

Argonne MCS Seminar – JUNE 20, 2023

Mochi in practice:

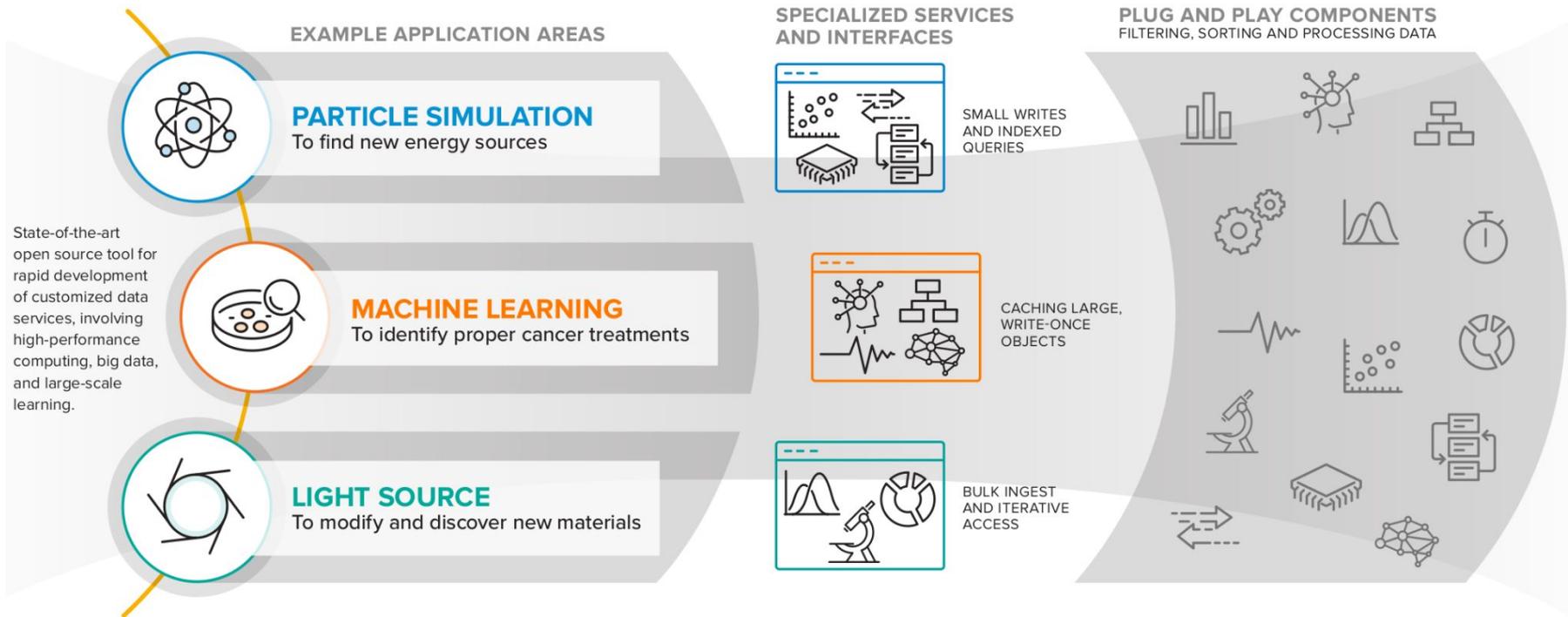
Developing data services for high-energy physics and elastic in situ visualization workflows

MATTHIEU DORIER, Software Development Specialist in the RADIX-IO team



Argonne National Laboratory is a
U.S. Department of Energy laboratory
managed by UChicago Argonne, LLC.

In last week's episode: the Mochi concept



The Mochi methodology

How to rapidly develop data services

The HEPnOS storage service

A custom service for high energy physics workflows

Autotuning a Mochi service

Using the DeepHyper AutoML framework to tune HEPnOS

Towards elastic data services

The Colza elastic in situ visualization framework

The Mochi methodology

Rapidly developing custom data services

Mochi: Composing Data Services for High-Performance Computing Environments, *Robert B Ross, George Amvrosiadis, Philip Carns, Charles D Cranor, Matthieu Dorier, Kevin Harms, Greg Ganger, Garth Gibson, Samuel K Gutierrez, Robert Latham, Bob Robey, Dana Robinson, Bradley Settlemyer, Galen Shipman, Shane Snyder, Jerome Soumagne, Qing Zheng*, Journal of Computer Science and Technology (Springer JCST)

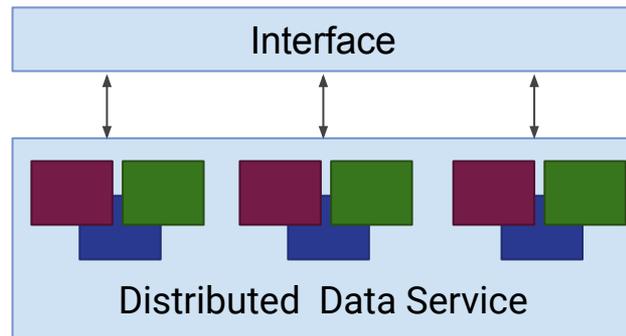
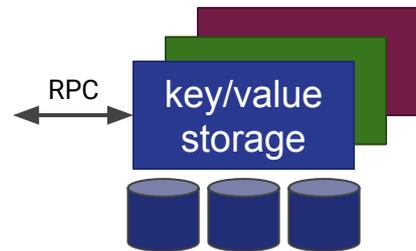
Services and Components

What is a Mochi component?

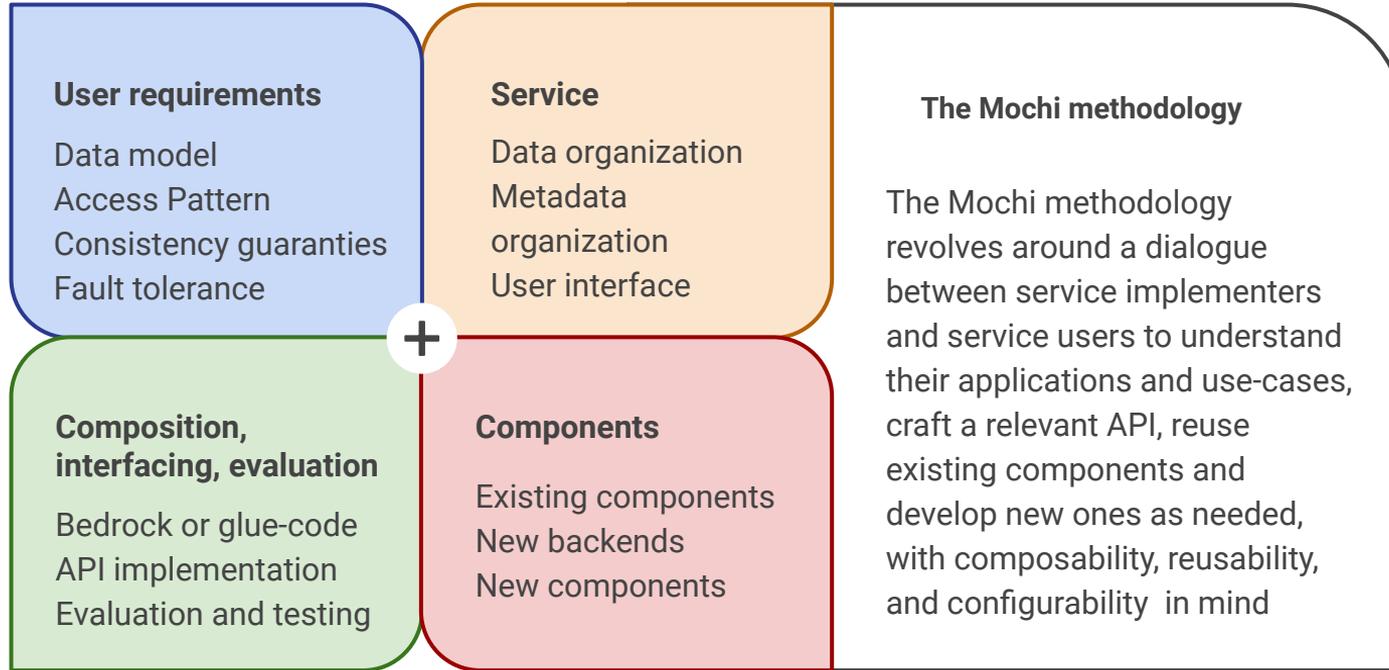
- Provides a single functionality (e.g., key/value storage)
- Accessible via RPC/RDMA
- Shares process resources with other components
- Multiple backend implementations for the functionality

What is a Mochi service?

