



The University of Texas at Austin
Oden Institute for Computational
Engineering and Sciences

Exascale Predictive Simulation of Inductively Coupled Plasma Torches

Bob Moser and George Biros

Oden Institute for Computational Engineering and Sciences, UT-Austin

PSAAP3 Kickoff Meeting, 18 August 2020



PECOS

Predictive Engineering and Computational Sciences

Texas MSC Project Objectives

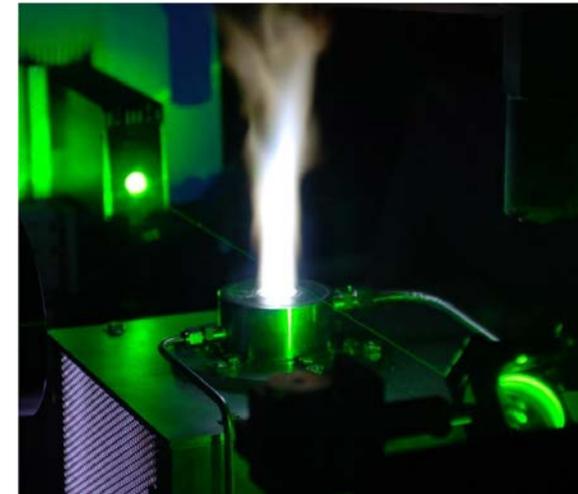
Perform predictive simulations of an inductively coupled plasma (ICP) torch

- Predict outlet flow characteristics and limits of stable operation
- Consider Argon and air feed gas, and varying flow rate and pressure
- Make validated predictions with quantified uncertainty
- Identify processes that limit stable operation
- Enable improved ICP torch design

We have an ICP torch facility at the Pickle research campus

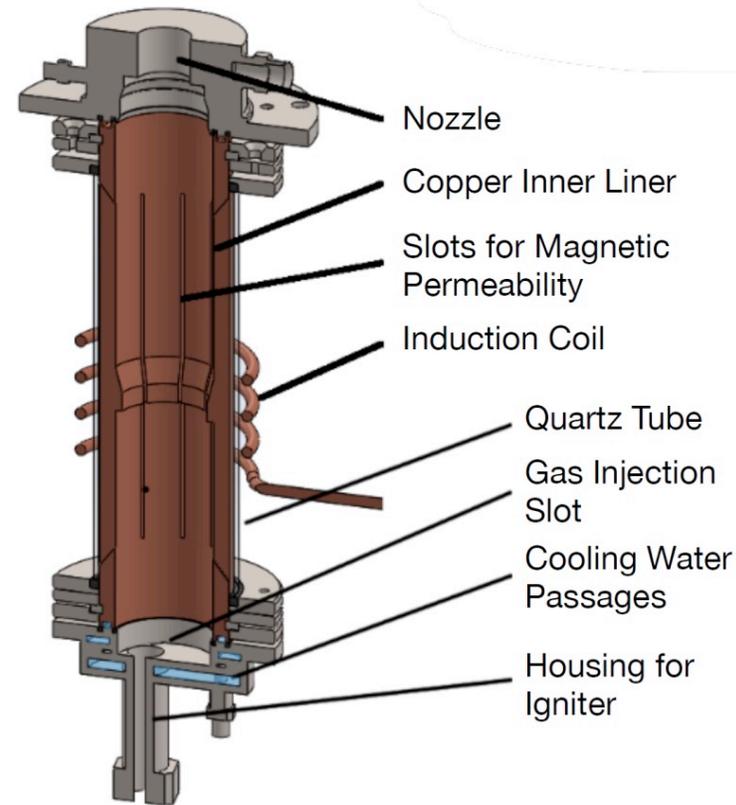
Advance computational science capabilities

- Advance plasma physics modeling
- Advance exascale computational performance and productivity
- Advance exascale and UQ algorithms
- Advance predictive validation and UQ
- To the benefit of the NNSA labs



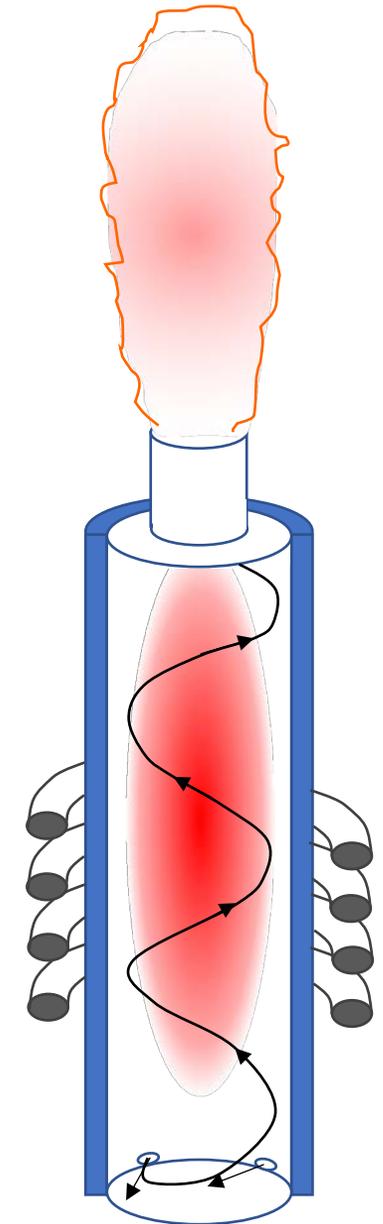
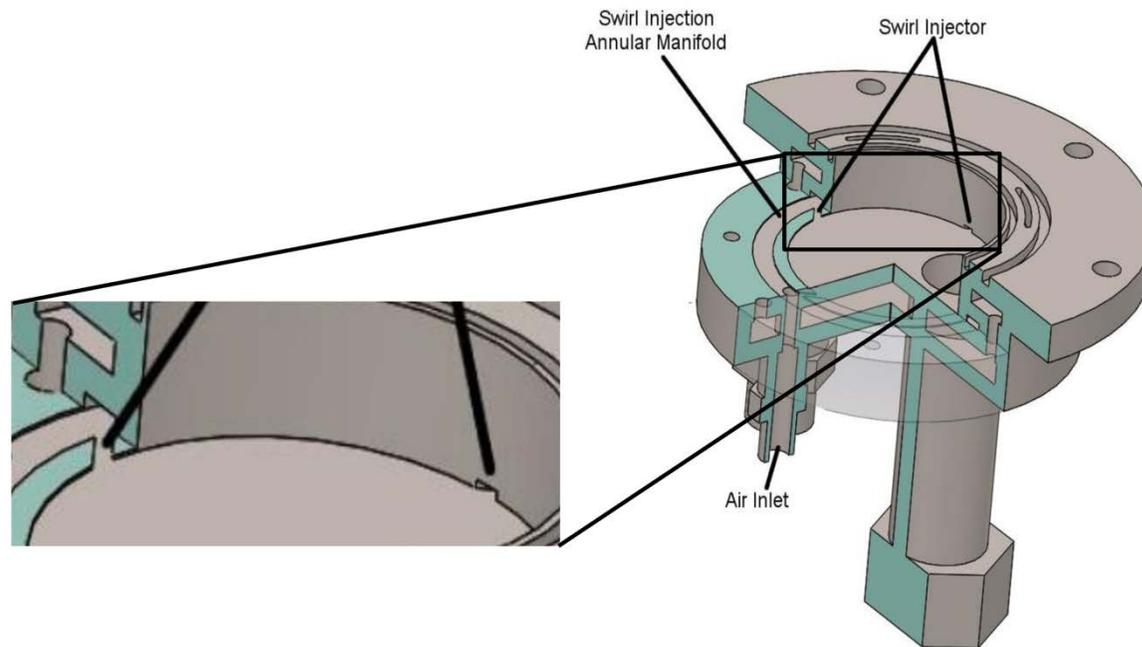
Inductively Coupled Plasma Torch

