
WINNER: <i>AI-Driven Multiscale Simulations Illuminate Mechanisms of SARS-CoV-2 Spike Dynamics</i>	Used DeepDriveMD built on RADICAL-Cybertools to steer ensembles of MD simulations using AI yielding 10x performance improvement ; part of CANDLE	Entire RADICAL stack: <ul style="list-style-type: none"> ▪ Ensemble-Toolkit ▪ RADICAL-Pilot ▪ SAGA
<i>Enabling Rapid COVID-19 Small Molecule Drug Design Through Scalable Deep Learning of Generative Models</i>	Flux is the scalable backbone of the Rapid COVID-19 Small Molecule Drug Design workflow whose scalable generative machine-learning task was featured in this paper; part of CANDLE and ExaLearn	The overall workflow is composed of Flux , the Maestro workflow manager, and a custom generative molecular design (GMD) workflow pipeline
<i>A Population Data-Driven Workflow for COVID-19 Modeling and Learning</i>	Swift/T managed a workflow containing the CityCOVID agent-based model and large numbers of small ML optimization tasks. The workflow consumed real-world infection data and produced data used by city and state governments	Swift/T managed the workflow

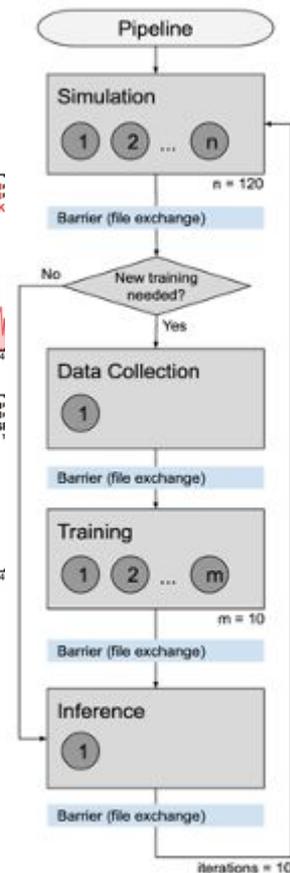
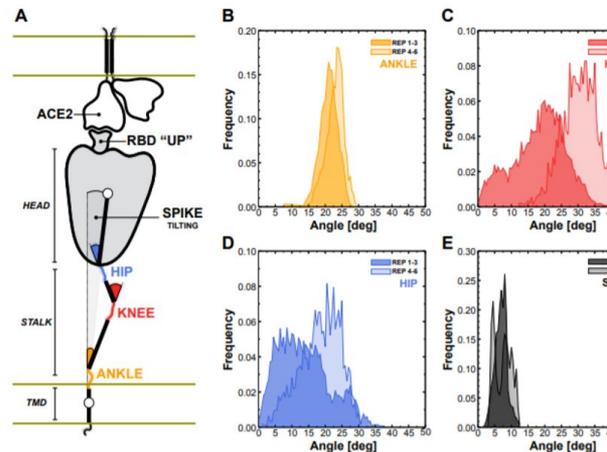
ExaWorks technologies were leveraged in 3 of 4 finalists and the Winner of the SC21 Gordon Bell Covid-19 Competition

The Winner:

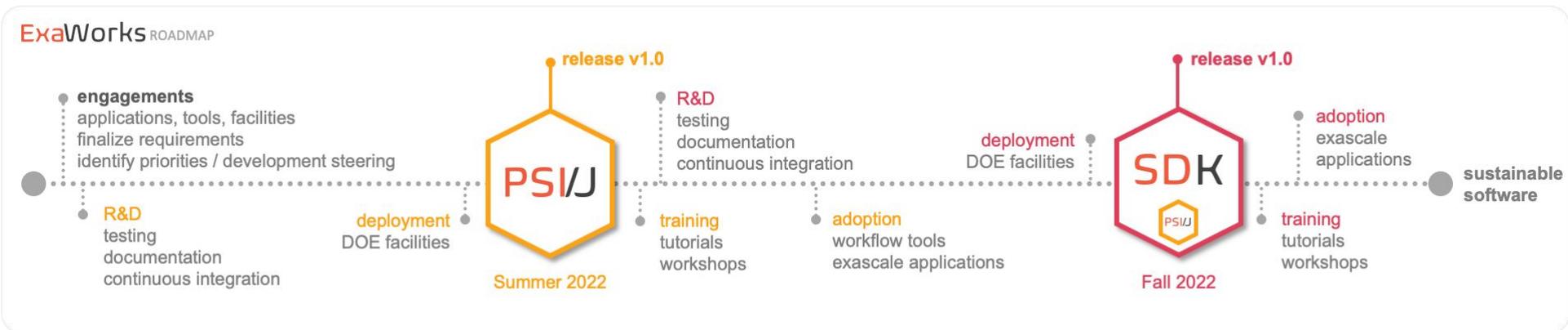
DeepDiveMD -- an extension of **RADICAL tools** -- workflow infrastructure adaptively couples ML + NAMD simulation workflow

Effective speedup of 1 order of magnitude sampling efficiency: with DeepDriveMD observed 25% more conformations of the knee bending in only 12% of the time!