- Specific composition of Mochi components
- Specific (usually application-tailed) interface on top
- Specific data semantics and access requirements



Designing a Mochi service



Components: don't reinvent the wheel!

mochi-yokan	Key/value and document storage
mochi-bake	Blob storage
mochi-poesie	Embedded scripting
mochi-abt-io	Wrappers for POSIX I/O
mochi-remi	File migration
mochi-ssg	Gossip-based failure detection / group membership
mochi-raft	Replicated state machine
mochi-bedrock	Bootstrapping and configuration

Sharing components in the community = everyone benefits from contributions

Oh, you want to implement a new component?

mochi-hpc / margo-microservice-template Public template

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

main 2 branches 0 tags

Go to file Add file Code Use this template

Your main branch isn't protected
Protect this branch from force pushing or deletion, or require status checks before merging. [Learn more](#) Protect this branch

Author	Commit Message	Commit Hash	Time	Commits
mdorier	Merge branch 'main' of github.com:mochi-hpc/margo-microservice-templ...	df8ec8c	last month	62
	.github	updated script that sets up project	2 months ago	
	examples	moved json files around	2 months ago	
	include/alpha	added margo_instance_id to resource create and open functions	last month	
	src	added margo_instance_id to resource create and open functions	last month	
	tests	moved json files around	2 months ago	
	.gitignore	Initial commit	3 years ago	
	CMakeLists.txt	Force Catch2 3.0.1 or greater in CMakeLists.txt	last month	
	COPYRIGHT	added copyright file	3 years ago	
	README.md	Update README.md	last month	

Starting project with the annoying code already filled in so you can focus on what matters:

- The API
- The features

The templates also provide

- Unit tests (catch2)
- Github actions for automated testing and code coverage (codecov.io)

Rely on spack for dependencies (spack.yaml) and on cmake for building the code

Composition with Bedrock: example configuration

```
{
  "margo" : {
    "mercury" : { },
    "argobots" : {
      "abt_mem_max_num_stacks" : 8,
      "abt_thread_stacksize" : 2097152,
      "pools" : [
        {
          "name" : "my_rpc_pool",
          "kind" : "fifo_wait",
          "access" : "mpmc"
        }
      ],
      "xstreams" : [
        {
          "name" : "my_rpc_xstream",
          "cpubind" : 2,
          "affinity" : [ 2, 3, 4, 5 ],
          "scheduler" : {
            "type" : "basic_wait",
            "pools" : [ "my_rpc_pool" ]
          }
        }
      ]
    },
    "progress_pool" : "__primary__",
    "rpc_pool" : "my_rpc_pool"
  },
}
```

```
"abt_io" : [
  {
    "name" : "my_abt_io",
    "pool" : "__primary__"
  }
],
"ssg" : [
  {
    "name" : "mygroup",
    "bootstrap" : "init",
    "group_file" : "mygroup.ssg"
  }
],
"libraries" : {
  "module_a" : "examples/libexample-module-a.so",
  "module_b" : "examples/libexample-module-b.so"
},
"clients" : [
  {
    "name" : "ClientA",
    "type" : "module_a",
    "config" : {},
    "dependencies" : {}
  }
]
],
```

```
"providers" : [
  {
    "name" : "ProviderA",
    "type" : "module_a",
    "provider_id" : 42,
    "pool" : "__primary__",
    "config" : {},
    "dependencies" : {}
  },
  {
    "name" : "ProviderB",
    "type" : "module_b",
    "provider_id" : 33,
    "pool" : "__primary__",
    "config" : {},
    "dependencies" : {
      "ssg_group" : "mygroup",
      "a_provider" : "ProviderA",
      "a_local" : [ "ProviderA@local" ],
      "a_client" : "module_a:client"
    }
  }
]
}
```

Advantages of Bedrock

Composition

- Describe it in JSON (or programmatically in Python or Jx9)
- Decouples threading/scheduling aspects from functionalities aspects

Interface

- Query the configuration any time via RPC
- Change the configuration (add/remove components, wire them differently)
- Critical when exploring the parameter space of a service

Ok, do you have a concrete example of applying this methodology?



Argonne National Laboratory is a
U.S. Department of Energy laboratory
managed by UChicago Argonne, LLC.

The HEPnOS storage service

A custom service for high energy physics workflows

HEPnOS: a Specialized Data Service for High Energy Physics Analysis, *Sajid Ali, Steven Calvez, Philip Carns, Matthieu Dorier, Pengfei Ding, James Kowalkowski, Robert Latham, Andrew Norman, Marc Paterno, Robert Ross, Saba Sehrish, Shane Snyder, Jerome Soumagne*. ESSA Workshop, May 14th, 2023, St. Petersburg, Florida, USA



Argonne National Laboratory is a
U.S. Department of Energy laboratory
managed by UChicago Argonne, LLC.

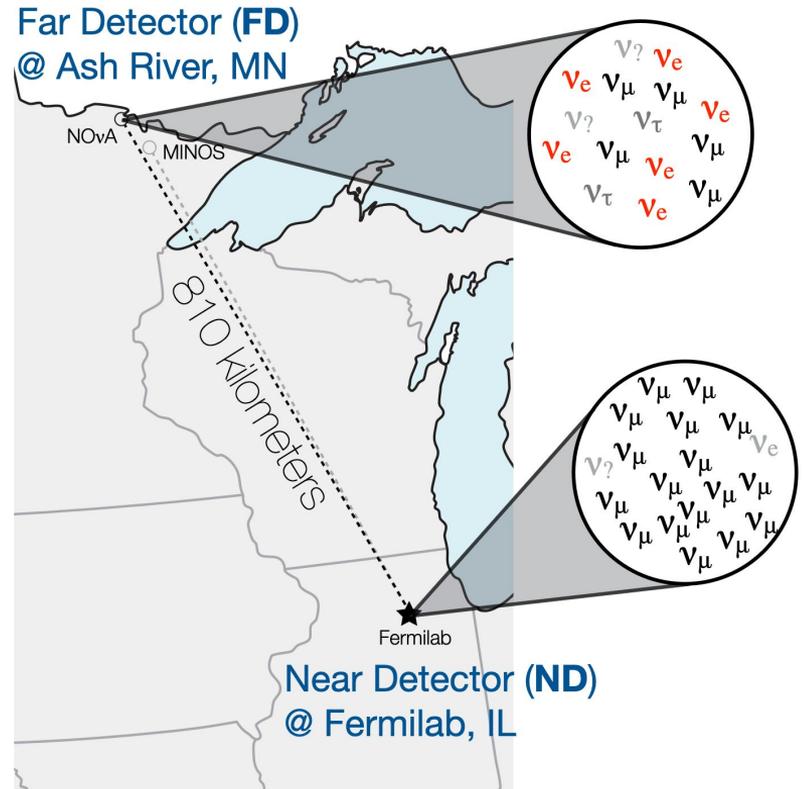


The NOvA experiment

Fermilab's accelerator complex produces the most intense (muon) neutrino beam in the world and sends it through the earth to northern Minnesota.

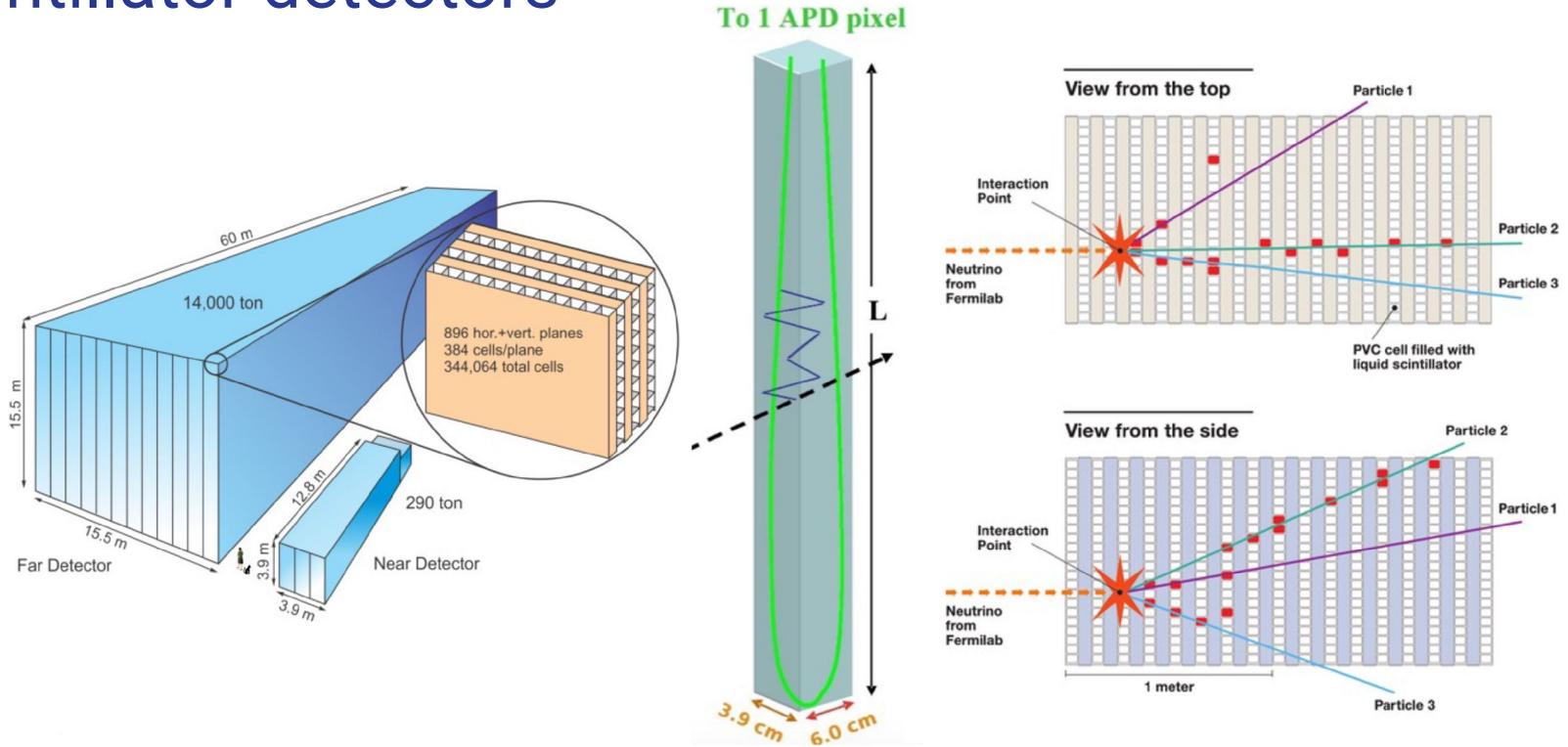
Moving at close to the speed of light, the neutrinos make the 800-km journey in less than three milliseconds.