- Feed gas enters at bottom
- Power deposited into gas by RF induction coil
- Electromagnetic fields accelerate electrons, which heat and ionize through collisions
- Peak plasma temperature $\sim 10,000\text{K}$ with 1% ionization
- The relatively high pressure outlet plasma can be used for various purposes

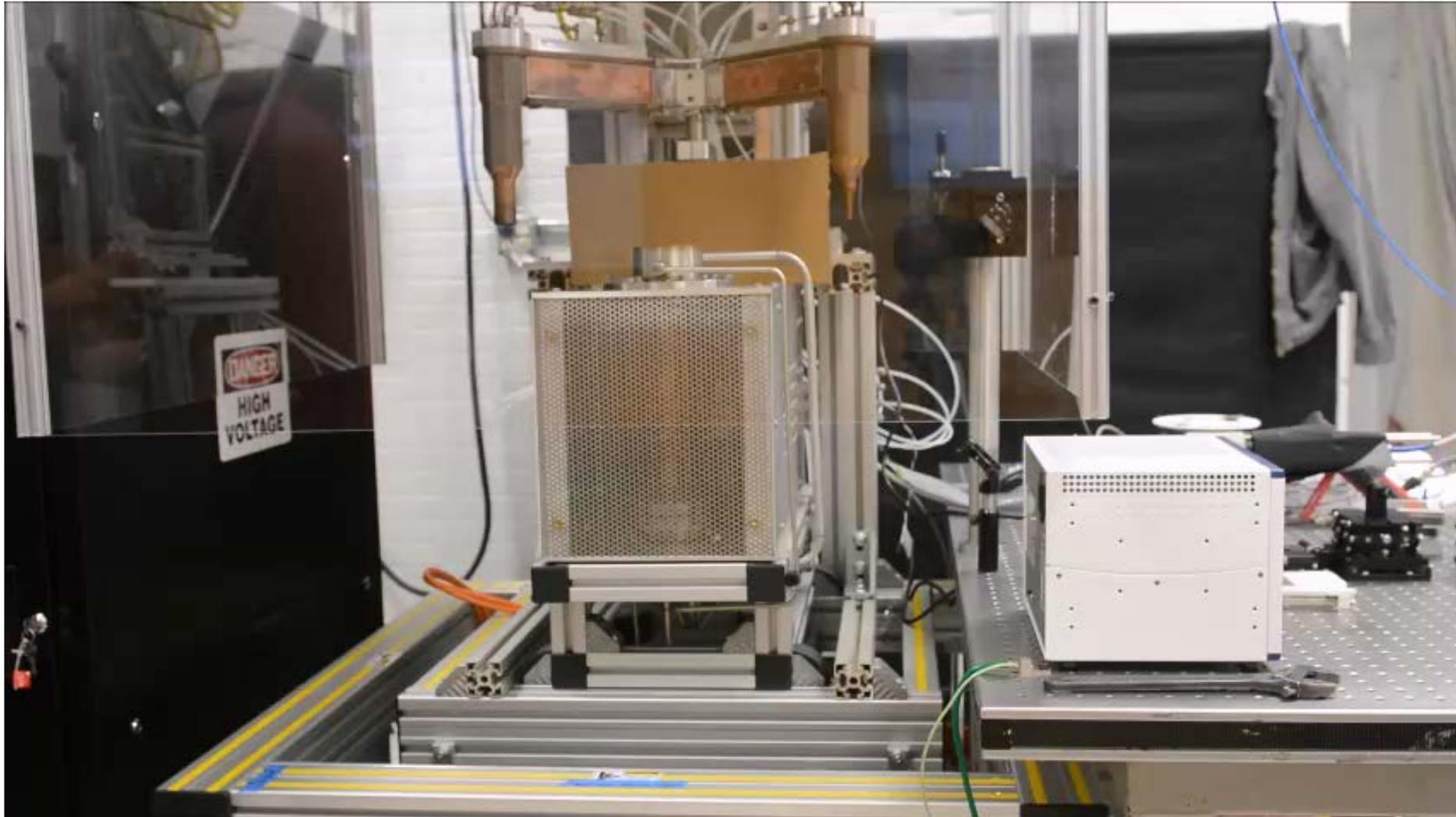


ICP Torch Characteristics

- High-speed tangential inlet jets introduce swirl
- Swirl stabilizes plasma by segregating low density gas toward the center
- Flow in the hot region is laminar, while inlet and exit jets are turbulent
- Thermo-chemical state of the outlet jet is of interest in applications
- Stable operation only possible over limited range of conditions (flow rate, feed gas, pressure) but governing mechanism not well understood



ICP Torch at UT Austin



ICP Torch Applications

ICP torches have a wide variety of uses

- Materials testing for thermal protection systems
- Gas pyrolysis (e.g. hazardous material breakdown)
- Material synthesis
- Coating deposition

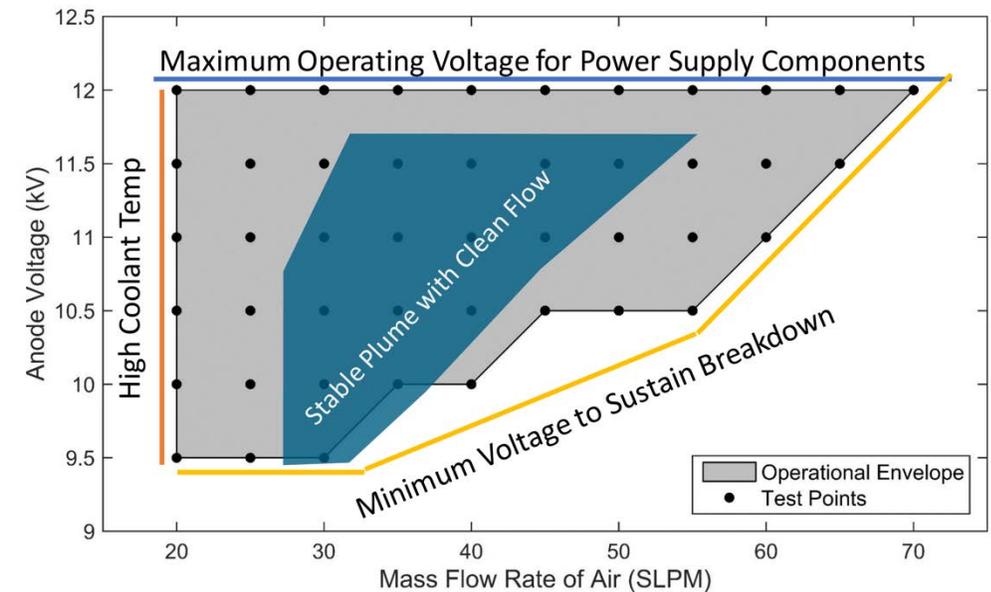
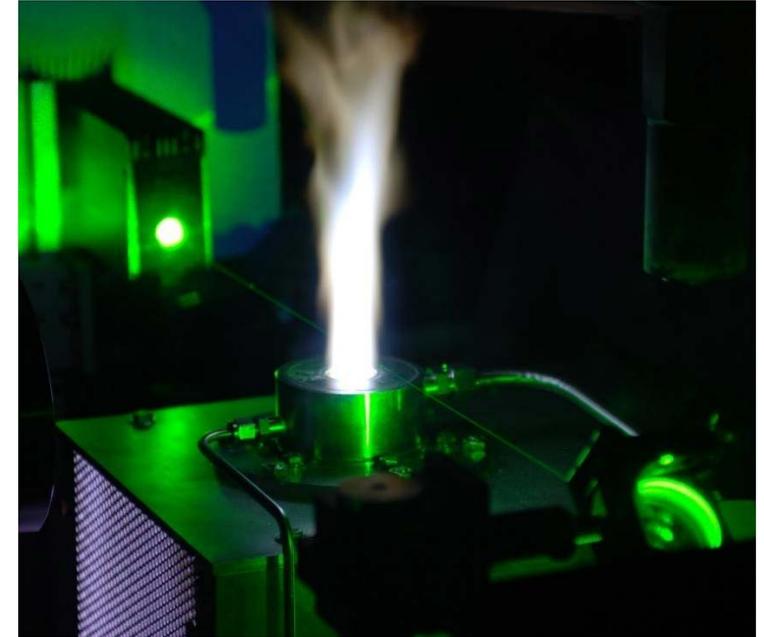
Plasmas in similar regimes are common

- Torch may serve as a laboratory surrogate for plasmas observed in other applications

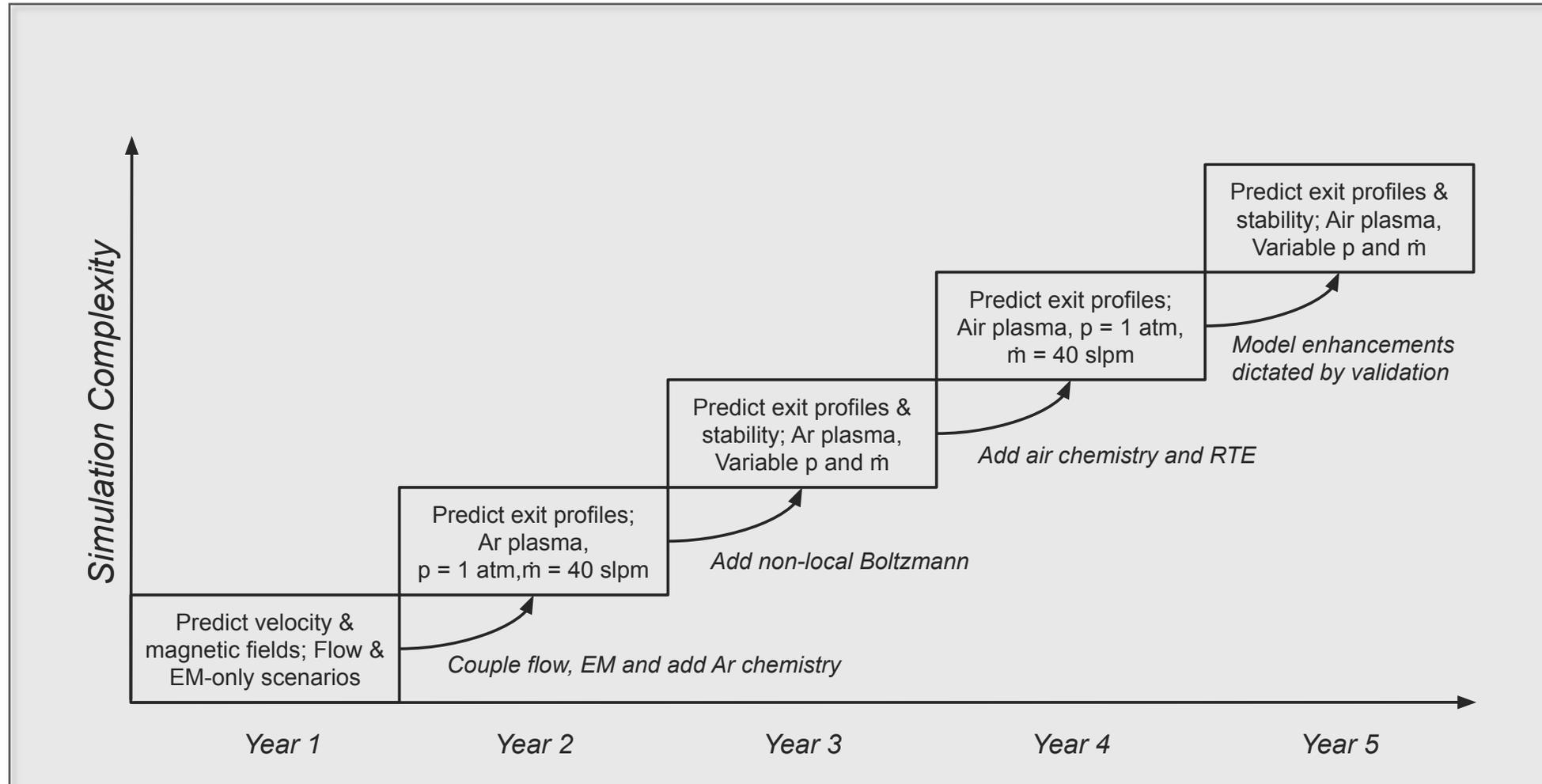


Science Questions

- ICP torch performance in applications depends on outlet plasma properties
 - What physical mechanisms control performance?
 - What level of physics modeling necessary for reliable predictions?
- Only operates stably over a restricted range of conditions
 - Predict limits of stable operation
 - What processes dictate these limits?



Simulation Milestone Objectives



ICP Torch Physics Overview

The torch represents a highly complex, multi-physics, multi-scale simulation challenge

Important Physical Phenomena

- Fluid & plasma flow
- Electromagnetism
- Non-equilibrium plasma dynamics
 - Ground- and excited-state neutral and charged species
 - Wide range of transformation/excitation reactions
 - Species and thermal transport
 - Heavies: Maxwellian translational energy distribution
 - Electrons: non-Maxwellian energy distribution
- Quasi-neutral plasma
- Participating media radiative energy transfer

Relevant Scales

- $D = 6\text{cm}$, $D_{\text{noz}} = 3\text{cm}$, $L = 25\text{cm}$
- Inlet jet width: 0.5mm
- $V \sim 10\text{m/s}$ (chamber), 300m/s (inlet)
- Turbulent scales near inlet: 10μ
- Excitation frequency: 6MHz
- EM wavelength: 50m
- EM skin depth: 6mm
- Wall sheath thickness: 50μ
- Electron mean free path: $7\text{-}70\mu$
- Heavies mean free path: 2μ

Resolution and Computation Requirements

Straightforward Representation

- Air feed gas, ~40 species (with excited states)
- 10^{11} grid points
- 4000 DOF/point for Boltzmann & RTE
- 10^{17} flop/step $\times 10^7$ steps
- 300 exaflop-hours

Discretization & Algorithms Research

- Multiple spatial grids
- New Boltzmann & RTE representations
- Multirate-timestepping
- $\rightarrow 10^{11}$ DOF, 30 petaflop-hours

For Industrial Applications

- Perhaps 4 times bigger
- Complex hydrocarbon feed gas
- 1000 times computational cost

Research challenges

- Physical modeling
- Efficient discretizations
- Validation and uncertainty quantification
- Exascale computer science
- Exascale algorithms

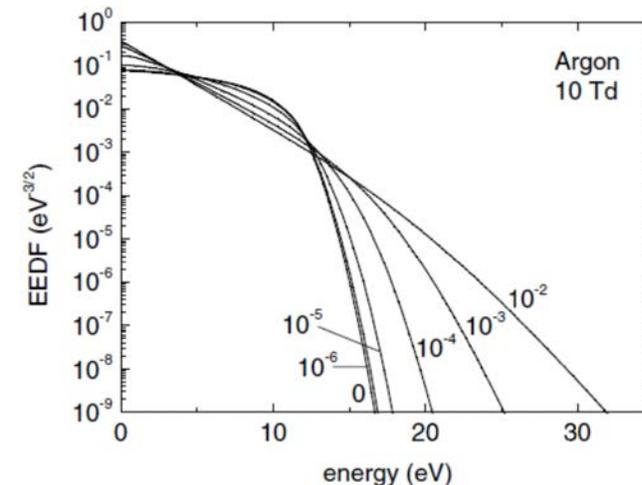
Physical Modeling Approach

Initial Model Components

- Compressible Navier-Stokes (DNS)
- Darwin approximation to Maxwell's equations
- Multi-species transport, including excited states
- Mass action reaction kinetics
- Separate heavy species and electron energy equations
- Non-Maxwellian electron kinetics from local Boltzmann
- Optically thick radiative transport for resonant lines, otherwise optically thin

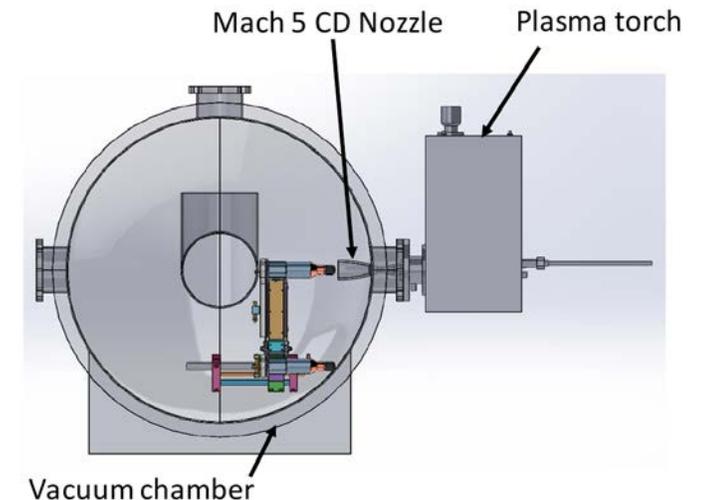
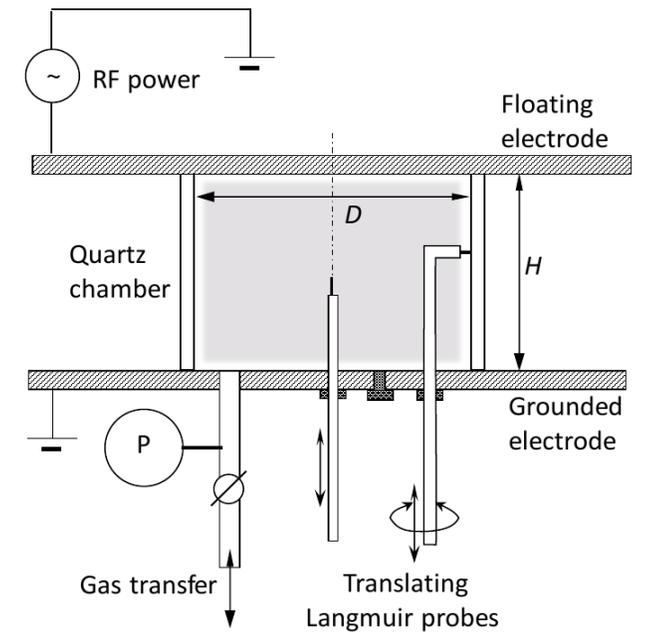
Refinements for More Challenging Cases

- 6-D Boltzmann for electron kinetics—needed to explore boundaries of stable operation
- Radiative transfer equation for lines that are neither optically thick nor thin—needed for air
- Other refinements as driven by validation



Model Validation

- Most aspects of physical model may be validated outside of the torch via experiments accessing relevant plasma regimes
- 6 MHz glow discharge device at $P \sim 1-100$ Torr gives electron density similar to torch, useful for validating
 - Electron kinetics
 - EM coupling of energy into the plasma
 - Wall sheath representation
- Mach 5 shock-induced non-equilibrium facility
 - Sub-mm spatially resolved measurements along stagnation line will be used to test models of non-equilibrium kinetics
- Full model will also be validated against torch measurements
 - Measure temperatures and densities in plasma chamber and jet exit
 - Determine limits of stable operations
 - Assess both models and validation process itself



UQ Algorithmic Research

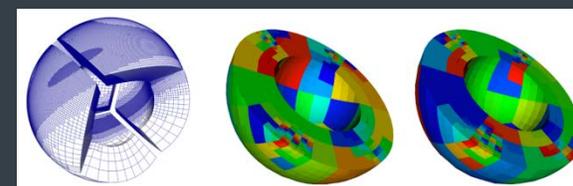
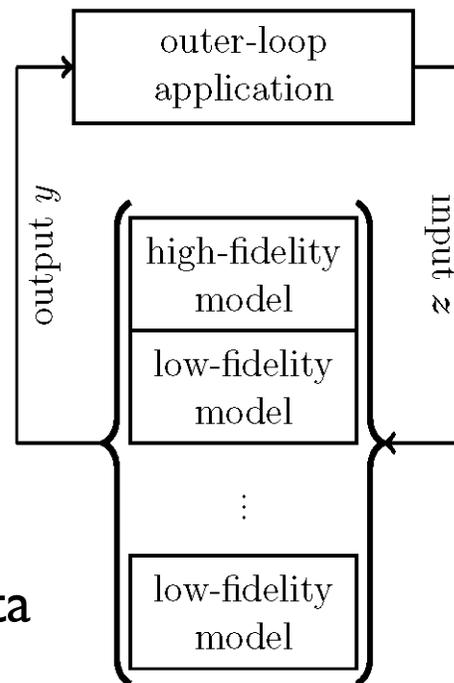
Need new UQ methods and algorithms that overcome the challenges of (1) extreme computational cost of the forward simulation, (2) multiple disparate sources of uncertainty / high-dimensional uncertain parameters

- Multifidelity approaches
- Decomposition approaches

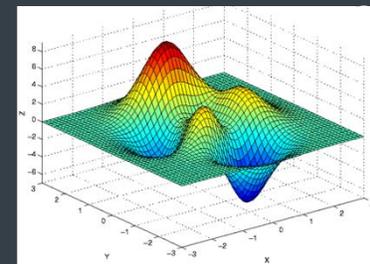
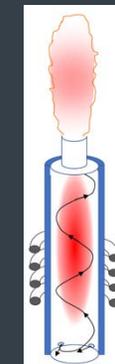


Multifidelity UQ

- Multifidelity methods: Leverage cheap approximate models to accelerate a UQ task
- Challenges:
 - number of evaluations of high-fidelity model is extremely limited
 - highly nonlinear, chaotic behavior
 - more complicated Qols (e.g., locating the stability boundary)
- Approaches:
 - new approaches to exploit relationships among models, and with experimental data
 - leverage error estimates
 - leverage multifidelity sensitivity/adjoint information



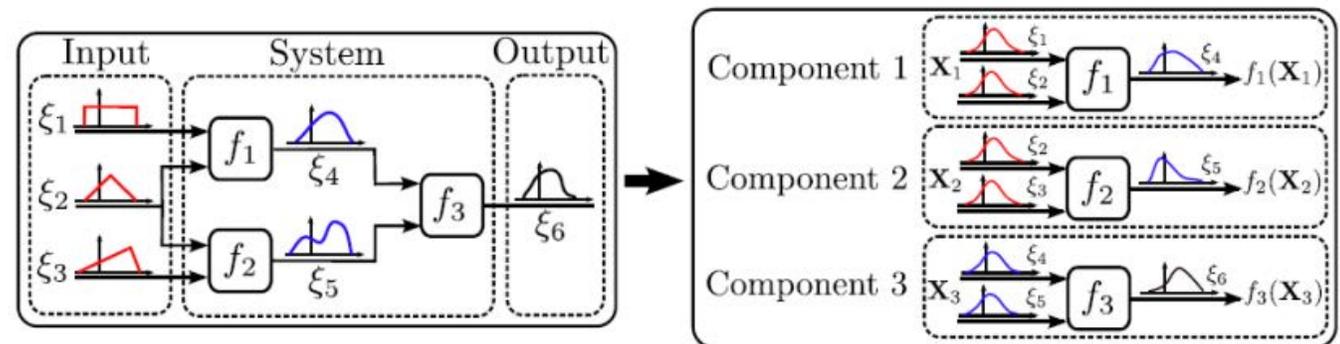
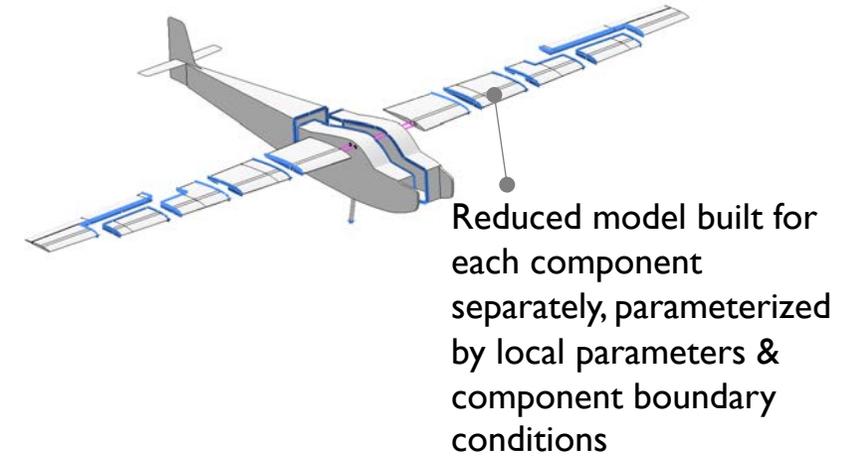
We will exploit coarse grids, simplified physics, closure models, geometric reductions, data-fit surrogates, physics-based reduced models, ...



Decomposition Strategies for Scalable UQ

“Divide & Conquer”

- Challenges:
 - hundreds of parameters – need to identify “optimal” decompositions that may go beyond physical intuition
 - composition: re-integration of local component-level UQ to provably recover system-level UQ results
- Approaches:
 - build on Static Condensation Reduced Basis Method and Domain Decomposition UQ
 - exploit information from lower fidelity models (e.g., sensitivities) to identify decompositions
 - explore machine learning methods to identify decompositions



Exascale



Electron Boltzmann and Radiation

Electron Boltzmann equation

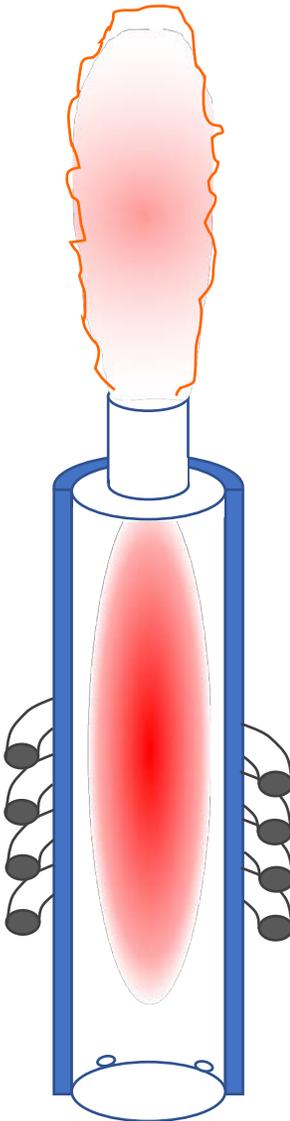
- 6D+time advection driven by collision integral
- Spatially adaptive discretization in both space (static) and velocity (dynamic), capitalizing on our work on [fast spherical harmonics methods](#)
- Treatment of advection: Both explicit as well as semi-Lagrangian for larger CFL, capitalizing on our work on [scalable semi-Lagrangian methods on heterogeneous architectures](#)

Radiative transfer equation

- 5D+quasi-steady in time with composition & temperature dependent coefficients
- Static space adaptivity with [dynamic angle adaptivity](#), using coarser (than flow) space/time discretization, also exploiting [fast spherical harmonics methods](#)
- [DG discretization](#) and Krylov solvers for spatial operators
- [Parallel sweeping methods](#) (connection to graph algorithms)

Turbulent flow + species advection

- High order finite elements on unstructured meshes
- Locally hp-refined meshes for boundary layers and turbulent inlet
- Investigating **multirate time stepping** to handle wide range of flow scales (highly turbulent to laminar), requiring **adaptations of our dynamic load balancing schemes**
- Implicit solver for stiff component: employ our **hybrid spectral-geometric-algebraic multigrid**
 - may require specialized coarse grid solve via **H-matrix approximation**
 - adapt to **CPU-GPU nodes**



Core CS requirements for the Torch simulation

- Multiple physics, models, discretizations, and integration with V&V and UQ algos
- Algorithmic work optimality
- Performance and parallel scalability

Existing HPC frameworks focused on performance portability challenge

But none provides what is needed for the Torch simulation

- **Performance portability**
- **Rapid development**
 - Faster algorithmic and model exploration
 - Minimize device-specific code
 - Improve maintainability
 - Fewer lines of code
- **Interoperability**
 - Interface with other frameworks
 - Yet, allow efficient integration

Our proposal: Parla

Goals: Rapid development, performance portability, interoperability

- **MPI+'X' (Parla)**
 - MPI has solved the internode scalability problem
 - Per node performance and portability remains an outstanding challenge
- **Python API**
 - The most popular rapid development framework
 - Huge community / industry support
 - Extensive package support
 - 2018 Gordon Bell Prize, several GBP finalists (E.g., PyFR)
- **Not monolithic**
 - Leverages existing HPC efforts on performance portability
 - Enables interoperability/incremental adoption

MPI + Parla

Parla Python API

Data and task parallelism / nd-arrays
Memory placement/movement
Parla Kernel Generator (PKG)

PKG
Python → LLVM

Parla Runtime

Scheduler (task graph/data migration)
Performance prediction / Synthesis
Runtime interoperability

Interoperability w/
ROCm, CUDA,
Kokkos, RAJA,
OpenMP, pthreads

Parla Productivity tools

Debugger
Profiler/roofline analysis
Code migration

Parla proposed components

Programming model

(a) Parla kernels

```
@specialized
@jit(void(float64[:,:]), nopython=True, nogil=True)
def cholesky_inplace(a):
    for j in range(a.shape[0]):
        b = (a[j,:j] * a[j,:j]).sum()
        a[j,j] = sqrt(a[j,j] - b)

    for i in prange(j+1, a.shape[0]):
        a[i,j] -= (a[i,:j] * a[j,:j]).sum()
        a[i,j] /= a[j,j]

@cholesky_inplace.variant(gpu)
def cholesky_inplace(a):
    a[:] = cupy.linalg.cholesky(a)

@cholesky_inplace.variant(fpga)
def cholesky_inplace(a)
    ca = fpga(a)
    ca = fpga_cholesky(ca)
    a[:] = ca
```

(b) Parla tasks

```
def cholesky_blocked_inplace(a):
    T1 = TaskSpace("T1")
    T2 = TaskSpace("T2")
    T3 = TaskSpace("T3")
    T4 = TaskSpace("T4")
    for j in range(a.shape[0]):
        for k in range(j):
            @spawn(T1[j, k], [T4[j, k]])
            def t1():
                a[j,j] -= a[j,k] @ a[j,k].T
        @spawn(T2[j], [T1[j, 0:j]])
        def t2():
            cholesky_inplace(a[j,j])
        for i in range(j+1, a.shape[0]):
            for k in range(j):
                @spawn(T3[i, j, k], [T4[j, k], T4[i, k]])
                def t3():
                    a[i,j] -= a[i,k] @ a[j,k].T
            @spawn(T4[i, j], [T3[i, j, 0:j], T2[j]])
            def t4():
                ltriang_solve(a[j,j], a[i,j].T)

    return T2[j]
```

C++ w/ CUDA

```

/**
 * \file cuda_2d_pad_reduce.c
 * \brief Implements a CUDA implementation of
 * dense matrix-vector
 * multiply based on a 1-D row blocking.
 */

#include <assert.h>
#include <stdio.h>

#include "reals.h"
#include "cuda_2d_pad_reduce.h"

/** Row thread block size. */
static size_t RB_ = 0;

/** Column thread block size. */
static size_t CB_ = 0;

static
int
is_power_of_2 (size_t x)
{
    size_t num_ones = 0;
    while (x) {
        num_ones += x & 1;
        x >>= 1;
    }
    return (num_ones == 1) || (!num_ones);
}

void
matvec_init__cuda_2d_pad_reduce (void)
{
    gpu_init ();
    fprintf (stderr,
            " ==> Thread block size: %lu x %lu\n",
                (unsigned
long)matvec_getRB__cuda_2d_pad_reduce (),
                (unsigned
long)matvec_getCB__cuda_2d_pad_reduce ());
}

size_t
matvec_getRB__cuda_2d_pad_reduce (void)
{
    if (RB_ <= 0) {
        const char* s = getenv ("ROW_BLOCK_SIZE");
        size_t rb = 16; /* default */
        int rb_raw;
        if (s && ((rb_raw = atoi (s)) > 0)) {
            rb = (size_t)rb_raw;
            assert (is_power_of_2 (rb));
        }
    }
}

```

```

/**
 * \brief Performs  $y \leftarrow y + A * x$  on the GPU.
 *
 * \pre The dimensions (m, n) _must_ be multiples
 * of (RB, CB),
 * respectively.
 */
__global__
void
gpu_gemv__cuda_2d_pad_reduce (int M, int N,
                             size_t RB, size_t CB,
                             const real__gpu_t*
A_gpu,
                             const real__gpu_t*
x_gpu,
                             const real__gpu_t*
y_gpu,
                             real__gpu_t* Y_gpu)
{
    const int I = blockIdx.x;
    const int di = threadIdx.x;
    const int i = I * RB + di;
    const int J = blockIdx.y;
    const int dj = threadIdx.y;
    const int j = J * CB + dj;

    extern __shared__ real__gpu_t SCRATCH[]; /*
Length: (RB+1) * CB */
    real__gpu_t* XB = SCRATCH; /* Length: CB */
    real__gpu_t* YB = SCRATCH + CB; /* Length: RB *
CB */
    real__gpu_t yi0;

    /* iteration variable for within-block reduction
step */
    int ds;

    /* Load elements of x and y for this block */
    if (dj == 0) /* Col 0 in this block loads y */
        yi0 = y_gpu[i];
    else
        yi0 = 0.0;

    if (di == 0) /* Row 0 in this block loads x */
        XB[dj] = x_gpu[j];

    __syncthreads ();

    /* Local update */
    yi0 += A_gpu[i + j*M] * XB[dj];
    YB[di + dj*RB] = yi0;
    __syncthreads ();
}

```

```

real__gpu_t*
y_gpu)
{
    const int I = blockIdx.x;
    const int di = threadIdx.x;
    const int i = I * RB + di;
    real__gpu_t y0 = Y_gpu[i];
    int J;
    for (J = 1; J < NB; ++J)
        y0 += Y_gpu[i + J*M];
    y_gpu[i] = y0;
}

void
matvec__cuda_2d_pad_reduce (size_t m, size_t n,
                             const real__gpu_t* A_gpu,
                             const real__gpu_t* x_gpu,
                             real__gpu_t* y_gpu)
{
    /* Dimensions, rounded to padded size */
    size_t RB = matvec_getRB__cuda_2d_pad_reduce ();
    size_t CB = matvec_getCB__cuda_2d_pad_reduce ();
    int MB = (m + RB - 1) / RB;
    int NB = (n + CB - 1) / CB;
    int M = MB * RB;
    int N = NB * CB;

    dim3 GB (MB, NB, 1); /* thread-block grid */
    dim3 TB (RB, CB, 1); /* thread block */
    dim3 GB_reduce (MB, 1, 1);
    dim3 TB_reduce (RB, 1, 1);
    size_t SMB = (RB+1) * CB * sizeof
(real__gpu_t); /* shared memory bytes */

    real__gpu_t* Y_gpu;

    assert (SMB > 0 && SMB < 16384);

    /* Space for local results */
    Y_gpu = reals_alloc__gpu (M * NB); assert
(Y_gpu);
    gpu_gemv__cuda_2d_pad_reduce<<<GB, TB, SMB>>>
(M, N, RB, CB,
                                     A_gpu,
x_gpu, y_gpu, Y_gpu);
    gpu_reduce__cuda_2d_pad_reduce<<<GB_reduce,
TB_reduce, SMB>>> (M, RB, NB, Y_gpu, y_gpu);
    reals_free__gpu (Y_gpu);
}

/* eof */

```

```

/**
 * \file cuda_2d_pad_reduce.h
 * \brief Implements a CUDA implementation of
 * dense matrix-vector
 * multiply based on a 1-D row blocking.
 */

#ifndef INC_CUDA_2D_PAD_REDUCE_H
#define INC_CUDA_2D_PAD_REDUCE_H /*!<
cuda_2d_pad_reduce.h included. */

#include "reals_cuda.h"

#ifdef __cplusplus
extern "C" {
#endif

    /** Initialize module. */
    void matvec_init__cuda_2d_pad_reduce (void);

    /** Get row block size (for padding) */
    size_t matvec_getRB__cuda_2d_pad_reduce (void);

    /** Get column block size (for padding) */
    size_t matvec_getCB__cuda_2d_pad_reduce (void);

    /**
     * \brief CUDA-based matrix-vector multiply
     * operation,  $y \leftarrow y + A \cdot x$ .
     */
    void matvec__cuda_2d_pad_reduce (size_t m, size_t
n,
                                     const
real__gpu_t* restrict A,
                                     const
real__gpu_t* restrict x,
                                     real__gpu_t*
restrict y);

#ifdef __cplusplus
}
#endif

#endif
/* eof */

```

Parla usage plan

C++ main
MFEM or Albany
Time stepper (operator split)
Multiphysics

Fluids - MFEM or Albany

Boltzmann Parla
Radiation mpi4py

Maxwell - MFEM or Albany

Device Kernels
for Tasks

Python: Numba / PKG

C++: Kokkos or RAJA +
Cython or PyBind11

Vendor libraries
Kokkos kernels
ASICs/FPGAs

Performance-Portable Kernels

- semi-Lagrangian advection
- FFT/BLAS
- Collision / Sweeps
- Custom sparse direct solvers

Current state of Parla

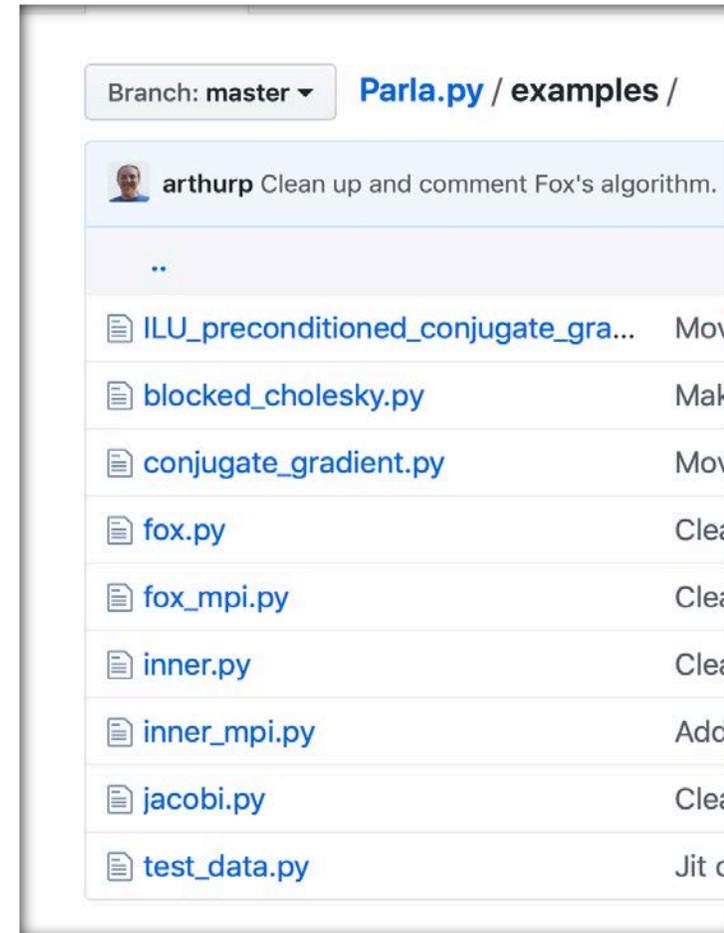
- Working prototype library that supports multicore and multi-GPU nodes
- Python implementation with online task scheduling
- Has been exercised at TACC
- Docker support
- Working examples for
 - Fox's matrix-matrix multiplication
 - Inner product
 - Recursive Block Cholesky
 - Simple Jacobi iteration
 - MPI interoperability

Developers

Ian Henriksen, Oden & CS

Arthur Peters, CS

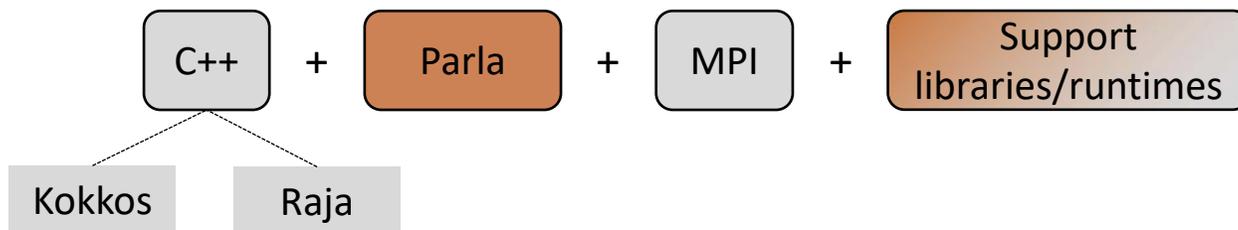
Will Ruys, Oden



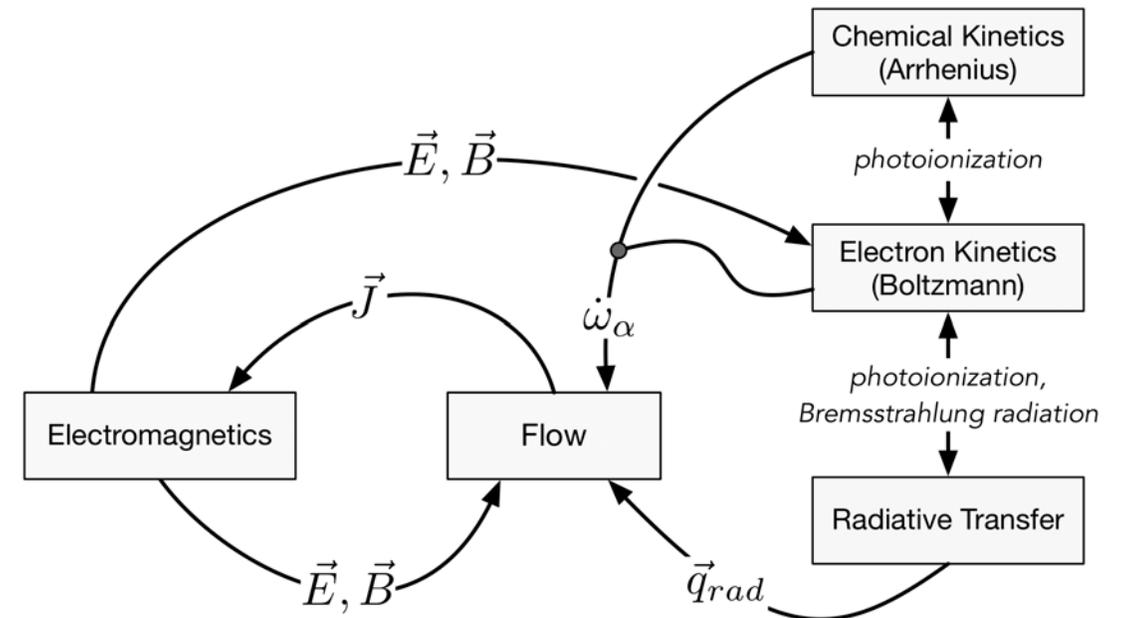
<https://github.com/ut-parla/Parla.py>

Software Development

- **Overarching Software Goal:** Develop performance-portable software for complex multi-physics modeling
- We will rely on rigorous development pipeline using modern software engineering practices to achieve full-system predictions in Year 2
- High-level application development:



Modeling Schema Interactions



Development Environment

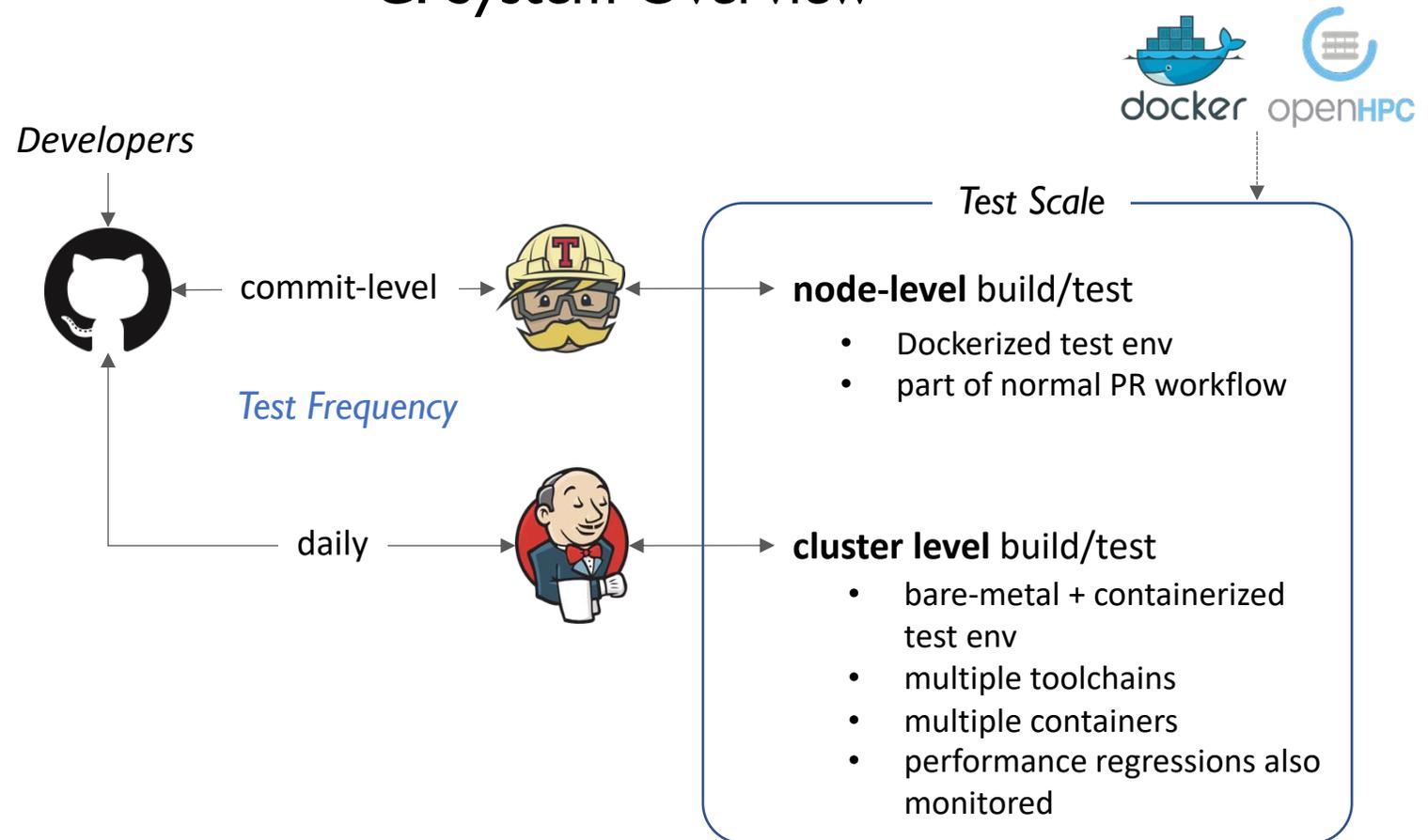
- An important evolving technology allowing for reproducibility, portability, and replication of underlying software environments is application **containerization**
 - allows us to mimic supercomputing OS envs and stacks for local development
 - improves application portability
- Plan to maintain project-specific Docker images that include all underlying software libs and tools required for ICP torch application
- Also utilizing HPC-centric containers for execution on production resources
 - Singularity
 - Charliecloud
- Companion effort to periodically benchmark containerized app version against bare-metal



Development Environment (cont.)

- Project Management: **GitHub**
 - milestone/issue tracking
 - release management
 - version control (git)
 - wiki
- Continuous Integration (CI):
 - **Travis** (community instance)
 - compilation
 - unit testing
 - smaller regression tests
 - **Jenkins** (local instance, 1K-2K cores)
 - OpenHPC based
 - vendor compilers and math libraries
 - include larger regression tests
- Code Coverage Tools:
 - gcc/lcov + coverage.py
- Source Code Documentation:
 - Doxygen (C++)
 - Sphinx (Python)

CI System Overview



Additional Supporting Software

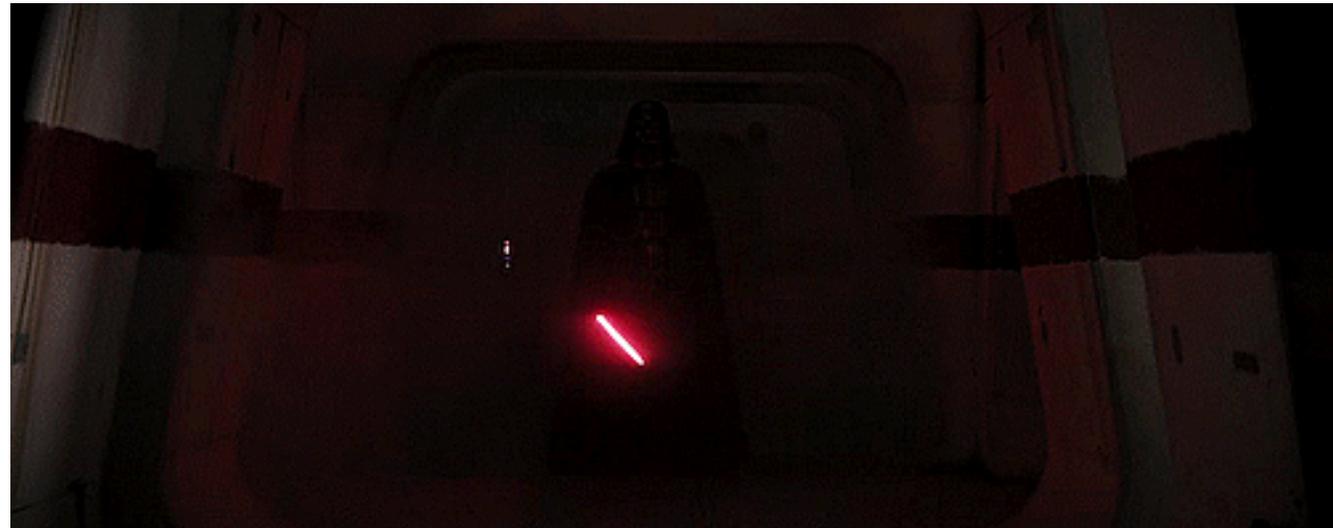
DoE driven efforts

Local UT efforts

- **Albany** and **MFEM** – FE discretization and numerics
- **BOLSIG+** and **Antioch** – chemical kinetics and transport
- **Trilinos** and **PETSc** – Krylov solvers, AMG
- **MASA** - manufactured solution verification library
- **Dakota** – optimization, sensitivity, and UQ analysis
- **QUESO** and **hippylib** – Bayesian UQ support
- **libGRVY** – HPC utility library
- **PAPI, TAU** - detailed performance profiling
- **HDF5** – I/O library

Summary

- Plasma torch physics
 - Exascale numerical algorithms
 - Core CS: Parla, compiler, and runtime, software productivity tools
 - Cutting-edge software engineering
-
- The real torch!



Senior Investigators

- **Robert Moser** (PI): Modeling, validation and uncertainty quantification
 - **George Biros** (Co-PI): Exascale algorithms and software
- Noel Clemens: Experiments
 - Mattan Erez: Exascale runtime systems
 - Omar Ghattas: Exascale and UQ algorithms
 - Milos Gligoric: Exascale tools
 - David Hatch: Plasma physics
 - John McCalpin: Performance optimization
 - Tommy Minyard: HPC testbeds
 - Tinsley Oden: Uncertainty quantification
 - Todd Oliver: Modeling, validation, uncertainty quantification
 - Keshav Pingali: Exascale tools and the Parla (parallel) language
 - Laxminarayan Raja: Plasma physics and modeling
 - Christopher Rossbach: Exascale runtime systems
 - Karl W. Schulz: Modeling, exascale, performance analysis, software engineering
 - Philip Varghese: Kinetics modeling
 - Karen Willcox: Uncertainty quantification and UQ algorithms
 - Martin Burtscher (Texas State University): Heterogenous architectures