RADICAL components of the ExaWorks tool set brought scalability, reliability, and agility to the project



ExaWorks RoadMap

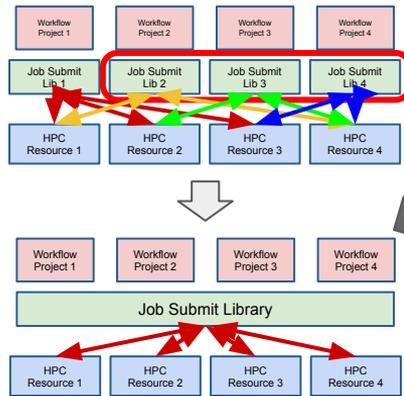


Exascale Workflows|Community

PSI/J was designed through an open community process

- Our survey, interviews, and co-design meetings highlighted need for portability layer for schedulers
- Community desired a light-weight user-space API
- Initial Python implementation is nearing version 1.0 release
 - Support for Slurm, LSF, Cobalt, Flux, RCT, SAGA
 - Working to add next set of schedulers (e.g., PBS)
 - Architected to allow seamless contributions from the community

1. Problem definition



2. Community specification

(<http://exaworks.org/job-api-spec/specification.html>)

Introduction

The purpose of this document is to provide an analysis of the design and implementation issues of a job management API suitable for managing jobs that are on research machines, as well as propose such an API. A job management API is a set of interfaces that allow the specification and management of the invocation of application executables. The corresponding implementation of a job management API is a job management library. A job management library, through its API, is intended to be used as a client application. Traditionally, job management is implemented on supercomputers by local Resource Managers (LRMs), such as PBS/Torque, SLURM, etc. To a first approximation, a job management API is understood as an abstraction layer on top of various LRMs.

A Note About Code Samples

There are various locations in this document where code is used to provide examples. Such code is not working code, but a `bashC++/R` inspired pseudo-code which almost surely will require modification to be usable.

Motivation and Design Goals

It is designed with a number of goals in mind. These are centered around usability and keeping the API and potential extensions simple, where trade-offs must be made, such as deciding between transfer and implementation simplicity. Usability is generally favored. However, an attempt is also made to draft and analyze the resulting complexities of a implementation in order to minimize the amount of work needed to write such an implementation.

Lastly, the following aspects have informed the design in a significant fashion:

- The proposed API is **asynchronous**. A detailed discussion about the choice between synchronous and asynchronous APIs can be found in Appendix B. In short, the implementation of a synchronous API would not scale well in most languages. Additionally, if (or needed) the API provides a **wait()** method that allows direct code to block, implements a synchronous wrapper around the API.
- Bulk versions of calls have been considered. The main reason for having bulk calls is to facilitate the use of more efficient mechanisms for transmitting job information to an underlying implementation. However, alternative methods exist that do not require a bulk call. Nonetheless, adding bulk calls to enable better performance in layers 1-2, or even in layer 3 if reasonably justified in the future, remains a possibility. For a technical discussion on the topic, please see Appendix C.

Layers

There are at least three major ways in which a job management API can be used:

- **Local** the relevant API functions are invoked by programs running on the target resource (or a specific node on the target resource, such as a log/head node).
- **Remote** the API functions are invoked by programs running on a different resource than the target resource (this requires some form of distributed architecture).

```
import jpsi

jex = jpsi.JobExecutor.get_instance('slurm')

def make_job():
    job = jpsi.Job()
    spec = jpsi.JobSpec()
    spec.executable = '/bin/sleep'
    spec.arguments = ['-l10']
    job.spec = spec
    return job

jobs = []
for i in range(N):
    job = make_job()
    jobs.append(job)
    jex.submit(job)

for i in range(N):
    jobs[i].wait()
```

4. Community SDK component

Author - Label - Projects - Assignees - Sort

- JobExecutor constructor missing 17
- API Compatibility Across Implementations? **Open** 15
- Interaction between workflow system and scheduler (malleable scheduling) **Open** 15
- waitfor proposal **enhancement** 13
- move 'cancel' to job **enhancement** 12
- Accommodating short-lived clients 11
- Pre, post, scripts, expansion, etc. **Open/Assumption** 6
- Wish list from Balazs **Open/Assumption** 7
- Add some text about pilot jobs / resiling **Open** 6
- Add a 'executor.list' method on the executor **enhancement** 6
- Launchers discussion/requirements **Open/Assumption** 5

3. Open discussion

ExaWorks is working towards a production quality continuous integration and deployment infrastructure for workflow tools

We have developed a GitLab CI infrastructure

We have set up CI at LLNL, ORNL, and ANL for the SDK components

We are testing PSI/J on an ECP testing cluster

We have developed a testing server to collect results of tests and enable dashboards and reporting from multiple sites

We are creating **Status Dashboard** to view what tests have been run on which systems

The screenshot shows a dashboard titled "Exascale dashboard testing service" with a sub-header "PSI/J Tests". The dashboard features a navigation bar with icons for home, calendar, grid, list, and settings. Below the navigation bar is a table with the following columns: "Tests Suite", "Tests", "Quickstart example", and "Simple Ensemble". The table lists test results for four sites: University of Oregon, Lawrence Livermore National Lab, Nercs National Lab, and Oakridge National Lab. Each site has a summary row and a list of specific test instances.