When a neutrino interacts in the NOvA detector in Minnesota, it creates distinctive particle tracks.



Credit: Maria Manrique Plata, NOvA in 10 minutes, New Perspectives 2022

Scintillator detectors



Credit: Maria Manrique Plata, NOvA in 10 minutes, New Perspectives 2022

Physics task at hand

- Classify types of interactions based on patterns found in the detector:
 - Is it a muon or electron neutrino?
 - Is it a charged current or a neutral current interaction?
- Classify a detector event by comparing its cell energy pattern to a library of 77M simulated events cell energy patterns, choosing 10K that are “most similar”.
- Compare the pattern of energy (hit) deposited in the cells of one event with the pattern in another event.
- Note: the “most similar” metric is motivated by an electrostatic analogy: energy comparison for two systems of point charges laid on top of each other.

Goals: Harness HPC resources

Present day analysis maps the work onto computer cores **by assigning each core one file** (which contains many events).

This **limits the maximum number of cores** that can be used for analyzing a dataset.

The goal is to **remove this bottleneck** and allow for faster processing of datasets by **harnessing HPC resources**.

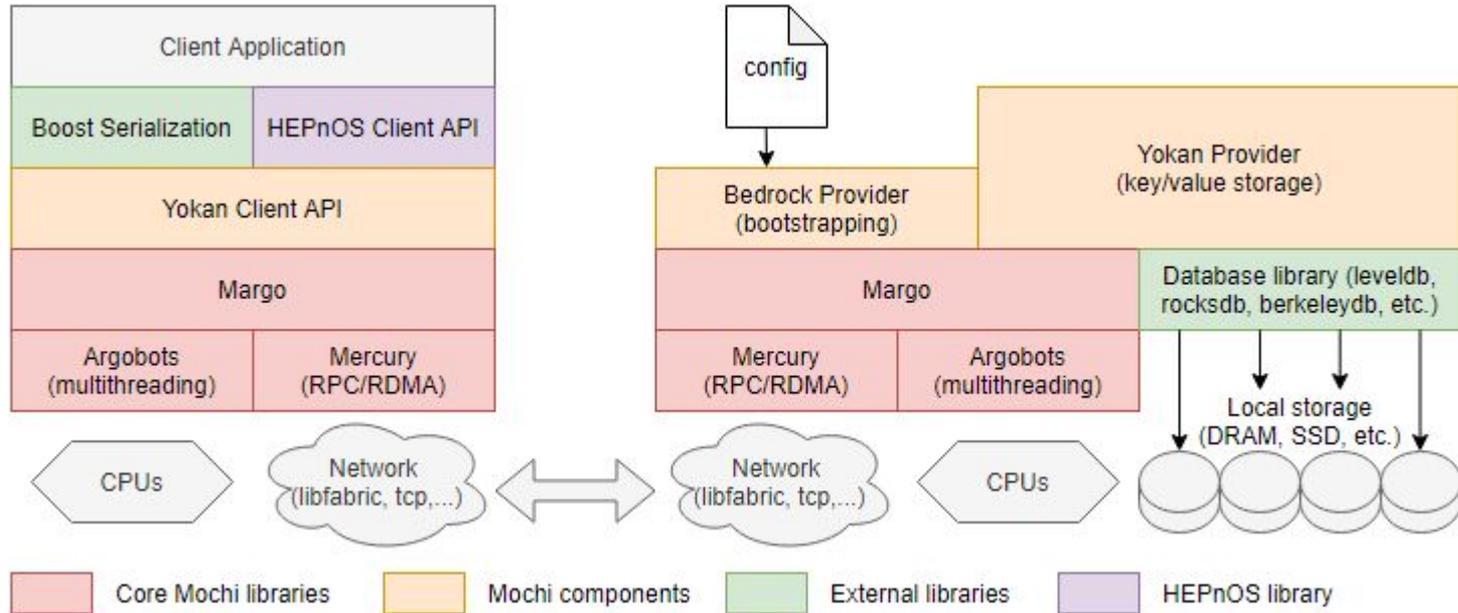
HPC clusters have nodes that are connected by **low latency, high bandwidth interconnects**.

Enters HEPnOS



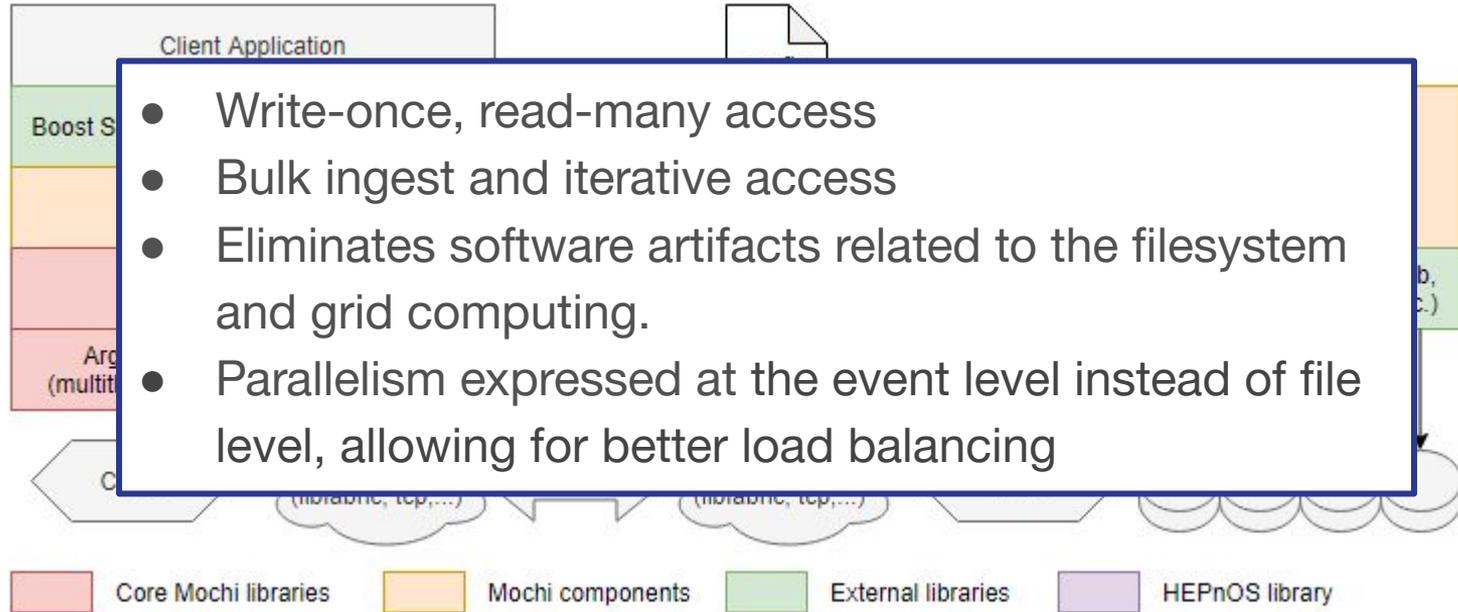
Argonne National Laboratory is a
U.S. Department of Energy laboratory
managed by UChicago Argonne, LLC.

High-Energy Physics' new Object Store: Architecture



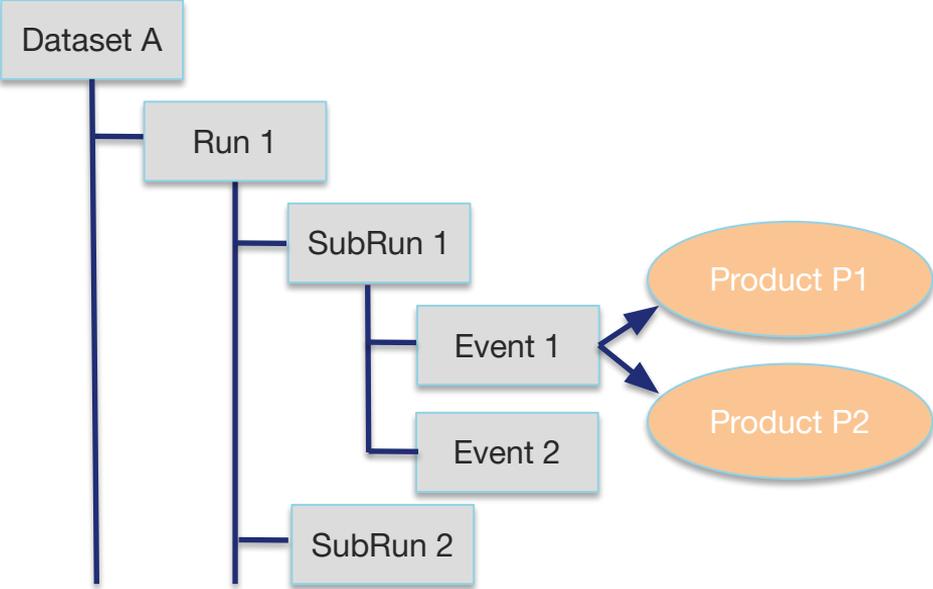
Architecture of HEPnOS: (Left) Client stack, (Right) Server stack

High-Energy Physics' new Object Store: Architecture



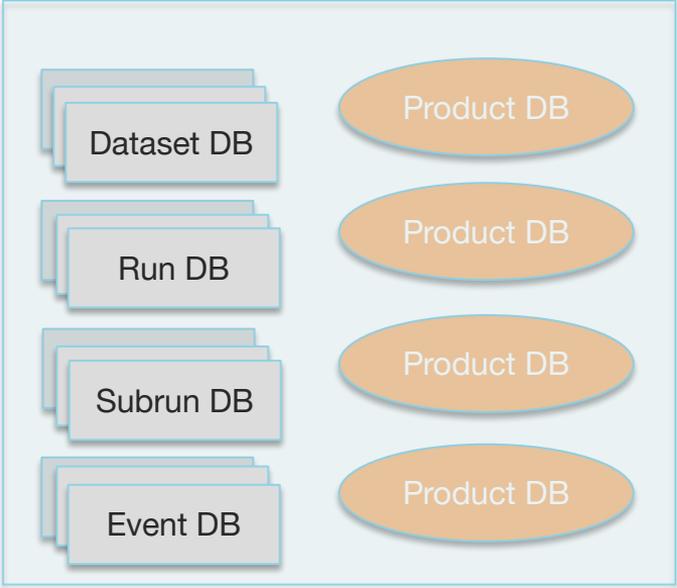
Architecture of HEPnOS: (Left) Client stack, (Right) Server stack

Data organization



 Stored with lexicographic ordering

 C++ object, stored without ordering



Yokan providers in a HEPnOS server instance

Example of HEPnOS's interface

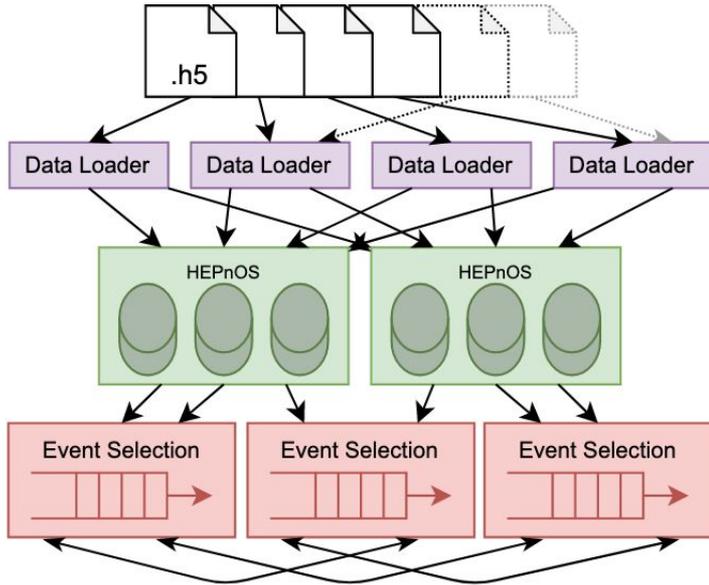
```
// initialize a handle to the HEPnOS datastore
auto datastore = hepnos::DataStore::connect("connection.json");
// access a nested dataset
hepnos::DataSet ds = datastore["path/to/dataset"];
hepnos::Run run = ds[43]; // access run 43 in the dataset
hepnos::SubRun subrun = run[56]; // access subrun 56
hepnos::Event ev = subrun[25]; // access event 25
// iterate over the subruns in a run
// using a C++ range-based for
for(auto& subrun : run) { ... }
```

Example of HEPnOS's interface

```
struct Hit {
    float energy; // member variables
    ...
    // serialization function for boost to use
    template<typename A>
    void serialize(A& a, unsigned long version) {
        ar & energy;
        ...
    }
};

...
hepnos::Event ev = subrun[25]; // access event 25
// store data (an std::vector of Hits)
st::vector<Hit> vh1 = ...;
ev.store("mylabel", vh1);
// load data
std::vector<Hit> vh2;
sv.load("mylabel", vh2);
```

New workflow with HEPnOS



Set aside some of the compute nodes to run the HEPnOS Server.

Load the data into the HEPnOS server.

Call the processing function on “events” on the client nodes, with the HEPnOS Parallel Event Processor.

Re-run the analysis as needed, without needing to reload data into the server!

Parallel Event Processor

Task distribution

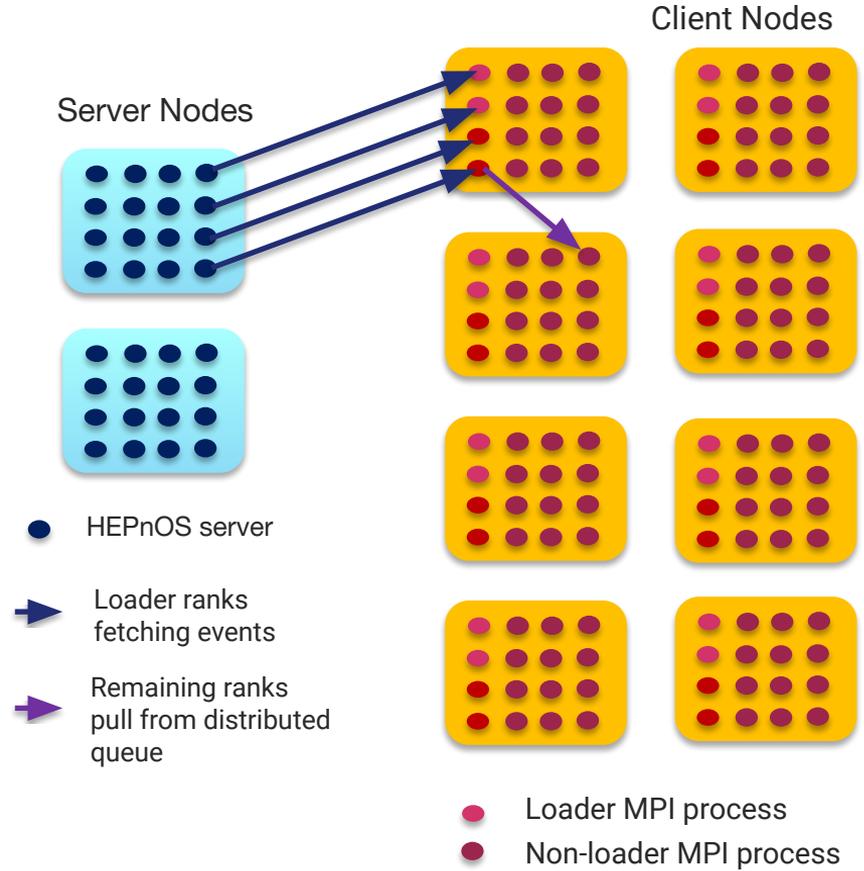
Subset of the MPI ranks are designated as “loader” ranks

Loader ranks fetch the “events” from the datastore (in batches) and collectively provide a distributed queue

All cores fetch events (in batches) from the queue

Implicit load-balancing at the event level

Desired products are pre-loaded by all the ranks, in batch, in the background



All the caller has to provide is a processing function to invoke on all the events!

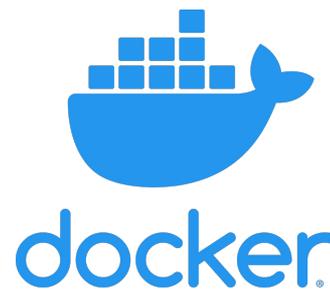
Experimental Setup: Dataset and Platform

Dataset of 1929 files containing 4,359,414 events and 17,878,347 slices; size: ~0.2TB, representing ~1.1% of the total data (duplicated 4x for scaling studies).

File-based workflow with the Python multiprocessing module used to map files to cores, with cores being idle at larger node counts.

HEPnOS-based workflow using two storage backends via the Yokan provider:

- In-memory backend (using the C++ `std::map`)
- Node-local SSD backend via the RocksDB library

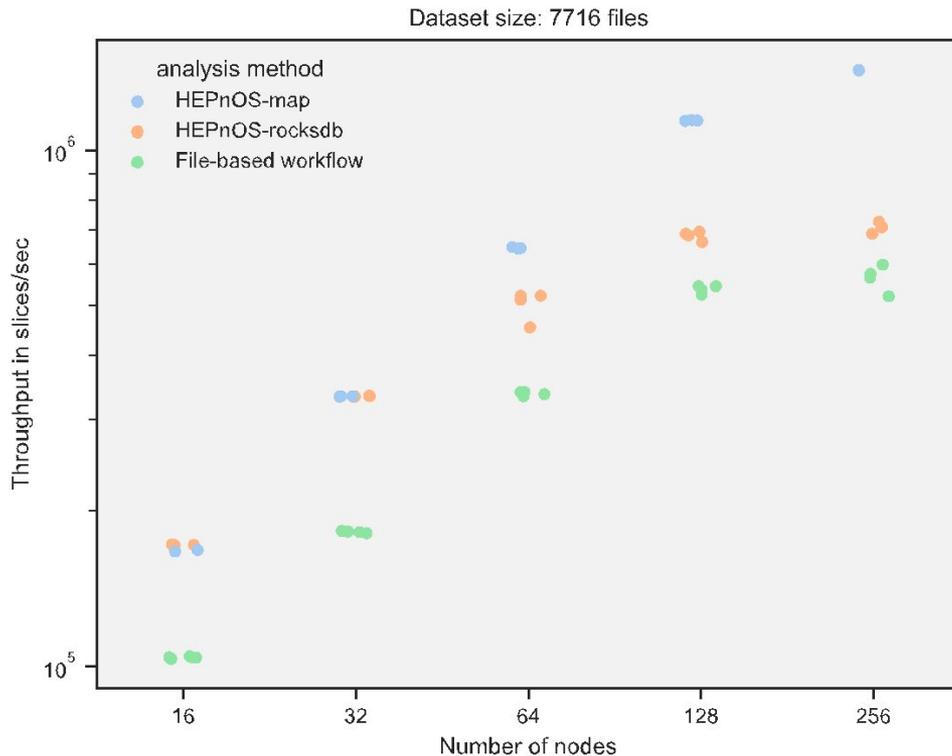


Throughput as a function of # nodes

Performance of HEPnOS with either backend is better than the file-based workflow.

In-memory backend of HEPnOS achieves ~85% scaling efficiency at 128 nodes.

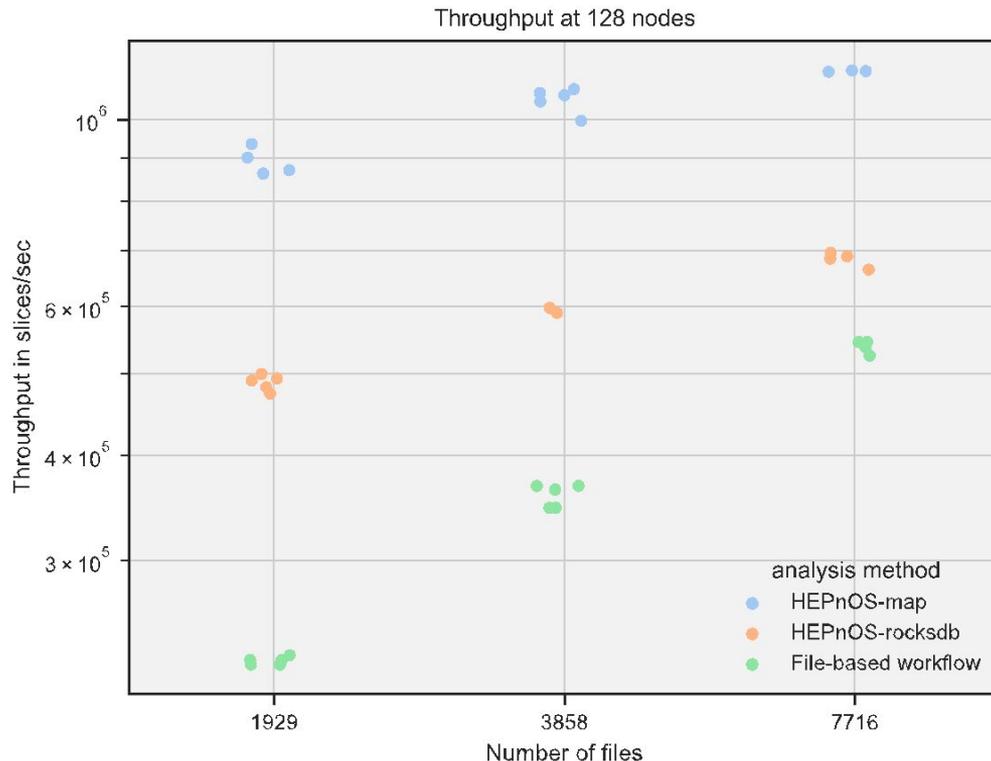
Typical data sizes for this workflow in production would allow for the usage of the in-memory backend.



Throughput as a function of # files

File-based workflow is unable to harness all the available cores with 1929 files where only ~24% of cores are used.

By using the HEPnOS-based workflow, we are better able to utilize compute resources.



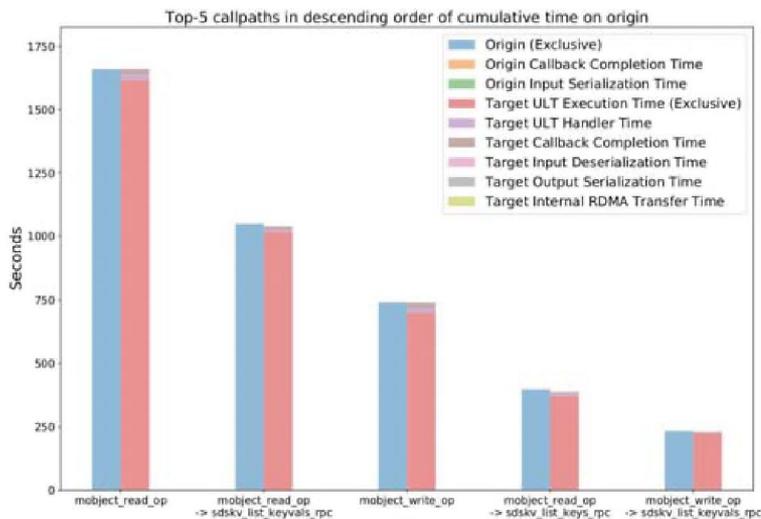
How did you tune this thing?



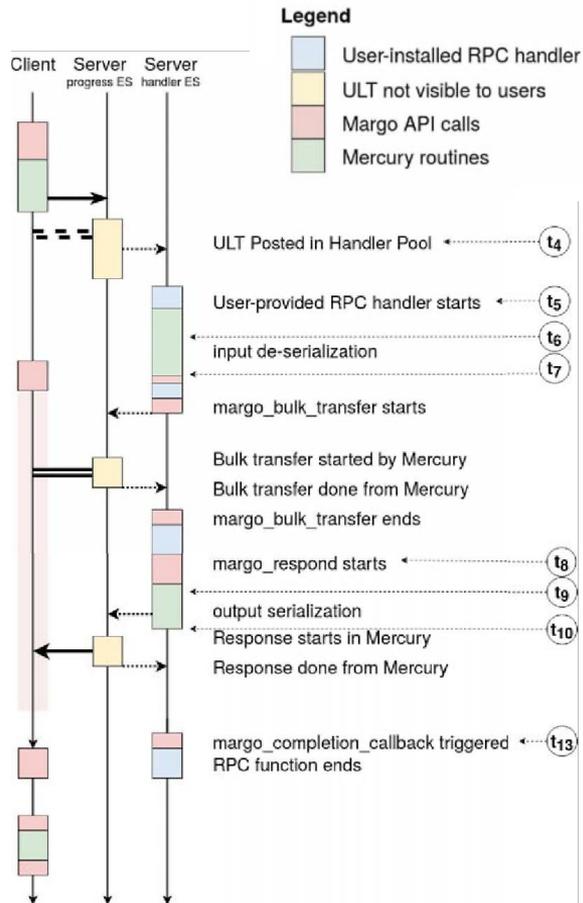
Argonne National Laboratory is a
U.S. Department of Energy laboratory
managed by UChicago Argonne, LLC.

Manual tuning efforts (it's hard!)

Callpath ancestry appended to RPCs allows tracking and ranking distributed callpaths (e.g., by time in the callpath)



Performance variables exported by Mercury in conjunction with ULT data allow detailed analysis of timing.



Autotuning a Mochi service

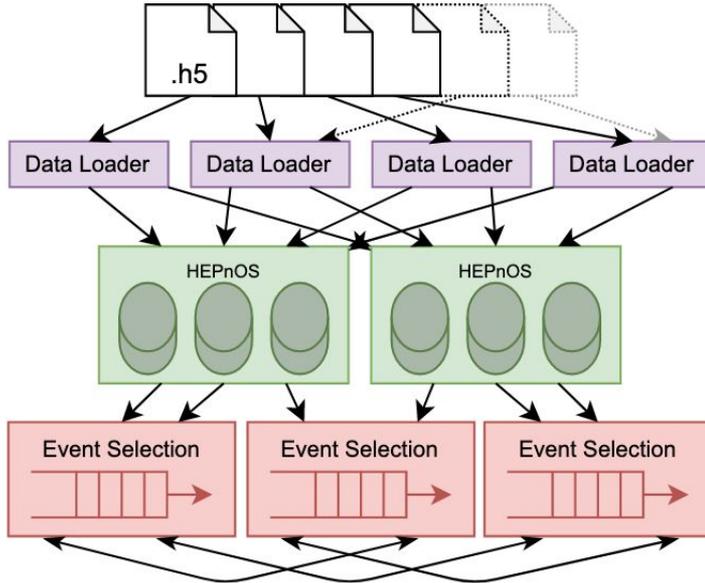
Using the DeepHyper framework to tune HEPnOS

MATTHIEU DORIER, ROMAIN EGELE, PRASANNA BALAPRAKASH, JAEHOON KOO,
SANDEEP MADIREDDY, SRINIVASAN RAMESH, ALLEN D. MALONY, and ROB ROSS



Argonne National Laboratory is a
U.S. Department of Energy laboratory
managed by UChicago Argonne, LLC.

Event Selection Workflow Parameter Space



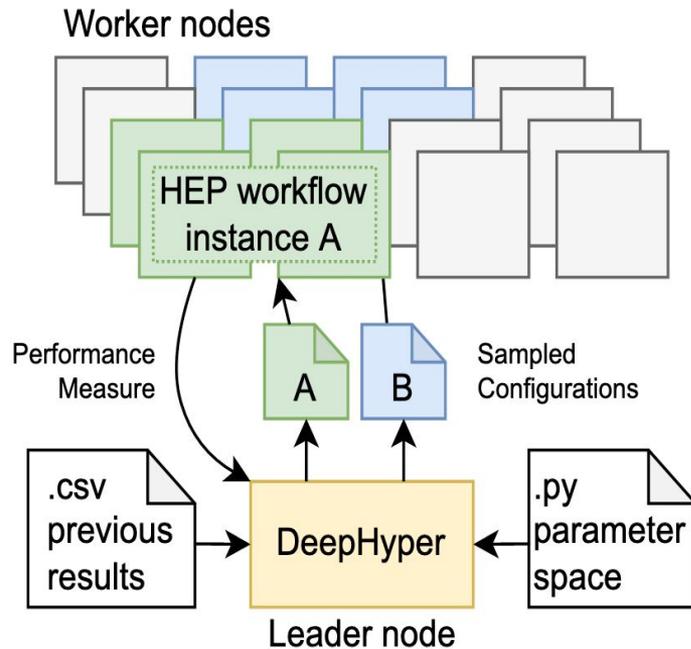
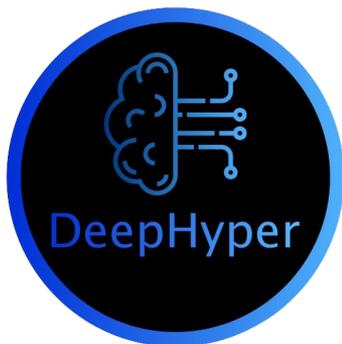
Parameter space (with values/ranges and distributions)

ProgressThread	(True/False, Uniform)	- Whether to use a dedicated network progress thread in Dataloader processes
WriteBatchSize	([1, 2048], Log-uniform)	- Size of the batches (in number of events used when sending data to HEPnOS)
PESperNode	({1, 2, 4, 8, 16, 32}, Uniform)	- Number of Dataloader processes per physical node
LoaderAsync	(True/False, Uniform)	- Use threads to asynchronously send batches to HEPnOS
LoaderAsyncThreads	([1, 63], Log-uniform)	- Number of threads for asynchronous store in Dataloader
ProgressThread	(True/False, Uniform)	- Whether to use a dedicated network progress thread in HEPnOS servers
NumRPCthreads	([0, 63], Uniform)	- Number of threads used by HEPnOS servers to service RPC
NumEventDBs	([1, 16], Uniform)	- Number of database instances per HEPnOS server for Events
NumProductDBs	([1, 16], Uniform)	- Number of database instances per HEPnOS server for Products
NumProviders	([1, 32], Uniform)	- Number of database providers per HEPnOS server
ThreadPoolType*	(fifo, fifo_wait, prio_wait), Uniform)	- Argobots thread pool type each provider uses
PESperNode*	({1, 2, 4, 8, 16, 32}, Uniform)	- Number of HEPnOS server processes per physical node
ProgressThread	(True/False, Uniform)	- Whether to use a dedicated network progress thread in PEP processes
NumThreads	([1, 31], Uniform)	- Uniform & Number of threads use to process data in parallel
InputBatchSize	([8, 1024], Log-uniform)	- Batch size (in number of events) to use when loading data from HEPnOS
OutputBatchSize	([8, 1024], Log-uniform)	- Batch size (in number of events) to use when sending data across PEP processes
PESperNode	({1, 2, 4, 8, 16, 32}, Uniform)	- Number of PEP processes per physical node
UsePreloading*	(True/False, Uniform)	- Use batch-prefetching of data products instead of per-product load
UseRDMA*	(True/False, Uniform)	- Use RDMA to transfer data
BusySpin	(True/False, Uniform)	- Network polling strategy (common to all three components)

Let's automatize: black-box tuning with DeepHyper

Parallel Asynchronous Bayesian Optimization

- Many instances evaluated **in parallel**
- **Asynchronous** updates



<https://deephyper.readthedocs.io>

But ML-based autotuning is not new...

Contribution: use transfer learning to leverage past autotuning!

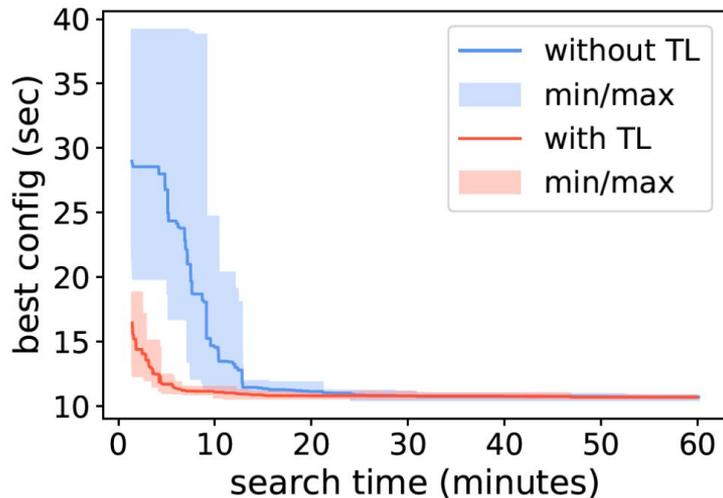


Argonne National Laboratory is a
U.S. Department of Energy laboratory
managed by UChicago Argonne, LLC.

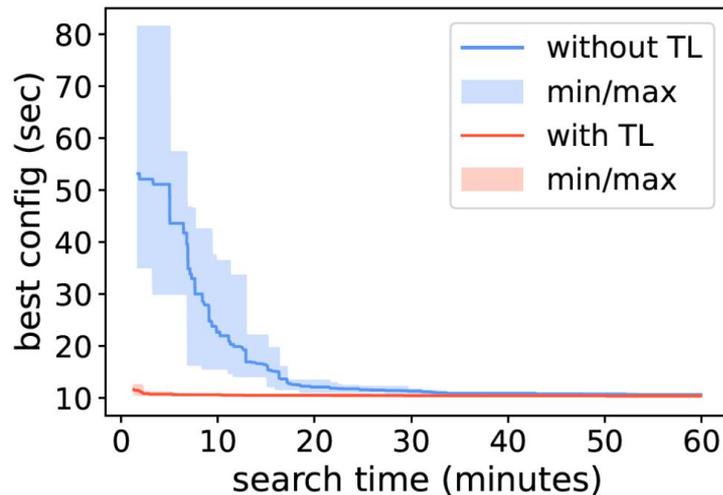
Five experimental setups

1. Initial: only the first step of the workflow, on 4 nodes per instance
 - 11 parameters
2. Full workflow: **2-steps workflow** on 4 nodes per instance
 - 16 parameters, w/ and w/o transfer-learning from setup 1
3. More parameters: 2-steps workflow on 4 nodes with **more parameters**
 - 20 parameters, w/ and w/o transfer-learning from setup 2
4. Full workflow with **8 nodes per instance**
 - 20 parameters, w/ and w/o transfer-learning from setup 3
5. Full workflow with **16 nodes per instance**
 - 20 parameters, w/ and w/o transfer-learning from setup 4

Highlight: transfer-learning to larger search space



From 1-step to 2-step workflow
(11 to 16 parameters) on 4 nodes per instance

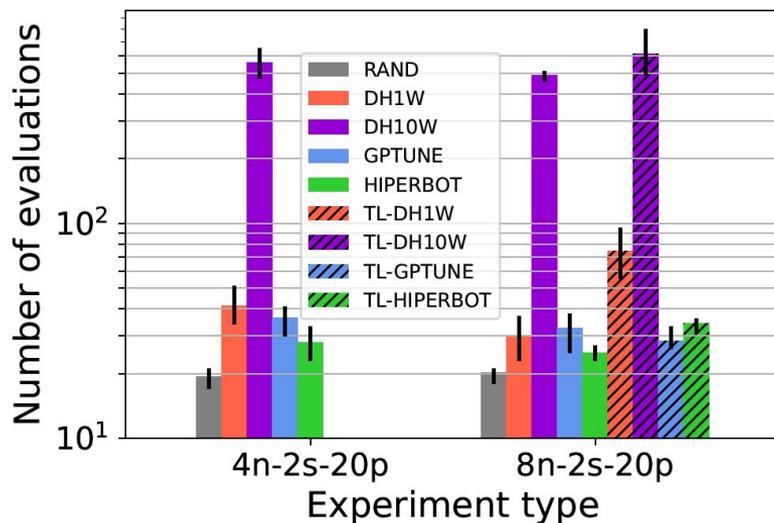


2-step workflow on 4 nodes per instance
From 16 to 20 parameters

The competition

- **GP Tune** (<https://github.com/gptune/GPTune>)
 - [Liu et al, 2021] *GP Tune: multitask learning for autotuning exascale applications* (PPoPP 2021)
 - Uses a Gaussian Process surrogate model ($O(n^3)$ complexity)
 - Only parallelizes the initial random search
- **HiPerBOT** (code provided privately by its authors)
 - [Menon et al, 2020] *Auto-tuning parameter choices in HPC applications using Bayesian optimization* (IPDPS 2020)
 - No parallelization
- We also implemented **Gaussian Process** in DeepHyper for comparison with the default **Random Forest**

Highlight result: DeepHyper outperforms its competitors



DeepHyper outperforms GPTune and HiPerBOt because of its use of parallelism and its asynchronous model updates

Transfer learning is beneficial to both DeepHyper and GPTune, but (strangely) damaging to HiPerBOt's results

With parallelism enabled, DeepHyper allows doing many more evaluations than its competitors

Bringing elasticity to Mochi services

Colza: an elastic in situ visualization framework

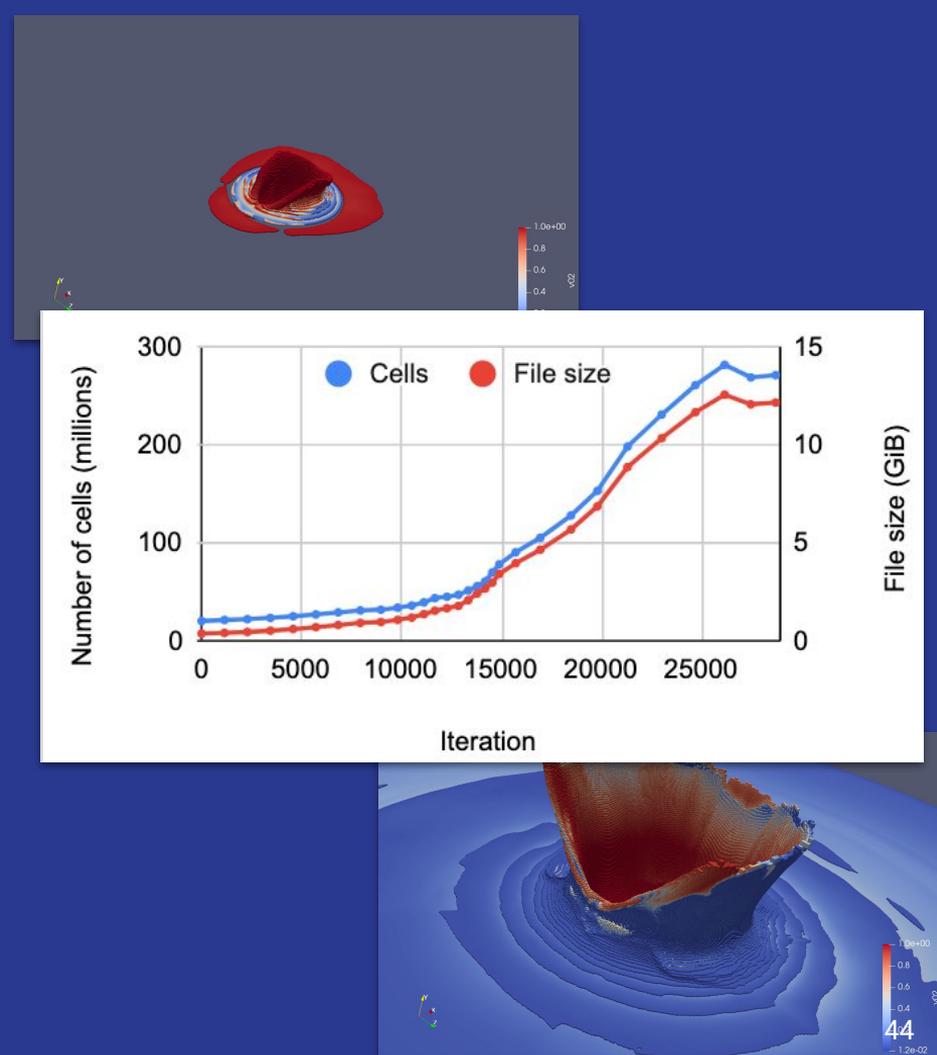


Argonne National Laboratory is a
U.S. Department of Energy laboratory
managed by UChicago Argonne, LLC.

Deep Water Impact

- Asteroid crashing into the ocean
- Unstructured meshes getting increasingly complex as the simulation progresses
- More and more data to store
- More and more complicated to render

We need to be able to add
in situ resources
incrementally as the
simulation progresses



Current in situ frameworks are not built for elasticity

Static algorithms

- Analysis and visualization algorithms assume a **fixed number of processes**
- But they have been optimized for decades, **we can't just throw them away**
- **Restart** the workflow?
- Change the number of processes **between simulation iterations?**

Reliance on MPI

- All the in situ libraries and frameworks today rely on MPI, which **doesn't allow adding and removing processes from a communicator dynamically**
- Some frameworks (e.g. Damaris) split MPI_COMM_WORLD, making it hard to rescale the analysis part without changing the simulation

Solution: replace MPI in existing frameworks with a communication layer that enables elasticity

How tightly coupled to MPI are existing frameworks?

Many frameworks already abstract communications, to some degree

- VTK/ParaView \Rightarrow `vtkCommunicator`, `vtkMultiProcessController`
- IceT \Rightarrow `IceTCommunicator`
- Damaris \Rightarrow `Reactor`, `Channel`

With a bit more work

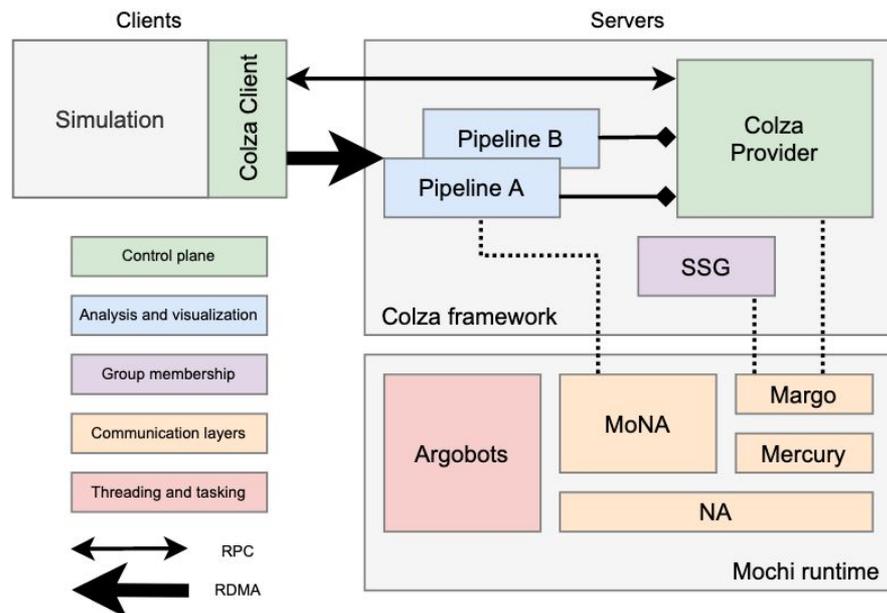
- DIY \Rightarrow communicator class relying on MPI could be made abstract
- VisIt \Rightarrow relies on VTK but hides it under a C interface expecting `MPI_Comm`

Some are too tightly reliant on MPI

- Ascent \Rightarrow Solution: use PMPI interface to overwrite MPI functions

The Colza in situ framework

- **Colza Provider:** lives in each Colza server node, responsible for managing pipelines, receiving RPCs and directing them to pipelines
- **Pipelines:** user-provided object, loadable via plugins, implements analysis/visualization tasks
- **SSG (Scalable Service Groups):** Group membership component, based on the SWIM gossip protocol, notifies the providers when nodes are added/removed
- **MoNA (Messaging over NA):** implementation of MPI-like collective algorithms on top of NA, the networking layer of the Mercury RPC library



Colza's simulation API

activate(iteration): tells all the Colza servers that the iteration of analysis is about to start. The group of Colza processes is no longer allowed to change. Implements two-phase commit to ensure that all the Colza servers have a consistent view of the group.

stage(iteration, data, metadata): sends data to the pipeline(s) using RDMA. The receiving pipeline instance is determined using a user-provided hashing function on the metadata.

execute(iteration): executes the pipeline's code.

deactivate(iteration): tells all the Colza servers that the iteration of analysis is completed. Processes may join and leave until the next activate call.

Collective communication with MoNA

Based on NA, Mercury's networking layer

- Uses unexpected messages for small messages
- Switches to rendez-vous for larger messages
- Switches to RDMA for even bigger messages (thresholds are configurable)

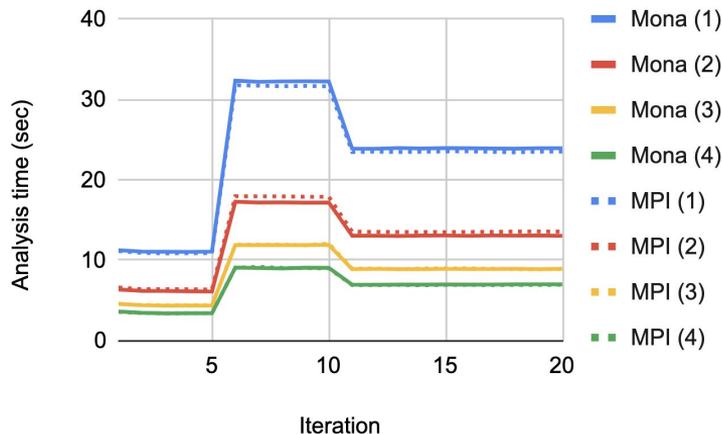
Collective operations

- **Same as MPI** (bcast, reduce, gather, etc.)
- Currently implemented using naive algorithms or inspired by MPICH

In an elastic context

- **No “World” communicator**
- Communicators can be built from any list of Mercury addresses
- Easy to **rebuild communicators** when processes **join** and **leave**

Highlight: AMR-Wind on ANL's Theta



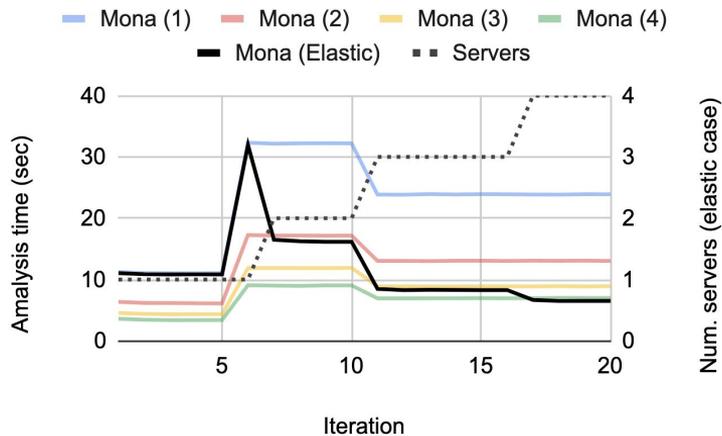
AMR-Wind (MoNA vs MPI)

Colza deploying on 1 to 4 nodes, using either MoNA or MPI

Visualization pipeline implemented using **Ascent**

MPI replaced by overloading PMPI functions to redirect them to MoNA

Highlight: AMR-Wind on ANL's Theta



AMR-Wind (elastic in situ)

Colza deploying on 1 to 4 nodes, adding one node every 120 seconds

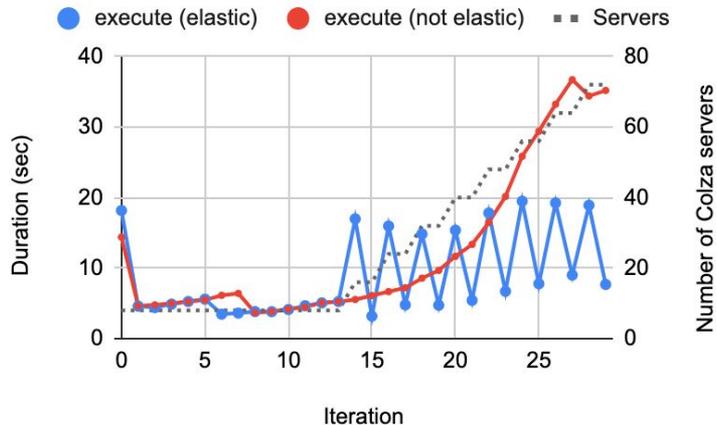
Visualization pipeline implemented using **Ascent**

MPI replaced by overloading PMPI functions to redirect them to MoNA

Elasticity experiment: we start with 1 Colza server and add a new one every 120 seconds

Elasticity allows selecting the desired performance level

Highlight: Deep Water Impact on NERSC' Cori



Deep Water Impact (elastic)

Starting with 1 server (8 processes), we add a new server with 8 processes every other iteration starting from iteration 13

Visualization pipeline implemented using **ParaView Catalyst**

Rendering time **maintained below 20 seconds** (for iterations that don't have to initialize a new server)

Overhead of initializing VTK in new processes when they are added

Showcases the benefit of elasticity to **maintain in situ analysis time under some time constraint**

Possible triggers of elasticity

User-driven

- Add/remove resources depending on what the user wants to do

Performance-driven

- Try to achieve full overlap between simulation and analysis

Data-driven

- Perform more complex analysis when interesting data appears

Platform-driven

- Allow the job scheduler to add/reclaim nodes to optimize resources

Conclusion



Argonne National Laboratory is a
U.S. Department of Energy laboratory
managed by UChicago Argonne, LLC.

The future is bright!

Mochi, componentization, and composition have accelerated our pace of R&D and collaborations

Smart devices, AI, and complex workflows set the stage for another decade of interesting challenges

Thank you!

THIS WORK IS IN PART SUPPORTED BY THE DIRECTOR, OFFICE OF ADVANCED SCIENTIFIC COMPUTING RESEARCH, OFFICE OF SCIENCE, OF THE U.S. DEPARTMENT OF ENERGY UNDER CONTRACT NO. DE-AC02-06CH11357; IN PART SUPPORTED BY THE EXASCALE COMPUTING PROJECT (17-SC-20-SC); AND IN PART SUPPORTED BY THE U.S. DEPARTMENT OF ENERGY, OFFICE OF SCIENCE, OFFICE OF ADVANCED SCIENTIFIC COMPUTING RESEARCH, SCIENTIFIC DISCOVERY THROUGH ADVANCED COMPUTING (SCIDAC) PROGRAM.



Argonne National Laboratory is a
U.S. Department of Energy laboratory
managed by UChicago Argonne, LLC.