MDSplusML - Optimizations for Data Access to Facilitate Machine Learning Pipelines

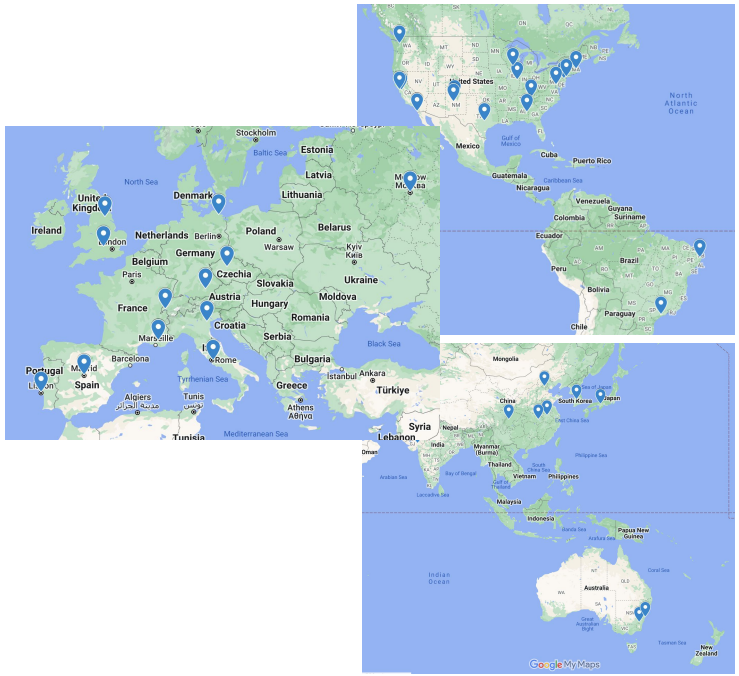
Joshua Stillerman[1], Stephen Lane-Walsh[1], Mark Winkel[1], Cristina Rea[1], Gregorio Luigi Trevisan[1], Aleksandar Jelenak[2], John Readey[2]

[1]MIT Plasma Science and Fusion Center

[2]HDF Group

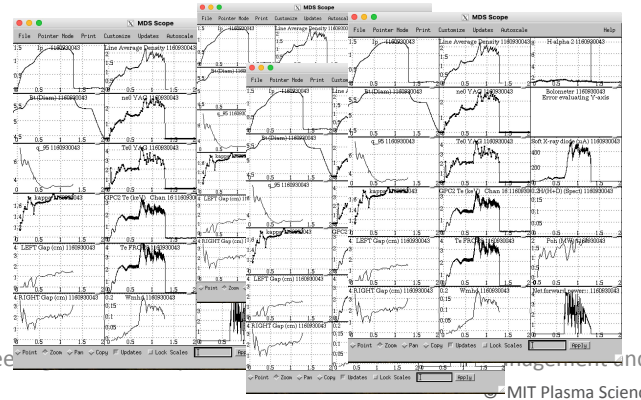
Introduction

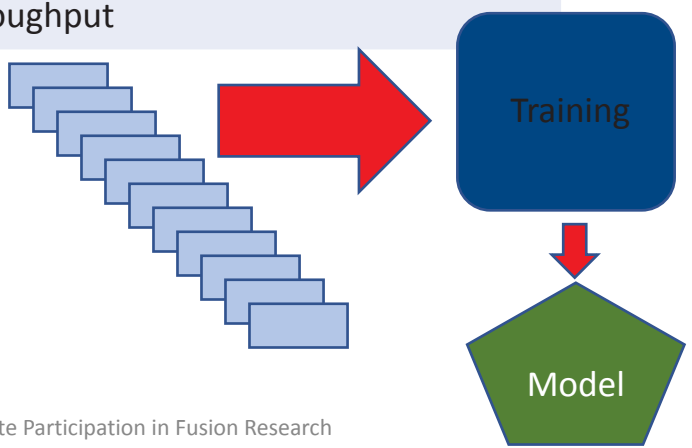
- **Introduction to MDSplus**
  - MDSplus is a widely used data management system in magnetic fusion energy research.
  - Provides data storage, management, and remote access through a vector-based interpreter API.
- **Current Optimization Focus**
  - Developed for rapid single-shot analyses.
  - Long term data storage and comprehension.
- **Need optimization for machine learning (ML) applications.**



Requirements

Shot by Shot Data Analysis	Machine Learning
Single Shot	large number of shots
Single Experiment	Multiple experiments
FAIR - less crucial	FAIR - required
Many signals	Selected signals
On prem	On-prem, Cloud, Off-site
Speed	Throughput





Design inputs

- Users want to do their own data preparation
- Users want fast access to data for data preparation
- Users want parallel threads to not interfere
- Significant application specific data processing
- Users want to produce their own ‘input artifacts’
- Atypical user cohort

Provide scalable fast access to measured quantities

FAIR

- **F**AIR Findable – cross machine data dictionary IMAS+ (Cummings)
- F**A**IR Accessible – Unified API
- FA**I**R Interoperable – IMAS+ dictionary (Cummings)
- FAI**R** Reusable - Well described data allows diverse community of consumers

Approach

- Cache popular quantities from all (or most) shots
  - Evaluated data expressions
  - Cache implementation can vary between platforms
  - Use data warehouse
  - Fall back on local native data access
  - Update cache list nightly
  - Update cache contents nightly
- Start with inputs to IMAS described quantities
- Parallelization
  - Need wide / complete data request up front
  - MDSplus
    - Thick/distributed (low level IO across network)
    - Thin (high level IO across network)
    - getMany group together data requests within a shot
    - getManyMany (group together ‘getMany’ across shots

Testing Caches

- Working with HDF group
- Using ‘disruption-py’ data requests as example (GL Trevisan et al., APS-DPP 2024)
  - ~60 interesting quantities per shot
  - Caching inputs to calculations
  - (real application we could cache the returned answers)
- MDSplus distributed client particularly bad:
  - Every IO over the network
  - Long search path

    alcdata-test::/cmod/trees/test/~t/;
    alcdata-new::/cmod/trees/new/~t/;
    alcdata-models::/cmod/trees/models/~t/;
    alcdata-archives::/cmod/trees/archives/~i~h/~g~f/~e~d/~t;
    alcdata-saved::/cmod/trees/saved/~t/

Testing Caches (2)

- MDSplus Thin Client much better
  - path still long – Shorten path for server process ✅
  - Cache miss triggers tree open – Give up on cache miss ✅
- MDSplus getMany would be even better
- HSDS – Distributed implementation of HDF5 API
  - Scalable network architecture
  - Investigated for cloud deployments
  - Parallel data requests
  - Cache miss issue also
  - Individual record requests too small to win
  - getMany equivalent will likely improve this ✅
- MongoDB
  - In memory database
  - Very fast for local access
  - Network transaction cost
  - getMany equivalent helps a lot ✅

Testing Caches (3)

- HDF5 Files
  - Local SSD very fast
  - No network protocol (except NFS)
  - Could be very good solution:
    - Cache records by shot
    - Retrieve shot caches over network to local SSD
  - the same solution will work both on-prem and in the cloud

Factors

- Bandwidth is cheap
- Latency is expensive or even unfixable
- Transaction costs overwhelm bandwidth for our needs
  - Even with local sub-millisecond delays
- Cloud based storage
  - Tends to be high throughput
  - Tends to be low performance (High transaction cost)
  - Parallel request needed to achieve performance

The plan

- Implement getManyMany api
  - Request list of quantities
  - From list of shots
  - Server deals with parallelization
  - Very small number of network transactions
  - Users do not need to deal with asynchronous IO, multi threading, ….
- Users consume data from memory
- Initial implementation can be created using existing MDSplus APIs
- Server is then free to use any caching / parallelism
  - Data warehouse of pre computed quantities
  - …
- XArray MDSplus API - regardless of caching

Questions?

This work is supported also by the U.S. Department of Energy,
Office of Science, Office of Fusion Energy Sciences,
under Award DE-SC0024368