	Tests Suite	Tests	Quickstart example	Simple Ensemble
▼University of Oregon	●	●	●	●
axis1.hidden.uoregon.edu	○	●	●	●
illyad.hidden.uoregon.edu	○	●	●	●
jupiter.hidden.uoregon.edu	○	●	●	●
reptar.hidden.uoregon.edu	○	●	●	●
saturn.hidden.uoregon.edu	○	●	●	●
▼Lawrence Livermore National Lab	●	●	●	●
▼Nercs National Lab	●	●	●	●
cori08.nersc.gov	○	●	●	●
▼Oakridge National Lab	●	●	●	●

PSI/J: Portable Submission Interface for Jobs

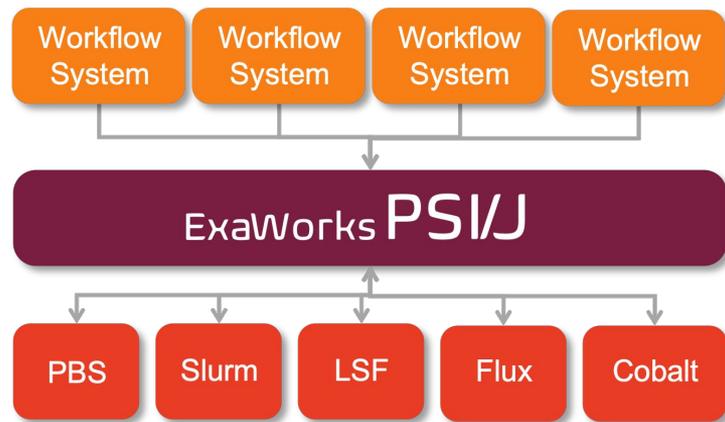
A set of **interfaces** that allow the specification and management of “jobs”

Support for Slurm, LSF, Cobalt, Flux, PBS

Open document to define a language-independent specification

Community specification

<http://exaworks.org/job-api-spec/specification.html>



PSI/J Python binding provides an intuitive Python-futures based API for job management

- PSI/J Python binding
 - Python library with asynchronous interface for interacting with job schedulers
 - Support for Slurm, LSF, Cobalt, Flux, RCT, SAGA
 - Working to add next set of schedulers (e.g., PBS)
 - Architected to allow seamless contributions from the community
- Eventually the PSI/J specification will cover more advanced job-management functionality, such as job submission on remote clusters (“layer 1”).
 - All effort so far has been on “Layer 0”, in which PSI/J talks only to the local resource manager.
- We have integrated PSI/J into both RADICAL CyberTools and Parsl

```
import jpsi

jex = jpsi.JobExecutor.get_instance('slurm')

def make_job():
    job = jpsi.Job()
    spec = jpsi.JobSpec()
    spec.executable = '/bin/sleep'
    spec.arguments = ['10']
    job.spec = spec
    return job

jobs = []
for i in range(N):
    job = make_job()
    jobs.append(job)
    jex.submit(job)

for i in range(N):
    jobs[i].wait()
```

Learn more...

<https://exaworks.org>

- Join our Slack Channel
- Read the documentation

Tutorial Sessions

- ISC-HPC (May 2022)
- PEARC (July 2022)

Engagements

- Get in touch to discuss how ExaWorks components can benefit your project

The screenshot shows the ExaWorks website homepage. At the top, there is a navigation bar with links for 'PS/J', 'SDK', 'Community', and 'About'. The main header features the ExaWorks logo and the tagline 'Technologies for Composable and Scalable HPC Workflows'. Below this, a central section describes the ExaWorks SDK, stating it provides access to a collection of hardened and tested workflow technologies. To the right of this text is a circular diagram representing the workflow process, with stages: Applications, Develop, Build, Composites, Test, and Deploy. Below the diagram is a 'Learn more' button. The lower section of the page highlights the ExaWorks PSI/J component, described as a portability layer across different HPC workload managers. This section includes a diagram showing 'Workflow System' and 'HPC Resource' blocks, and another 'Learn more' button. At the bottom, there is a section for 'ExaWorks: Workflows for Exascale', which includes a reference to a research paper presented at the 2021 IEEE Workshop on Workflows in Support of Large-Scale Science (WORKS), November 2021, and a 'Read the Paper' button.

Thank you!

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC