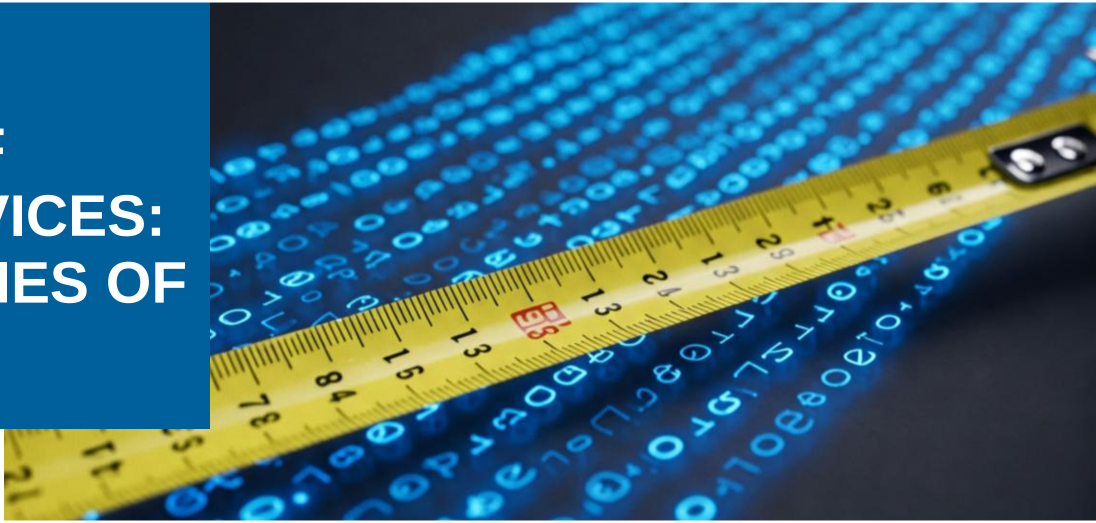
CUG 2025

TOWARDS EMPIRICAL ROOFLINE MODELING OF DISTRIBUTED DATA SERVICES: MAPPING THE BOUNDARIES OF RPC THROUGHPUT



PHIL CARNS, MATTHIEU DORIER, ROB LATHAM, SHANE SNYDER, AMAL GUEROUDJI, ROB ROSS
Argonne National Laboratory

SETH OCKERMAN
University of Wisconsin-Madison

JEROME SOUMAGNE
Hewlett Packard Enterprise

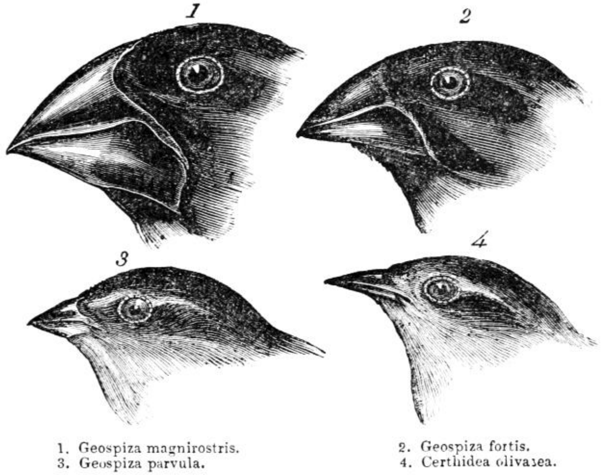
DONG DAI
University of Delaware

May 8, 2025

DISTRIBUTED DATA SERVICES

Parallel file systems and much more

- Distributed services are often used for data management. The most prevalent examples in HPC are parallel file systems.
- Data services can be much more than just file systems, though. Other notable examples include databases, object stores, telemetry capture, data transfer, workflow orchestration, etc.
- It is increasingly common for **specialized services** to be deployed **on demand**.

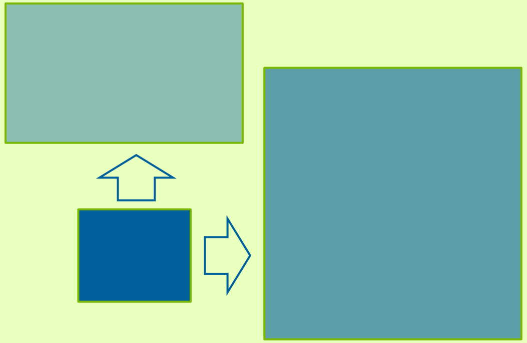


"Darwin's Finches" by James Gould

As with any other HPC software, performance is critical. What's the best way to assess the performance of distributed data services?

OBSTACLES TO UNDERSTANDING DATA SERVICE PERFORMANCE

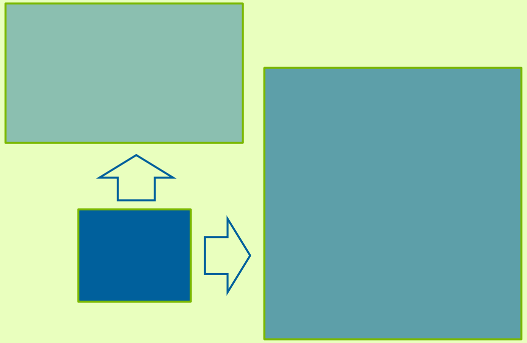
Services can be deployed on demand at a variety of scales.



➢ The model should generalize across provisioning scenarios.

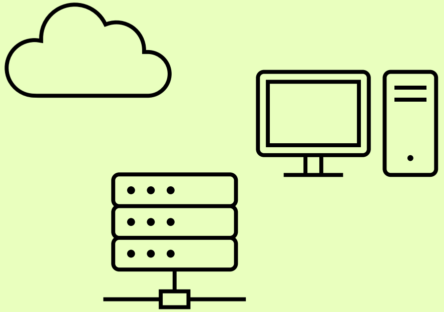
OBSTACLES TO UNDERSTANDING DATA SERVICE PERFORMANCE

Services can be deployed on demand at a variety of scales.



➢ The model should generalize across provisioning scenarios.

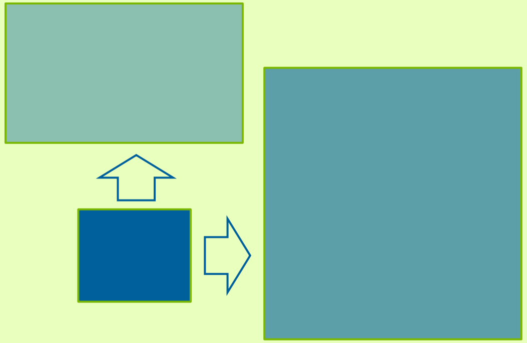
Services may require tuning when ported from a different native environment.



➢ The model should assess performance relative to platform capabilities.

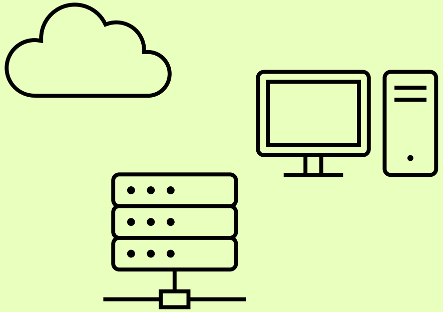
OBSTACLES TO UNDERSTANDING DATA SERVICE PERFORMANCE

Services can be deployed on demand at a variety of scales.



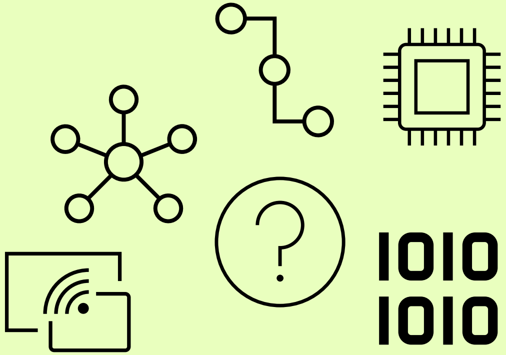
➢ The model should generalize across provisioning scenarios.

Services may require tuning when ported from a different native environment.



➢ The model should assess performance relative to platform capabilities.

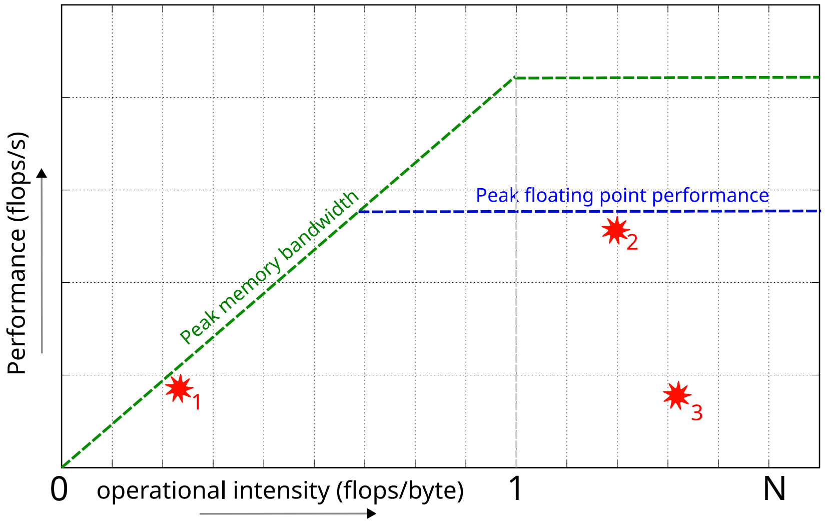
Services do many different things (streaming, queries, computation, staging, etc.).



➢ The model should use flexible metrics for different use cases.

ROOFLINE MODELING

Drawing inspiration from proven computer science techniques

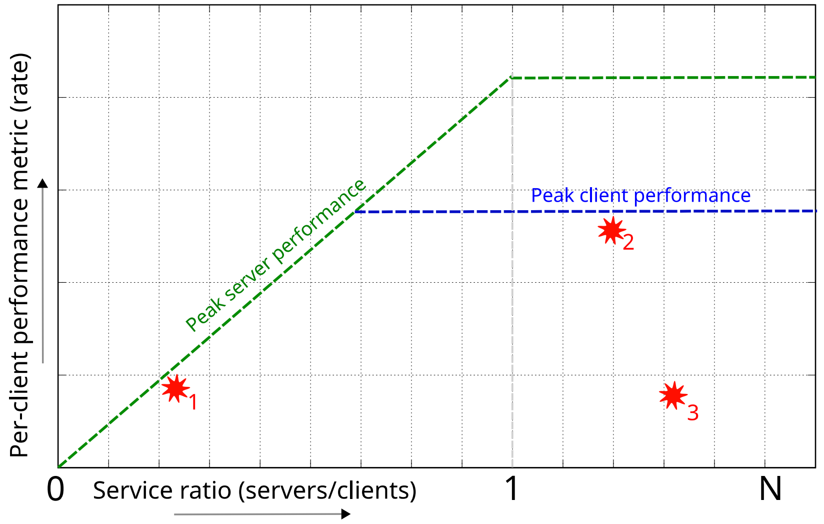


Williams, Samuel; Waterman, Andrew; Patterson, David "Roofline: An Insightful Visual Performance Model for Multicore Architectures" Commun. ACM. 52 (4): 65–76. 2009.

- Roofline models widely used to help understand computer performance.
  - Visually intuitive
  - Don't have to be perfectly precise
  - Can be constructed from a few simple parameters
- Can we adapt the concept?
  - We need to measure distributed data services, not applications.
  - We need to use metrics that are relevant to I/O performance.

ROOFLINE MODELING FOR DATA SERVICES

Drawing inspiration from proven computer science techniques



This is a conceptual diagram of what a roofline model for a distributed data service might look like.

- The X axis is still a ratio, but it indicates the server/client ratio instead of operational intensity.
- More than one metric may be applicable to the Y axis, as long it can be presented as a rate (bytes/s, ops/s, etc.).

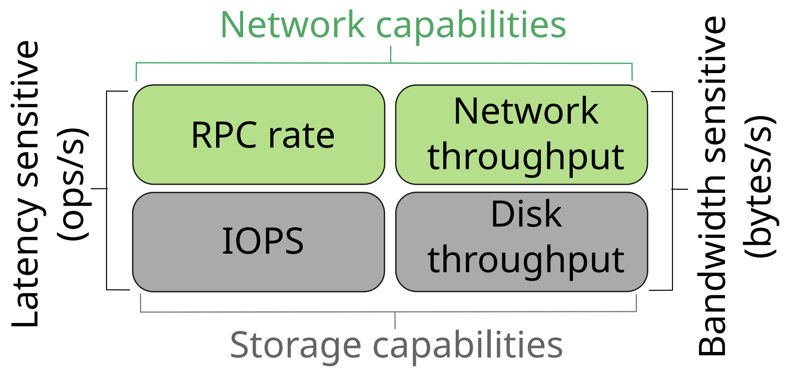
PROBLEM STATEMENT

Can we create empirical roofline models for distributed data services?

This presentation tells the story of how we explored this question. We had some surprises along the way! The plan sounded simple on paper:

1. Extract maximum baseline performance on today’s HPC systems
2. Capture model parameters with repeatable, compact experiments
3. Construct and validate models on several example platforms

THE SCOPE OF OUR INITIAL INVESTIGATION



- There are several obvious metrics that could be relevant for a data service.
- We'll start with network metrics as a proof of concept (top row, in green).
- We'll use Mochi as our framework for benchmarking, using a component called "Quintain" for parameterizing workloads.

Keep in mind that we want our models to give us an intuitive understanding of *data service performance*, not application performance.

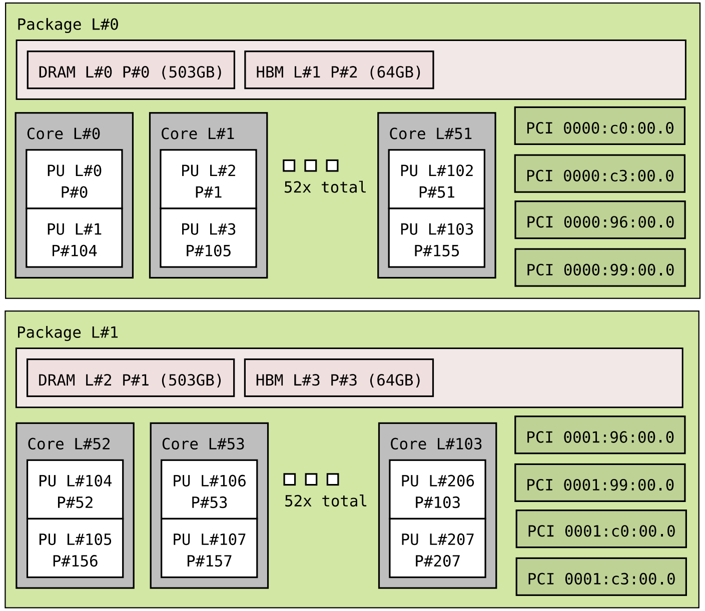
TEST PLATFORMS

- Aurora (ALCF)
- Polaris (ALCF)
- Frontier (OLCF)
- Perlmutter (NERSC)
- Improv (ANL LCRC)
  for select experiments

ADVENTURES IN OPTIMIZING AND MEASURING DATA SERVICE PERFORMANCE

GETTING RESOURCE SELECTION RIGHT

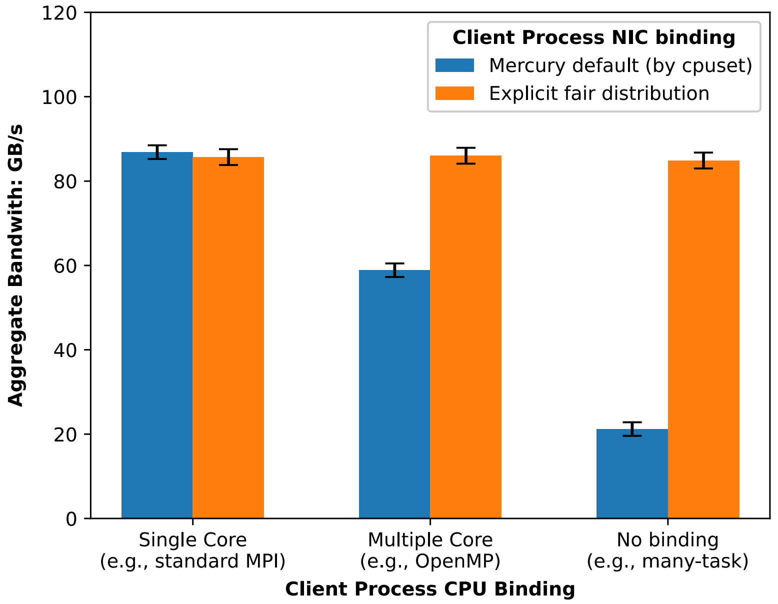
- We quickly found that would need to run many processes per node to maximize performance (more on that later); this led to a question of how to best select resources for each process.
- CPU core selection is easy; you can control that with well-known scheduler options.
- Network cards selection is somewhat trickier.
  - MPI implementations have great solutions for this already, but data services need to deal with *heterogeneous, ephemeral processes*, and we have *no global rank* to help arbitrate conflicts.



Simplified view of Aurora's node topology. Each PCI device corresponds to a distinct Slingshot network card attached to the same system fabric.

BALANCING NETWORK CARD TRAFFIC

How bad could it be?

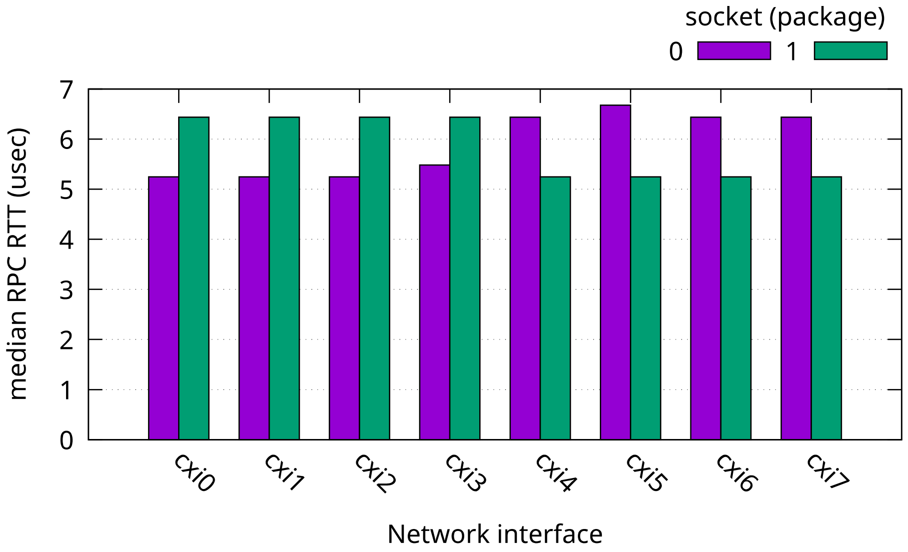


Aurora: bandwidth for 16 client processes in different application scenarios with static core/NIC mapping strategy.

- Our starting assumption: a static mapping of cores (designated by cpuset) to network cards would be fine for most cases.
- Conventional MPI (1 core per proc): great ☺
- MPI+X (2 cores per proc): the mapping is unbalanced; we lost some performance 😐
- Many-task (binding disabled): degenerates into everyone using the same NIC ☹
- Orange bars show what throughput could be if we explicitly mapped each process to a "good" network card instead.

CONSIDERING LOCALITY

Why not just assign network cards first come, first serve?



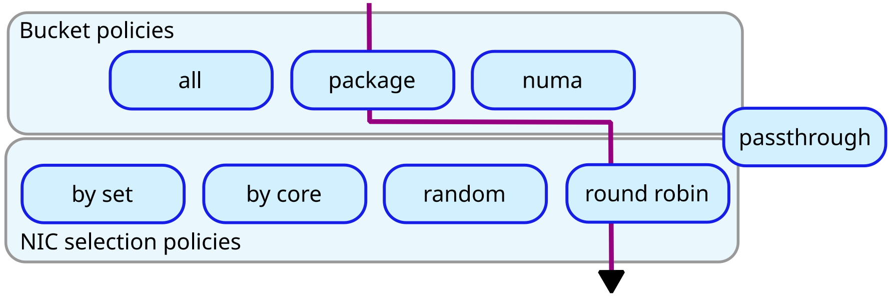
Aurora: round-trip RPC latency between nodes for each package/NIC combination.

- Is there a penalty if we end up mapping some processes to non-local NICs?
- Yes. On Aurora, for example, it costs roughly 1.2 microseconds per RPC.

Note that round-trip time in this context includes a full data service stack with encoding, decoding, and handler execution; none of these times are as fast as hardware specifications.

NETWORK SELECTION WITH MOCHI-PLUMBER

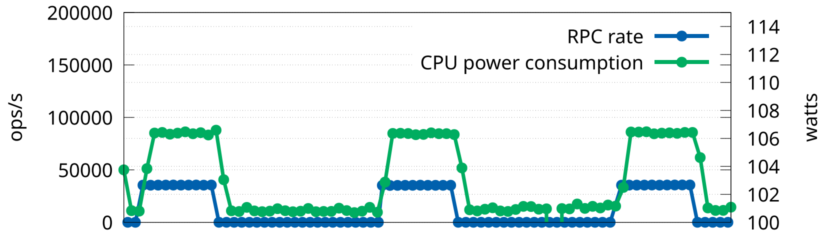
Configurable network card selection policies for data services



- Based on these experiments, we implemented a new Mochi component, called “mochi-plumber”, to provide transparent resource selection.
- Bucket policy: divides network cards into buckets. The default policy groups them by CPU socket locality.
- NIC selection policy: assigns network cards to processes within a bucket. The default policy uses control files in /tmp for round robin assignment.

Data shown from the Improv system at ANL using an InfiniBand network.

GETTING ADAPTIVE POLLING RIGHT



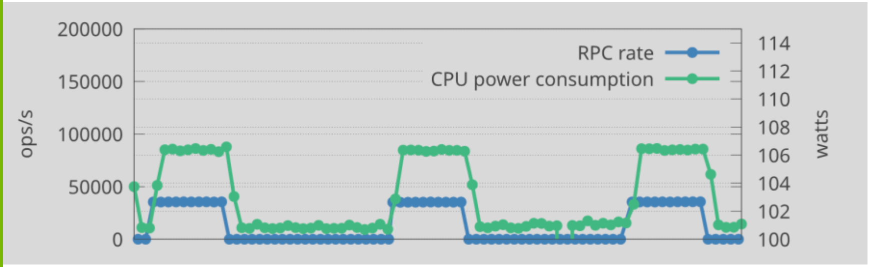
This figure shows the correlation between RPC rate and power consumption for bursts of sequential operations with one client and one server on Improv.

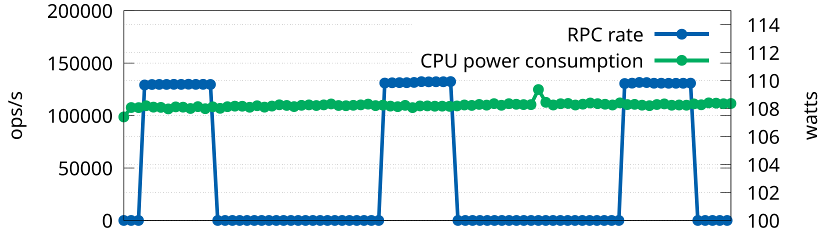
The correlation is as expected, but the rate is rather slow (<100,000 ops/s).

- The Margo component is responsible for Margo's polling strategy; it mostly controls this by varying the timeout parameter when it interacts with the RPC layer (Mercury).
- Our starting assumption was that adaptive polling was a pretty good general strategy across workloads: block when idle, poll when busy.
- However, we found some surprising behavior when we focused on point-to-point RPC rate for use as a roofline parameter.

Data shown from the Improv system at ANL using an InfiniBand network.

GETTING ADAPTIVE POLLING RIGHT

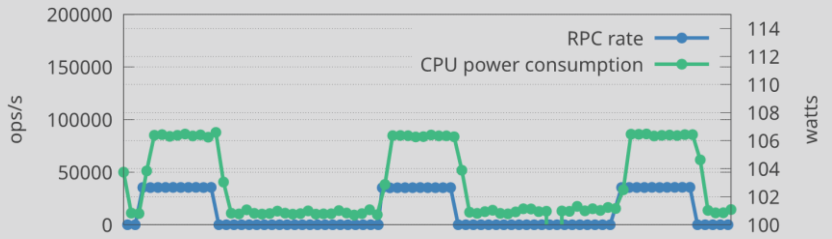


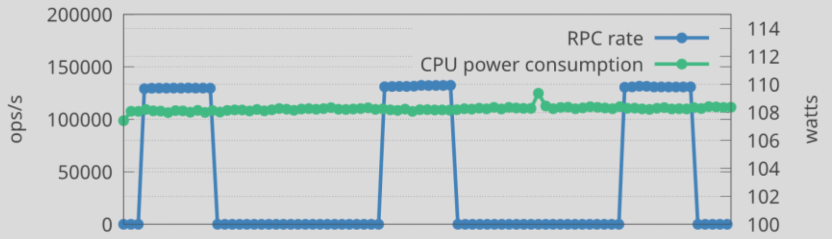


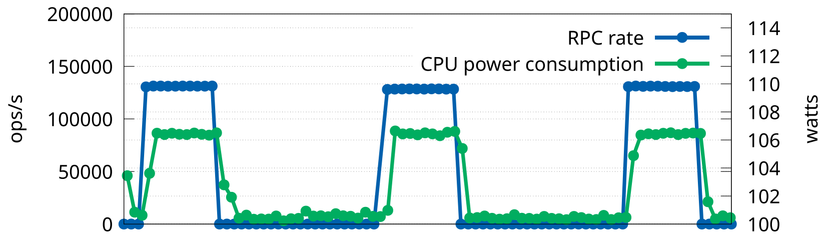
- We could busy poll for events at all times as an alternative to adaptive polling.
- This yields a higher rate during bursts, but at the cost of constant power draw and (not shown) CPU utilization/contention.
- Why would this be any faster, though?
- The problem turned out to be that our adaptive polling was too greedy; it attempted to return to an idle/blocking state in the ~2 microsecond window between sequential single-client RPCs!

Data shown from the Improv system at ANL using an InfiniBand network.

GETTING ADAPTIVE POLLING RIGHT







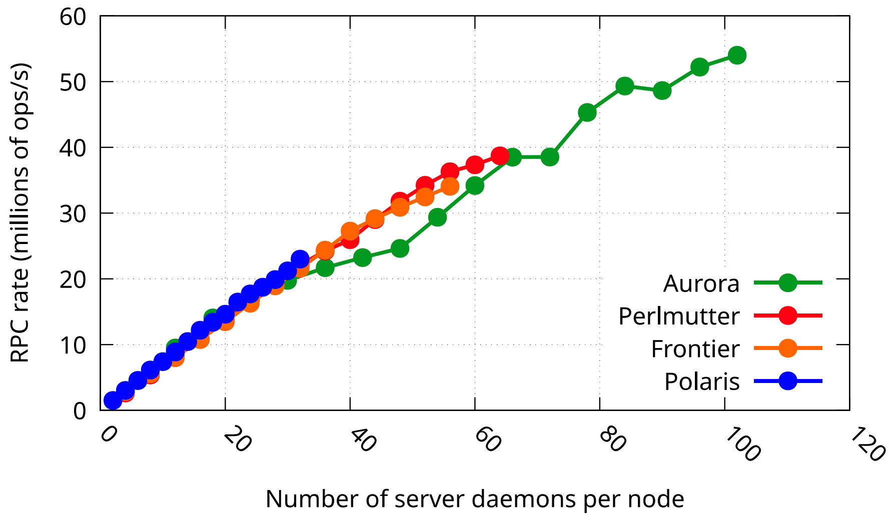
- Based on these measurements, our solution was to implement a new “spindown” policy.
- We still use adaptive polling, but it does not return to a pure idle state until some amount of time (default 50 milliseconds) has elapsed without a new RPC arrival.
- The result is finally what we expected: good idle behavior (which is important for mixed workloads) and maximum performance under load.

We would not have found this without systematic evaluation of corner cases.

HOW MANY PROCESS PER NODE?

Assuming we want to maximize dedicated, per-node performance

- How many server processes should you put on a node to maximize per-node performance?
- Our initial assumption was that we would need “a few” servers per node.
- To our surprise, the best configuration was actually “as many as possible” if we wanted to maximize RPC rate.
- RPC handling is highly CPU bound due to decoding, encoding, and spawning user-level thread handlers.

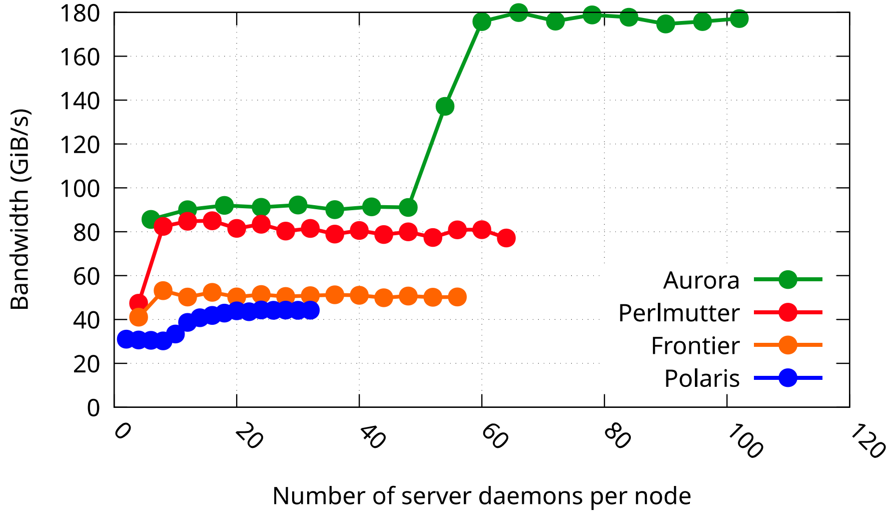


Maximum RPC rate per node as we increase the number of daemons per node. Aurora achieves over 50 Million ops/s per node simply because it has more available physical cores per node (102).

HOW MANY PROCESS PER NODE?

But what about bandwidth?

- After seeing the RPC numbers, we then guessed that there might be a tradeoff: use more cores for higher ops/s, use fewer cores for higher bandwidth (due to contention).
- Happily, we were wrong. Nodes saturate quickly as we add more processes, but they do not taper off at scale.
- Aurora has a notable “jump” in this graph as processes spill over to the 2nd socket in this configuration.



Maximum bandwidth per node as we increase the number of daemons per node. Aurora achieves nearly 180 GiB/s per node because it has 8 x 200 Gb/s network cards.

HOW TO ACCOUNT FOR VARIABILITY

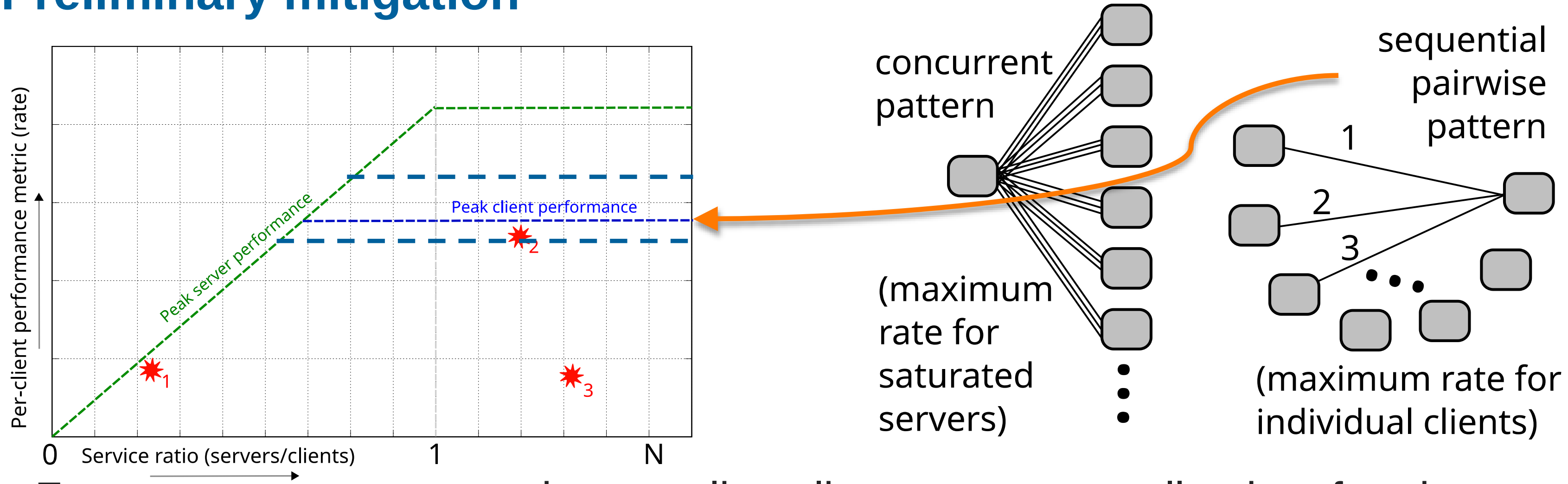
Not all nodes and links are equivalent

For a given network metric, we only need two parameters to construct our proposed roofline model:

- Peak server rate
  - We measure this with saturated servers; they achieve higher throughput under concurrent workloads.
- Peak client rate
  - We measure this with a single client accessing an otherwise idle server.
  - This measurement is susceptible to varying performance between nodes, mostly due to topology, but also occasionally due to node-level problems.
  - *We anticipated this problem, but it was worse than we expected.* On Polaris, **we observed a range of 84k to 180k RPC/s** between different pairs of nodes!

HOW TO ACCOUNT FOR VARIABILITY

Preliminary mitigation



Per-client performance metric (rate)
Peak server performance
Peak client performance
1
2
3
0
Service ratio (servers/clients)
1
N
concurrent pattern
(maximum rate for saturated servers)
sequential pairwise pattern
1
2
3
(maximum rate for individual clients)

- For now, we compensate by sampling client rate across all pairs of nodes (N(N-1)/2) in a job. For 17 node jobs, that yields 136 distinct pairs/samples.
- We use this data to express the peak client rate ceiling as a range in our model.
- Future work: we need to find a better way to account for this; more on that later.

PLATFORM SURVEY AND VALIDATION

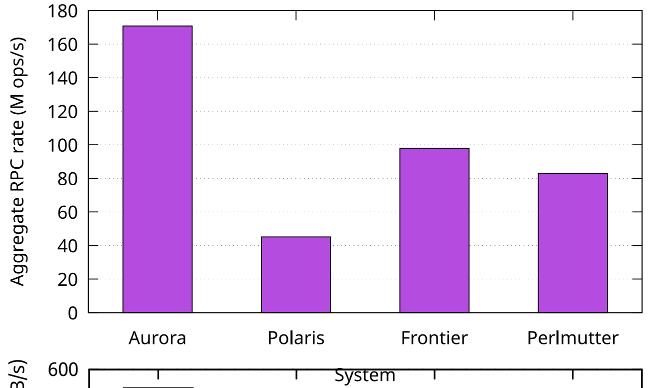
SURVEY METHODOLOGY

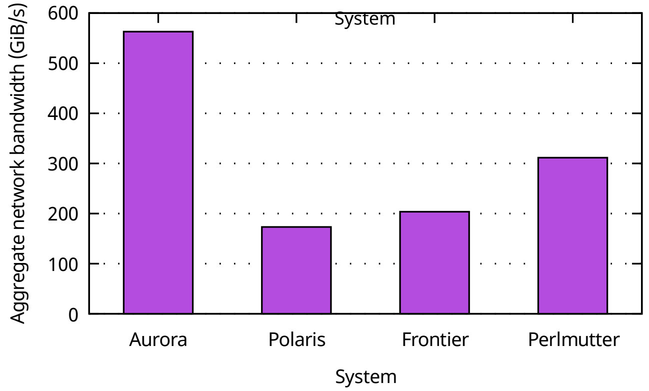
For each of the four primary systems in our study:

- We ran a benchmark suite in a 17 node job to extract model parameters.
  - RPC rates:
    - Server and client peak rate
  - Network bandwidth:
    - Server and client peak rate
- We then ran a separate validation job that varied the number of servers (1-4 nodes) and number of clients (1-16 nodes) and plotted every sample against the roofline to see how well it matched.

SOME HEADLINE NUMBERS

Peak validation rates observed with 4 servers on each system



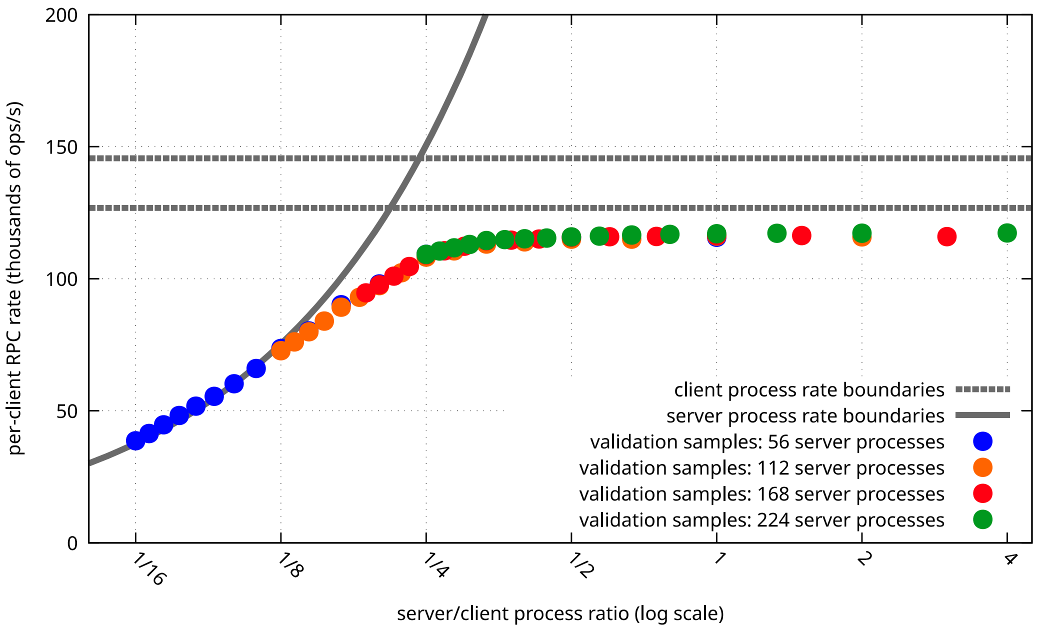


- Peak RPC rate (top) and bandwidth (bottom) mostly followed expectations based on node specifications. We show the fastest examples with 4 server nodes here.
- A few surprises show up when comparing Frontier and Perlmutter (they have similar core and NIC counts).
  - RPC rate: Frontier wins, we suspect due to core specialization helping to avoid jitter.
  - Bandwidth: Perlmutter wins, we suspect due to node topology (Frontier favors NIC connectivity to GPUs, not CPUs).

RPC RATE ROOFLINE: A GOOD EXAMPLE

Frontier

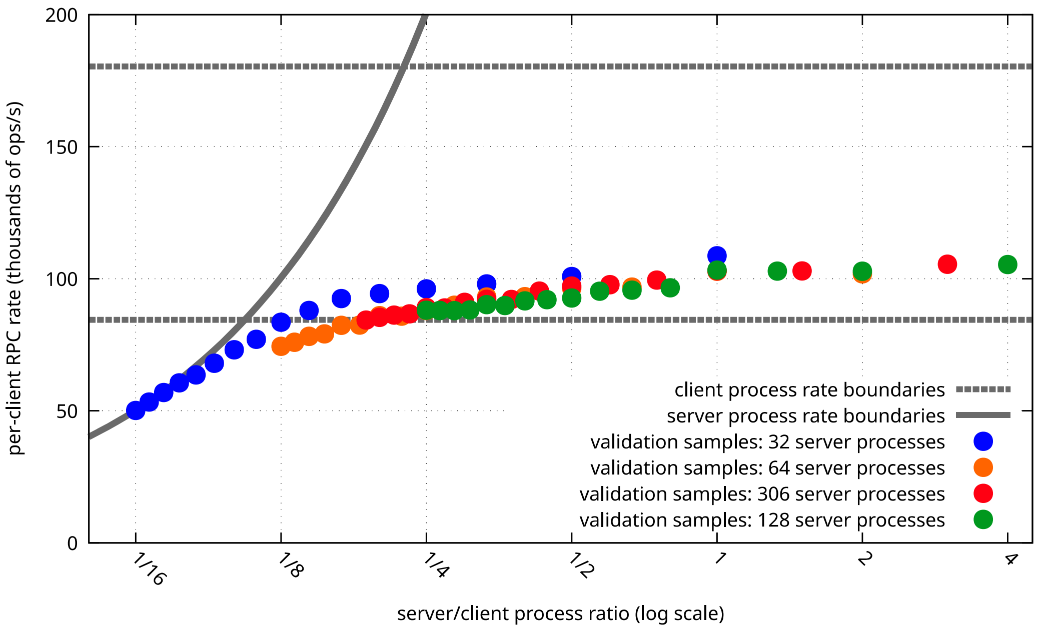
- Gray lines: the roofline model
- Colorful points: validation samples
- Note that the leading edge is curved; we are using log/linear axes here to show detail.
- Pretty good agreement, particularly when servers are saturated.
- The "optimal" server/client ratio is roughly 1/4 for RPC rate.



RPC RATE ROOFLINE: A WEAKER EXAMPLE

Polaris

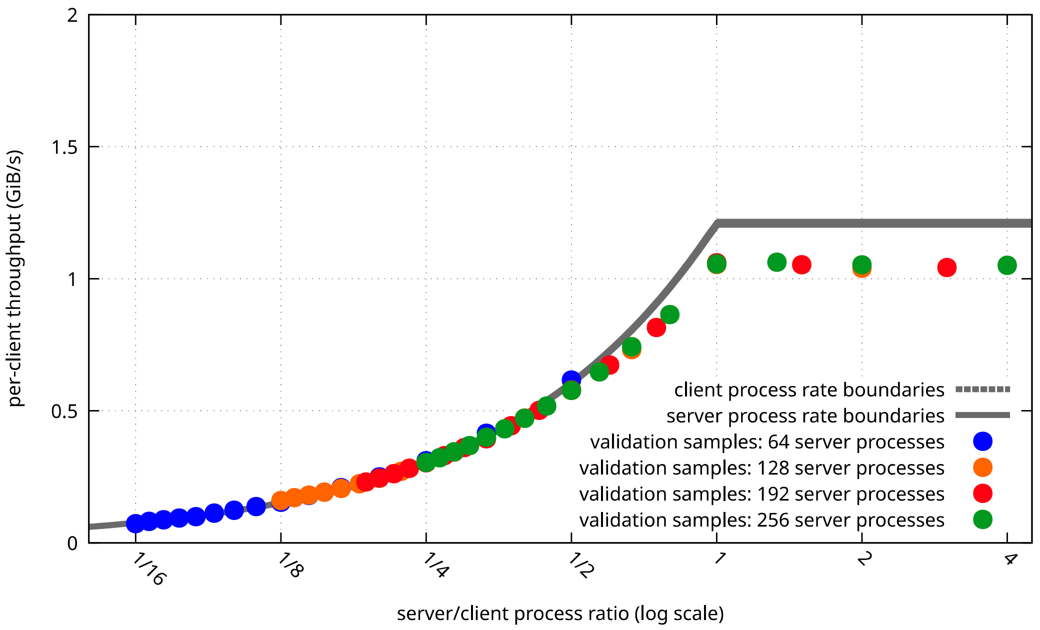
- Recall from earlier that Polaris exhibited far more pairwise variability than the other systems.
- The roofline range is therefore enormous. Its not clear how useful this is for non-saturated servers.
- Variability is evident in the validation samples as well.



BANDWIDTH ROOFLINE: A GOOD EXAMPLE

Perlmutter

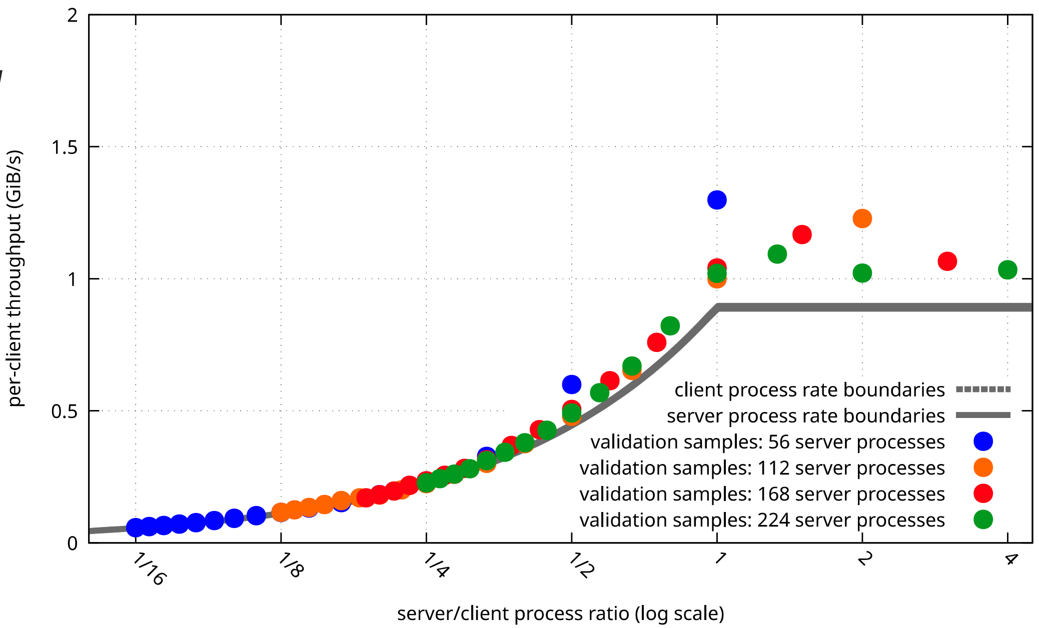
- Note that the client rate is not visible in these graphs.
  - A single client’s potential maximum rate is far higher than what a server process can sustain with multiple servers per node.
- We observe an “optimal” server/client ratio of 1/1, which is simply the equilibrium point for network link capacity.



BANDWIDTH ROOFLINE: A WEAKER EXAMPLE

Frontier

- This was the only example in our survey where validation measurements routinely *exceeded* the roofline model.
- This trend is consistent in follow-up experiments on Frontier: servers out-perform expectations when they are *not saturated*.
- We again suspect Frontier's node topology to be a key factor; it may yield different characteristics depending on utilization.



CONCLUDING REMARKS

SOME FINDINGS

We already pointed out a few along the way, but broadly:

- Getting performance out of “heavy” compute nodes takes some effort.
  - Processes per node, resource selection, and polling strategies all had a big influence. *We automated as much as we could for future users of Mochi.*
- Wow, these systems are ***really*** fast!
- It is possible to measure empirical roofline parameters in a relatively compact benchmark suite.
- Our proposed roofline approach worked best for:
  - Modeling performance for saturated servers
  - Gaining intuition about optimal server/client ratios
- The biggest challenge was:
  - Accounting for topology and variability

FUTURE WORK

This is promising, but there is much yet to do

- Find a better way to account/control for network topology.
  - Should we use bigger sample sets?
  - Can we instrument topology during measurements?
  - Can we can stage experiments that control for topology by requesting specific nodes and racks?
- Integrate storage device metrics (IOPs and bandwidth).
  - Will this be more stable (because network topology is not a factor) or less stable (because storage devices are notoriously inconsistent)?
- Reduce the run time of the benchmark suite. It isn’t that bad, but it could be a lot better. This may be a factor as we incorporate more metrics.
- Use the models to evaluate real-world data services to see what we can learn.

THANK YOU!

THIS WORK WAS SUPPORTED BY THE U.S. DEPARTMENT OF ENERGY, OFFICE OF SCIENCE, ADVANCED SCIENTIFIC COMPUTING RESEARCH, UNDER CONTRACT DE-AC02-06CH11357.

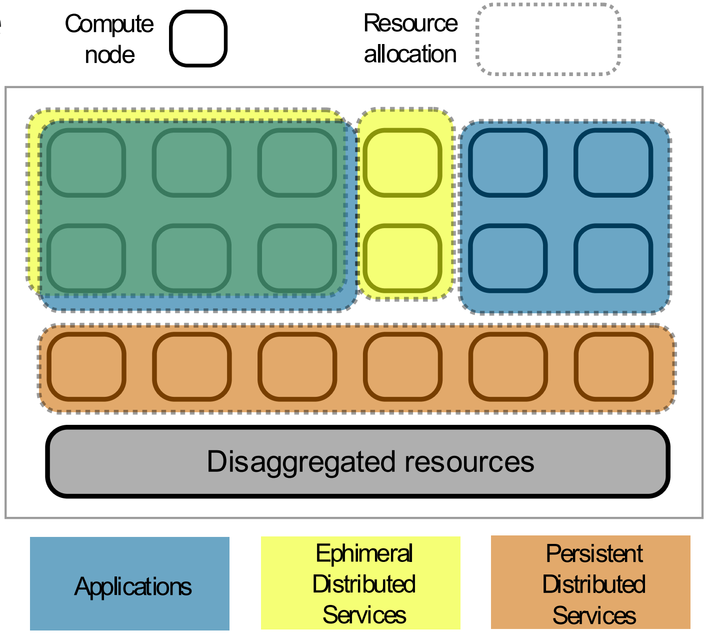
BACKUP SLIDES

BACKGROUND: DISTRIBUTED SERVICES

and the modern runtime ecosystem

A **distributed service** aggregates a collection of on-demand capabilities into a coherent whole, outside of the scope of an application.

- They are useful for a variety of reasons:
  - Decouple functionality from the application (i.e., “software disaggregation”)
  - Enable access to off-node resources
  - Manage state beyond the lifetime of a single application execution
  - Mediate shared access



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

Argonne NATIONAL LABORATORY

WHAT IS MOCHI, EXACTLY?

Mochi seeks to transform this data service monoculture into **an ecosystem of specialized services that are tailored to suit specific use cases and problem domains**. It accomplishes this by providing methodologies and tools for the rapid development of distributed HPC data services.

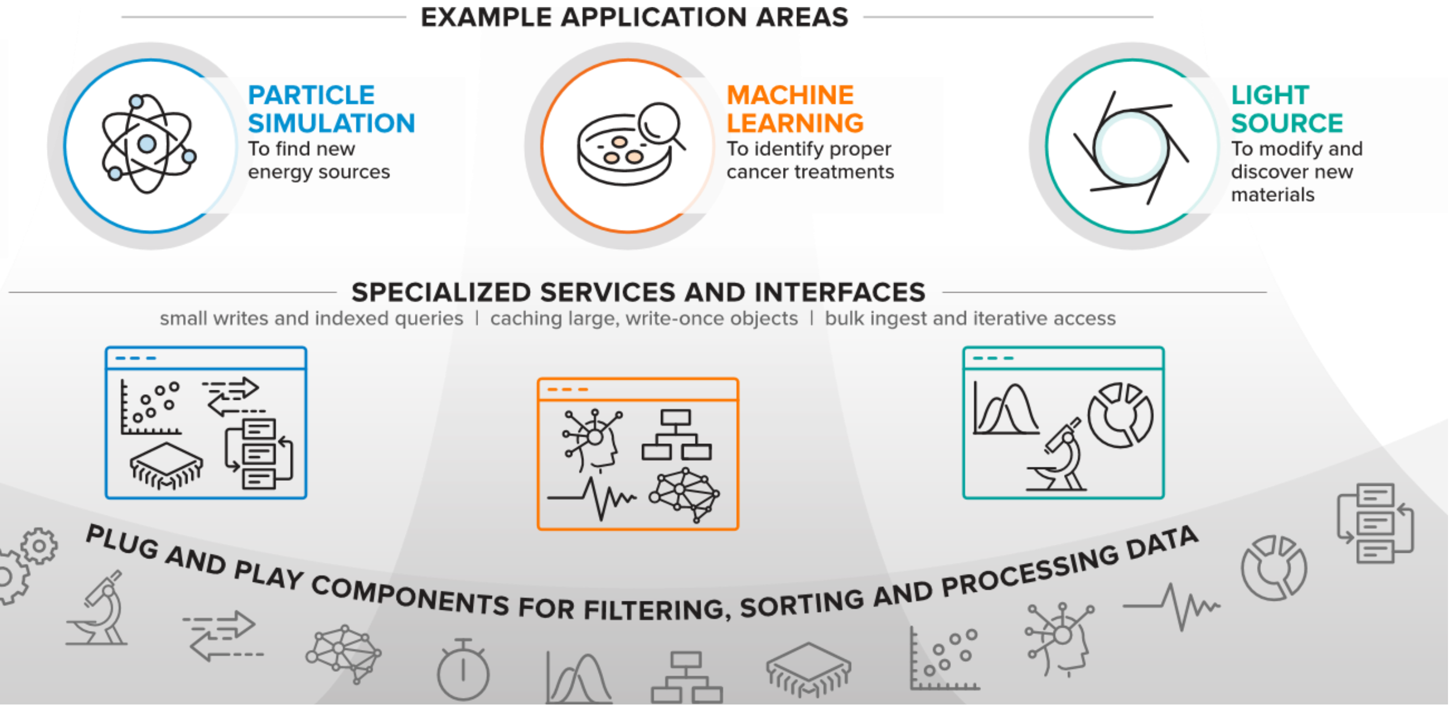
- A collection of reusable, robust, performant microservices and components
- A methodology for composing them into novel, domain-specific services
- API bindings in C, C++, or Python

Mochi services are intended to augment, not replace, mission-critical parallel file systems in the HPC runtime ecosystem.

MOCHI

A framework and methodology for customizable services

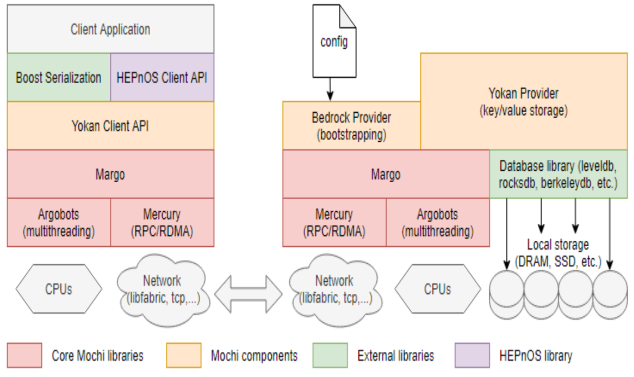
State-of-the-art open source tool for rapid development of customized data services, involving high-performance computing, big data, and large-scale learning.


EXAMPLE APPLICATION AREAS
PARTICLE SIMULATION
To find new energy sources
MACHINE LEARNING
To identify proper cancer treatments
LIGHT SOURCE
To modify and discover new materials
SPECIALIZED SERVICES AND INTERFACES
small writes and indexed queries | caching large, write-once objects | bulk ingest and iterative access
PLUG AND PLAY COMPONENTS FOR FILTERING, SORTING AND PROCESSING DATA

MOCHI EXAMPLES

Ali *et al.*, "HEPnOS: a Specialized Data Service for High Energy Physics Analysis," *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) 2023.*

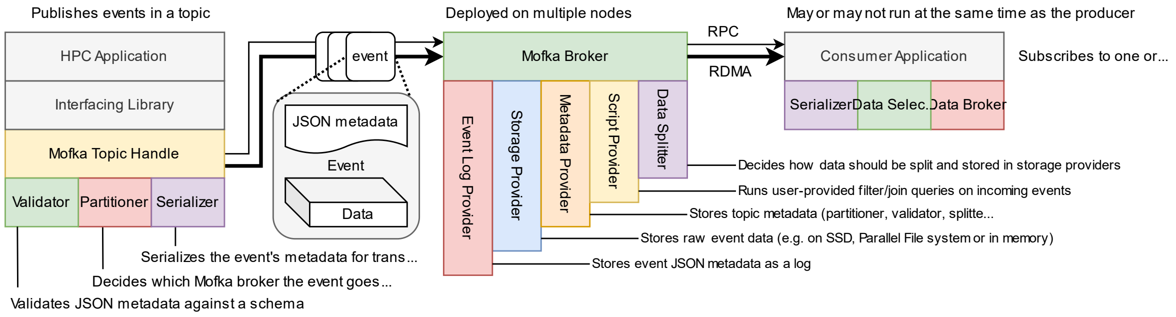
- **HEPnOS (below)**
- Domain-specific service for HEP experiment analysis
- Presents hierarchical sorted data amenable to analysis



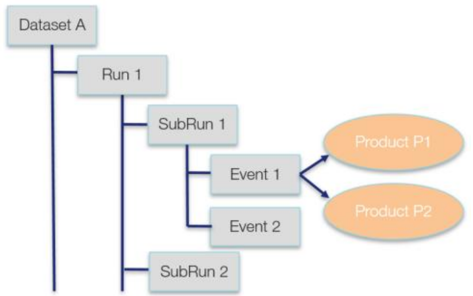
Client Application
Boost Serialization
HEPnOS Client API
Yokan Client API
Margo
Argobots (multithreading)
Mercury (RPC/RDMA)
CPUs
Network (libfabric, tcp,...)
config
Bedrock Provider (bootstrapping)
Yokan Provider (key/value storage)
Margo
Database library (leveldb, rocksdb, berkeleydb, etc.)
Mercury (RPC/RDMA)
Argobots (multithreading)
Local storage (DRAM, SSD, etc.)
Network (libfabric, tcp,...)
CPUs
Core Mochi libraries
Mochi components
External libraries
HEPnOS library

- **Mofka (below)**
- Streaming event service
- Analogous to Kafka but tailored to scientific computing

https://mofka.readthedocs.io



Publishes events in a topic
HPC Application
Interfacing Library
Mofka Topic Handle
Validator
Partitioner
Serializer
event
JSON metadata
Event
Data
Serializes the event's metadata for trans...
Decides which Mofka broker the event goes...
Validates JSON metadata against a schema
Deployed on multiple nodes
Mofka Broker
Event Log Provider
Storage Provider
Metadata Provider
Script Provider
Data Splitter
RPC
RDMA
May or may not run at the same time as the producer
Consumer Application
Subscribes to one or...
Serializer
Data Selec.
Data Broker
Decides how data should be split and stored in storage providers
Runs user-provided filter/join queries on incoming events
Stores topic metadata (partitioner, validator, splitte...
Stores raw event data (e.g. on SSD, Parallel File system or in memory)
Stores event JSON metadata as a log



Dataset A
Run 1
SubRun 1
Event 1
Event 2
SubRun 2
Product P1
Product P2