SNOflake – A Scalable Network Overlay for Multi-domain HPC Environments

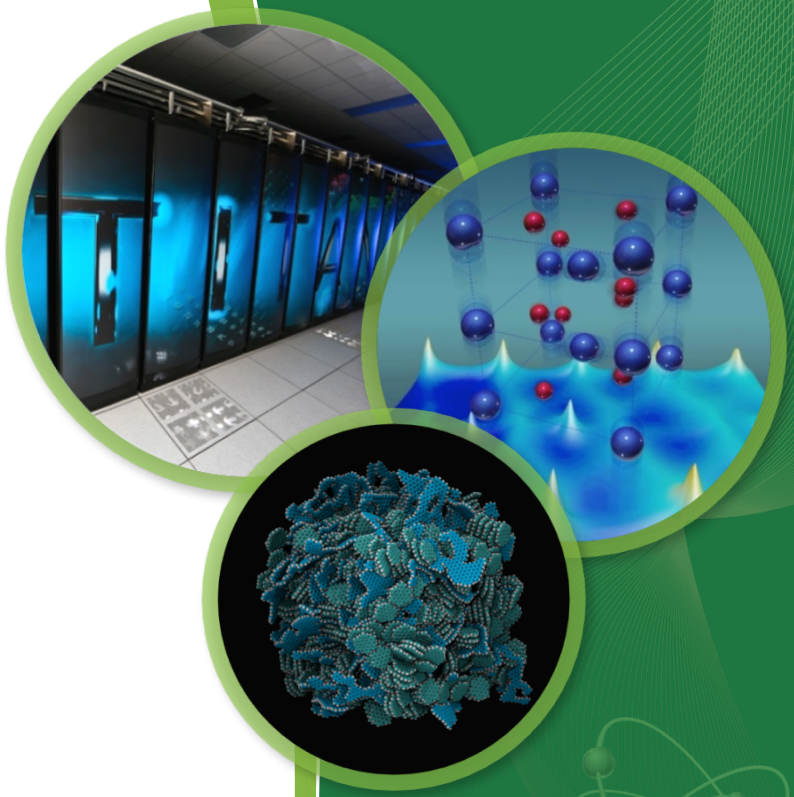
Michael J. Brim (brimmj@ornl.gov)

Research Staff, CSMD, ORNL

2017 Scalable Tools Workshop

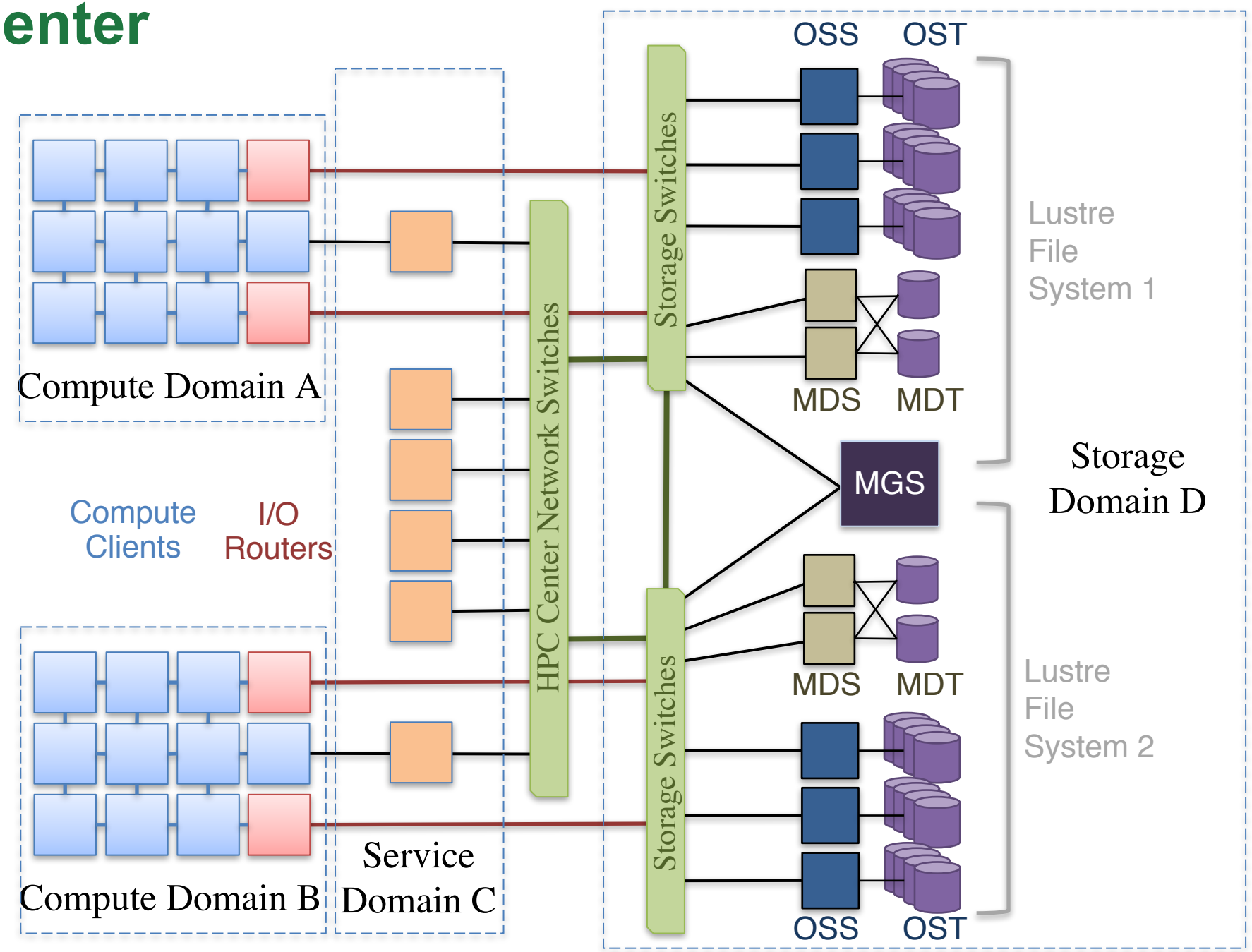
Granlibakken Resort, Lake Tahoe, CA

August 7, 2017



A Typical HPC Center Environment

- Multiple compute systems
- Service network
- Shared storage systems



Multi-domain Overlay Use Cases

- Distributed system services
  - resource management, scheduling, and monitoring
  - parallel application runtimes
  - system knowledge base (e.g., key-value store)
- Distributed tools
  - system administration and monitoring
  - performance and debugging tools
  - computational steering & visualization
- Distributed applications
  - data coupling for cross-domain scientific applications
  - Big Data analytics

Design Goals

- General-purpose overlay network infrastructure for constructing distributed services, tools, and apps
  - bootstrapping and distributed launching
    - system-level and user-level
    - deployments spanning multiple domains
  - peer and group communication
    - leverage advanced HPC network capabilities (e.g., RDMA or collectives)
  - integrated, customizable data analysis and aggregation
- Real Scalability: no changes to core design/architecture required for use on future “extreme scale” systems
- Real Resilience: overlay network should remain available for as long as any of the constituent distributed systems are operational

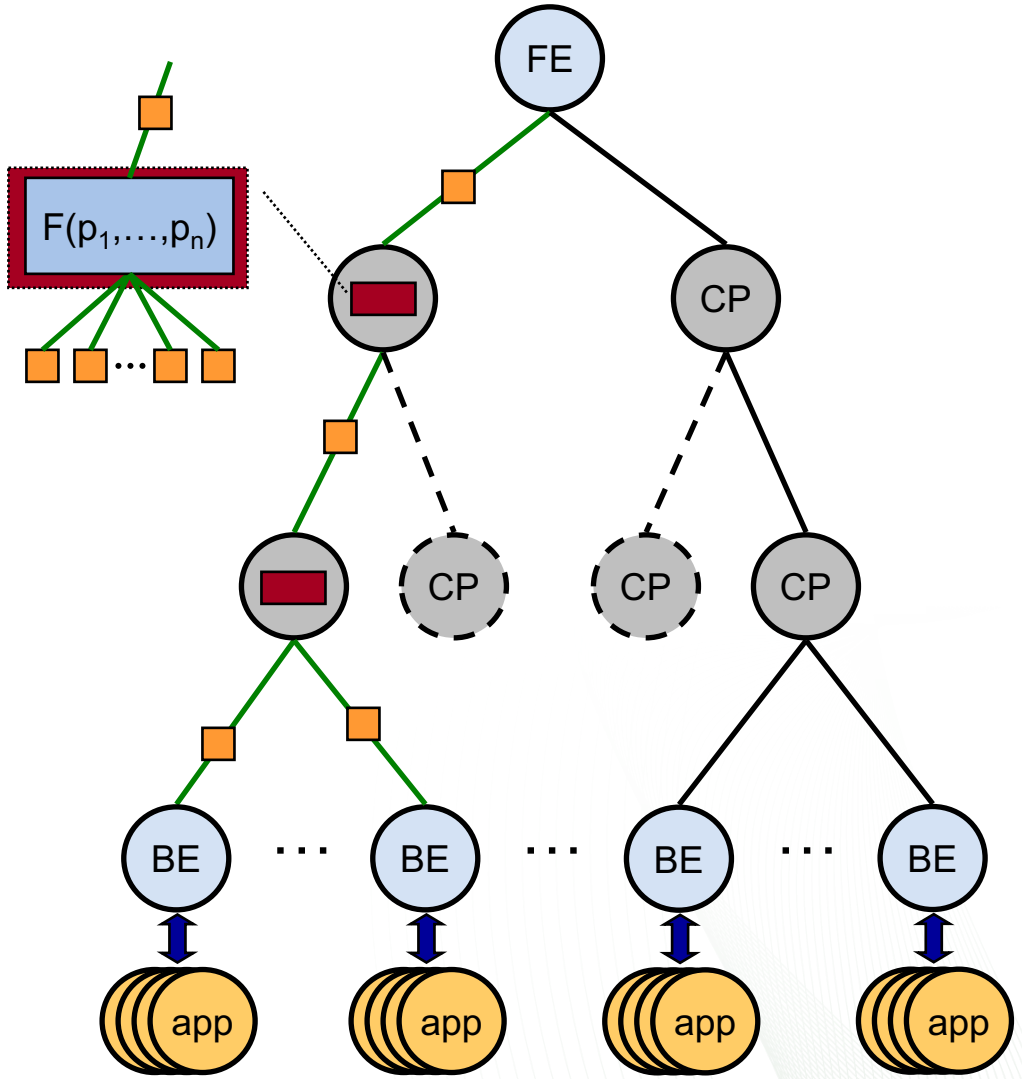
Tree-Based Overlay Networks (TBŌNs)

- Tree overlay architecture provides logarithmic scaling for operations between root and leaves
  - root ➔ leaves: broadcast and multicast
  - leaves ➔ root: data gather and reduction

- MRNet: the Multicast/Reduction Network
  - http://www.paradyn.org/mrnet/
  - general-purpose scalable infrastructure for tools and applications that use a master-worker architecture
  - currently used by many HPC performance and debugging tools
  - in production use at full-scale on leadership HPC systems including ORNL Titan and LLNL Sequoia
  - partial fault-tolerance via automatic recovery from loss of internal tree communication processes

MRNet Overview

- General-purpose TBŌN API
  - Network: user-defined topology
  - Stream: logical data channel
    - to a set of back-ends
    - multicast, gather, and custom filter reduction
  - Packet: collection of data
  - Filter: stream data operator
    - synchronization
    - transformation
- Tool developer writes front-end (FE), back-end (BE), and Stream Filter code using library API
- MRNet provides communication process (CP) executable



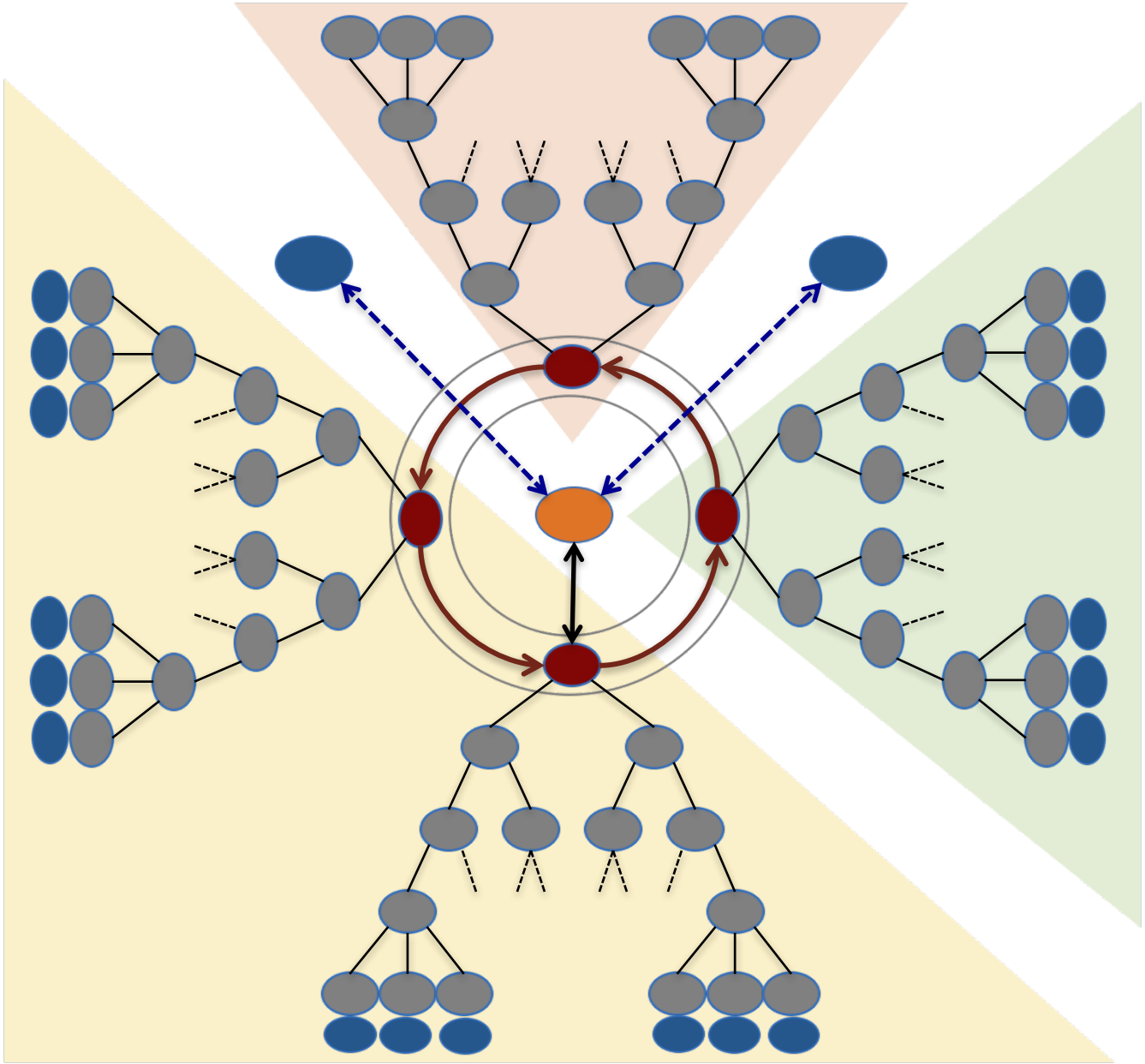
SNOflake Design Characteristics

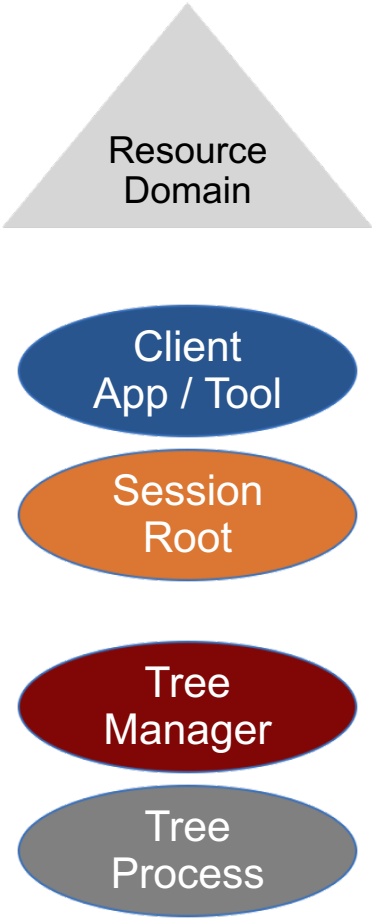
- Support for cross-domain overlay deployments
- Simple yet flexible API in C
  - *Session* represents an overlay shared among clients
  - each *Session* supports many logical *Services*
  - each *Service* supports many data *Streams*
  - *Streams* used to transfer/process opaque *Data Buffers*, rather than formatted Packets
  - *Filter Graph* instead of single filter per Stream
- Ability to leverage advanced HPC networking capabilities
  - for broadcast, multicast, gather
  - for high-bandwidth large data transfers

SNOflake Architecture : A Ring of TBŌNs

- Deploy TBŌNs on separate resource domains
  - place *Tree Managers* (i.e., TBŌN roots) on hosts with inter-domain communication capability
  - use separate trees for distinct resource classes within same distributed system (e.g., compute, management, storage)
- Ring of Tree Managers
  - data routing between TBŌNs
  - state replication within ring for fault tolerance
- SNOstorm - “SNOflake as a Service”
  - persistent, at-boot SNOflake provides bootstrap/launch service for scalable deployment of additional SNOflake-based services, tools, and apps

SNOflake Architecture: A Ring of TBŌNs





But, not a ring of MRNets

- MRNet is not a good fit for:
  - concurrent, independent services sharing same overlay
  - ephemeral tool behavior (especially at FEs)

- MRNet's C++ API and implementation is also pretty rigid
  - fixed threading model: 2 threads per peer
  - Packet data formats
    - basic types (char, int, unsigned, float, double) and strings
    - arrays of those things
  - Stream filtering model
    - single transformation filter (up or downstream)
    - synchronization filter only on upstream
  - custom event management

“We can rebuild him. We have the technology.”

- Overlay (aka session) can be shared by many services
  - Services support multiple streams
  - Filters registered in session
- Everything has a name: sessions, services, streams, filters
  - Attach/detach to services (FEs too)
  - Subscribe to streams (FEs too)
    - no explicit lists of participants to manage
- Flexible filtering through filter graphs
- Flexible threading model
  - thread pools of configurable size for send, recv, filter compute
  - can run single-threaded if necessary (explicit progress call)

SNOflake Data Filtering

- Stream filter graph specification
  - `SRC, SNK` are reserved words
  - `filter=“name”` specifies registered filter’s name
  - optional filter parameters encoded within name string
    - `filter="SYNC_BATCH(4,2000)"`
- Filter libraries are provided per domain to support heterogeneity

```
strict digraph {
  SRC -> SYNC -> AGGR-> SNK;
  SYNC [filter="SYNC_TAG_WAVE"];
  AGGR [filter="DATA_CONCAT"];
}
```

SNOflake Data Filtering – Built-in Filters

- `SYNC_TAG()`
  - group input data buffers by tag, output sets of same-tag buffers
- `SYNC_BATCH(size [, ms])`
  - group input data buffers into batches of given `size`, optionally waiting up to `ms` milliseconds for other data buffers
  - at full batch or expiration of timer, output all data buffers
- `SYNC_TAG_BATCH(size [, ms])`
  - same as SYNC_BATCH, but adds tag grouping like SYNC_TAG
- `SYNC_WAVE()`
  - wait for an input data buffer from each predecessor in the filter graph, or from all subscriber peers if predecessor is SRC, output full wave
- `SYNC_TAG_WAVE()`
  - same as SYNC_WAVE, but for tag sets
- `DATA_CONCAT()`
  - concatenate all input data buffers into single output data buffer

Other Notable Differences from MRNet

- client-to-client messaging is supported for sessions and services
  - uses min-path through the overlay

- Don’t reinvent the wheel
  - no data formats, just data buffers
    - use existing solutions (e.g., Protobufs or JSON) as you need/want
  - libev for event management
  - dot format used for topology and filter graphs
  - .INI format for session and domain configuration files

🚧 Under Construction 🚧

- 3.5 years of development (working to open source)
- WIP for v1.0
  - SNOstorm launching of additional sessions
  - Service backend launching
  - Dynamic filter configuration
  - Common Communication Interface (ORNL lib for HPC networks)
- To-Do for v1.x
  - libfabric, UCX (anyone interested in helping?)
  - SSL sockets
- To-Do for v2.0
  - Fault tolerance: overlay connection maintenance, domain state replication
  - Integrated key-value store (REDIS?)

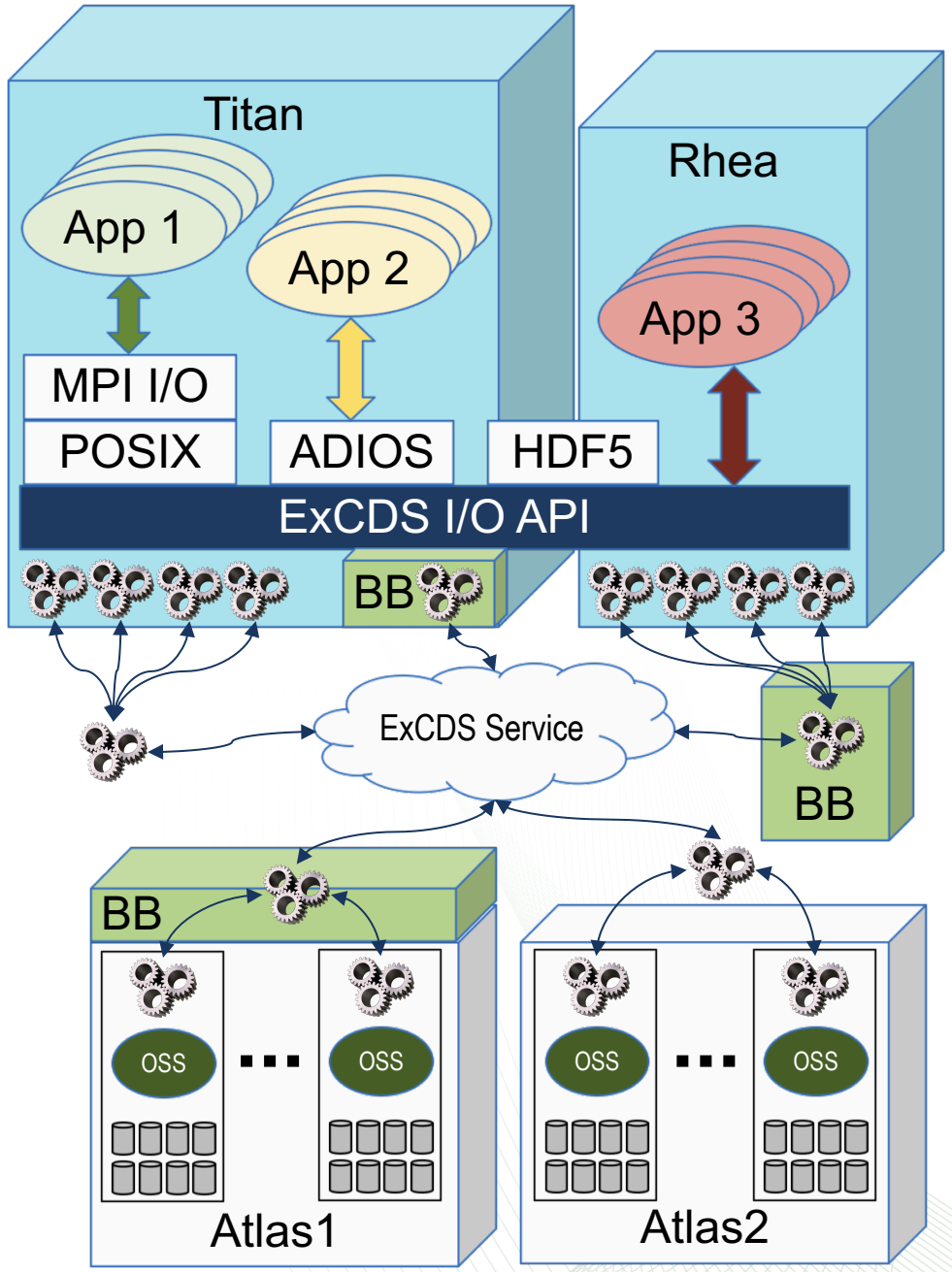
Exascale I/O Research Overview

- **Problem:** users face an increasingly complex storage hierarchy that requires advanced I/O methods to effectively utilize
  - existing application-centric approaches will fail to fully exploit available I/O performance

- **Goal**: create an intelligent system-wide data service that monitors I/O performance and resource contention to inform optimized data placement and I/O operations across storage system tiers
  - for concurrent workloads, including read-intensive analytics
  - applications should not be exposed to the complexity of managing multi-tier data
  - improve I/O performance and productivity for all workloads sharing a multi-tier storage system

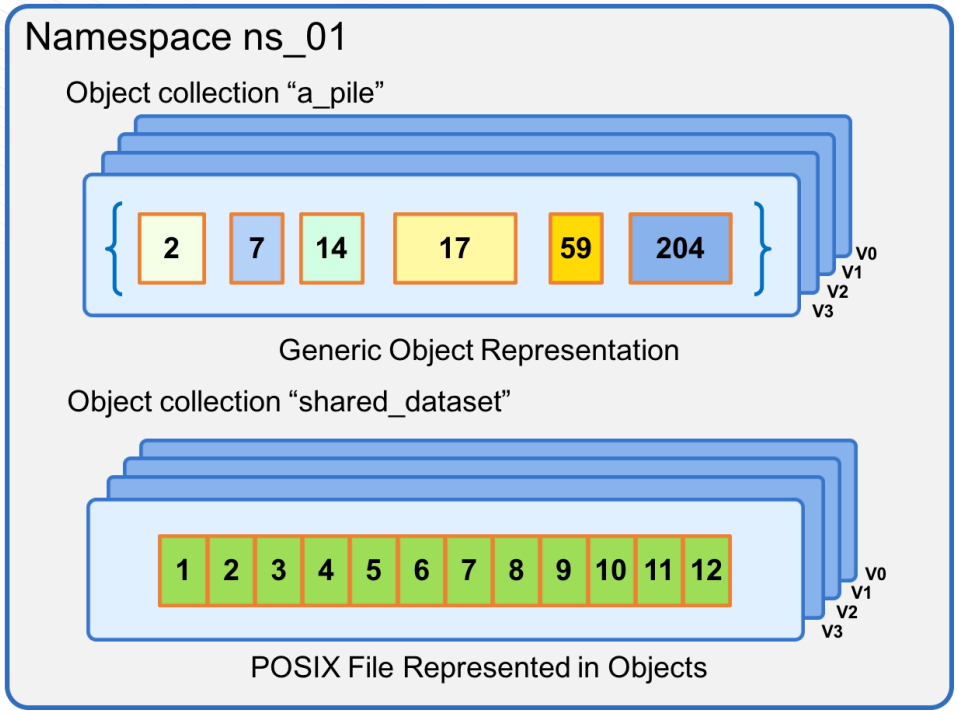
Exascale Computing Data Service (ExCDS) Technical Approach

- Integrate with applications to interpose on I/O requests
- Deploy data service across entire storage hierarchy
  - service process per node
  - connected via SNOflake
- Data service capabilities:
  - monitor host, network, and storage resources
  - control data placement and movement
  - tune BB and PFS behavior
  - aggregate and schedule application I/O requests

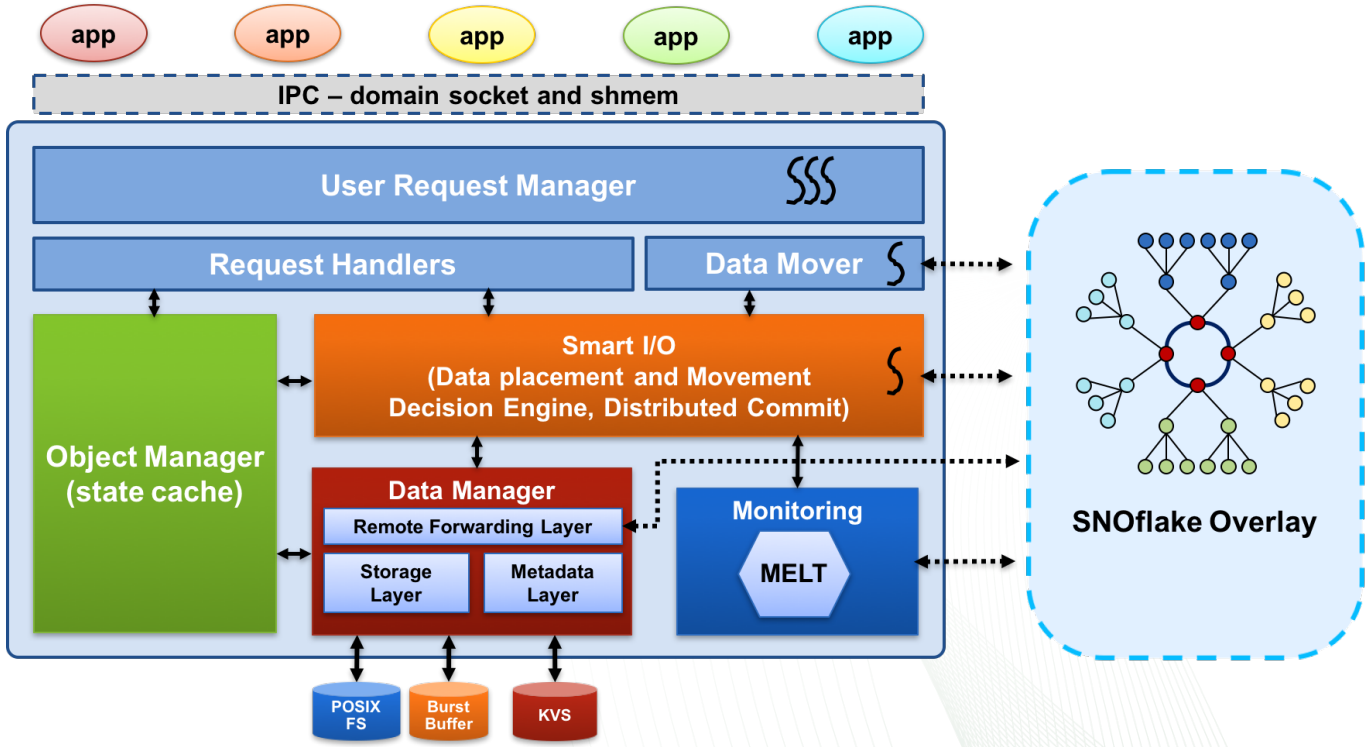


ExCDS Design

Data Abstractions



Service Process Architecture



Storage Monitoring Grand Vision

- Support for center-wide deployments
  - multiple compute and support systems sharing access to one or more Lustre file systems
- Full performance visibility
  - monitor all hosts: clients, MDS, OSS, LNet routers
  - aggregate by: system, job/app, ?project/user?
- Two usage modes
  1. always on, low-overhead monitoring
     - with active problem detection and alerting
  2. on demand, in-depth problem inspection and diagnosis
     - aka “Right Now Queries”

Monitoring Extreme-scale Lustre Toolkit (MELT)

- Collects Lustre performance metrics
  - on clients, OSS, MDS, LNet routers
- Uses SNOflake to:
  - aggregate metric data into performance summaries
    - for clients and LNet routers of each compute cluster
    - for OSS and MDS servers of each storage cluster
  - correlate data within and across compute/storage domains
    - within compute domain: e.g., system-level or job-level aggregation
    - across compute/storage domains: identify server or filesystem contention
- MELT workshop paper: http://arxiv.org/abs/1504.06836

MELT Features Summary

- Metric Data Collection (`meltd_{client,mds_mgs,oss}`)
  - completed: `io`, `meta`, and `load` metric classes
  - future work: `rpc` and `lock`
- Job Info Collection (`meltmon` and `melt_jobmon`)
  - Cray ALPS and SLURM supported
- Metric Data Export (`meltmon`)
  - formats: JSON, XML, and YAML
  - destinations: file, remote process via TCP socket
- Metric Thresholds (`meltmon`)
  - levels: OK, NOTICE, CRITICAL
  - notifications: file, syslog, Nagios check (based on file)
- Command-line Interface (`melt`)
  - work-in-progress: `status` display mode and dynamic collection control
  - future work: `top` display mode

MELT Large-scale Testing

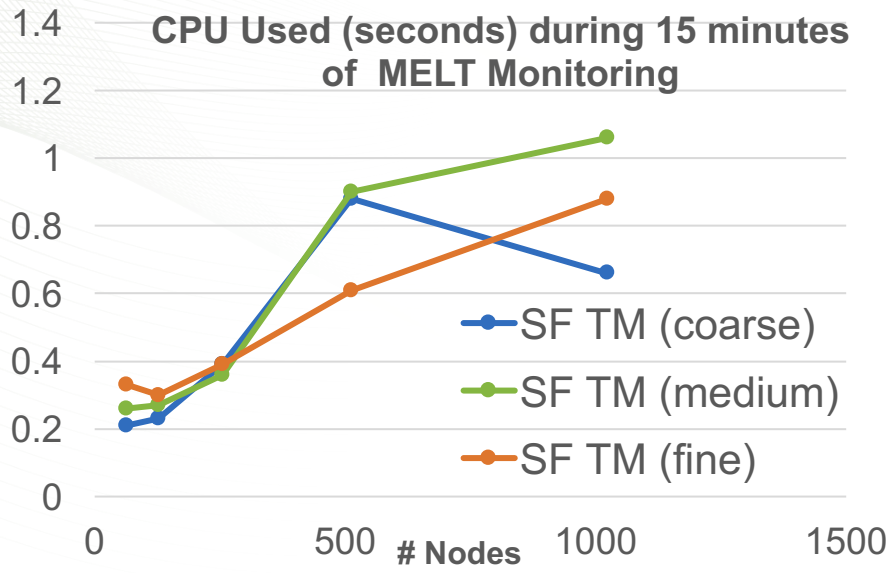
- On Titan Cray XK7 - jobs from 64 to 1024 client nodes
- SNOflake domain topology
  - TM runs on one node, TPs run on every other node
  - `meltd_client` process on each node connects to TM/TP
  - TM has 16 TP children, TPs assigned 32 children until total number of nodes reached
    - 64: `1 → 16 → 47`
    - 128: `1 → 16 → 111`
    - 256: `1 → 16 → 239`
    - 512: `1 → 16 → 495`
    - 1024: `1 → 16 → 512 → 495`

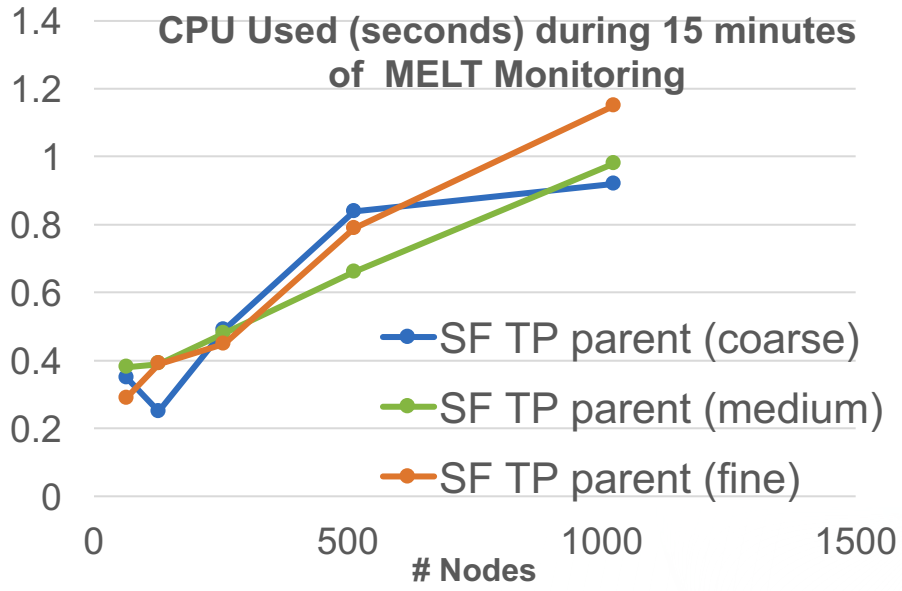
MELT Overhead Analysis

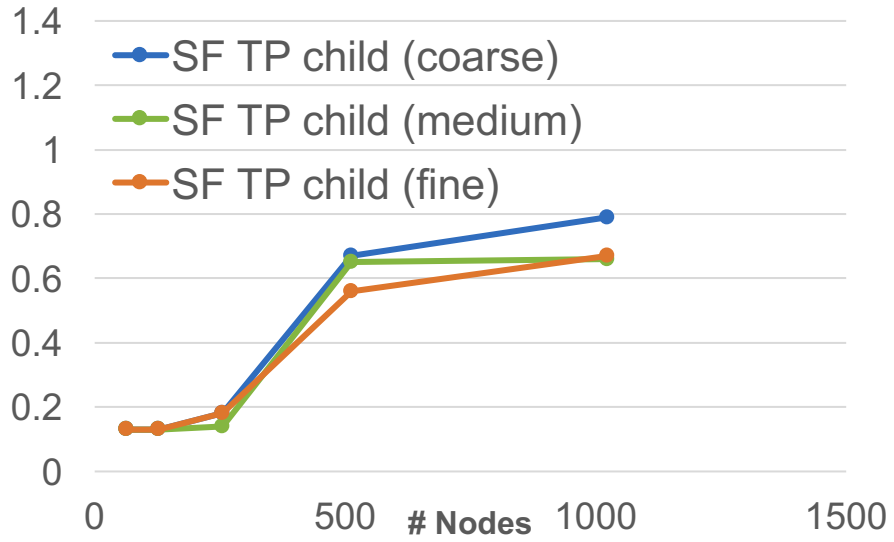
- Study overhead of MELT at large-scale
  - using fixed-time quanta (FTQ) benchmark to measure system noise
  - using different metric data collection intervals (seconds)
    - fine: io@60, meta@120, load@180; gather every 90
    - medium: io@120, meta@240, load@360, gather every 180
    - coarse: io@240, meta@480, load@720, gather every 360

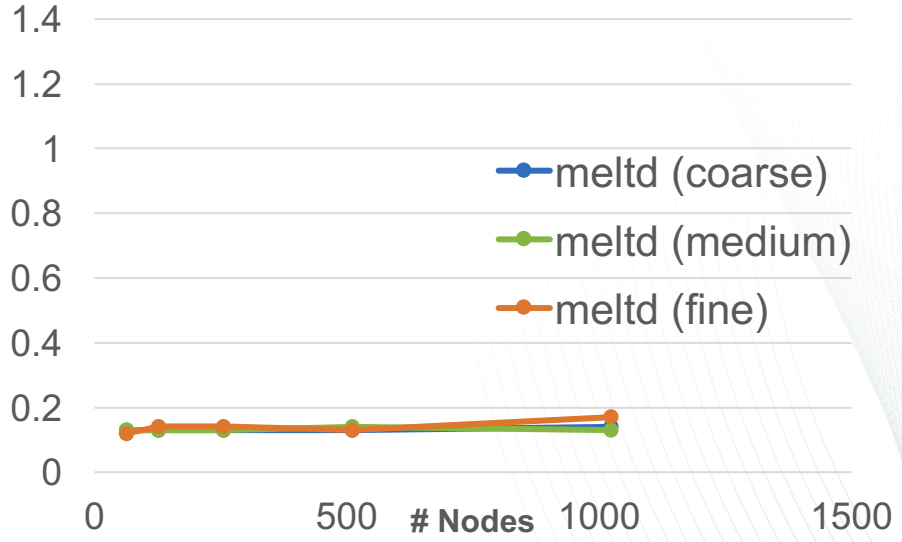
- Ran `top` in batch mode (in background) to capture utilization of SNOflake and MELT processes
  - `top -b -d 180 -u mjbrim`
  - each process also dumps rusage data on normal termination

MELT Overhead Analysis – CPU Utilization

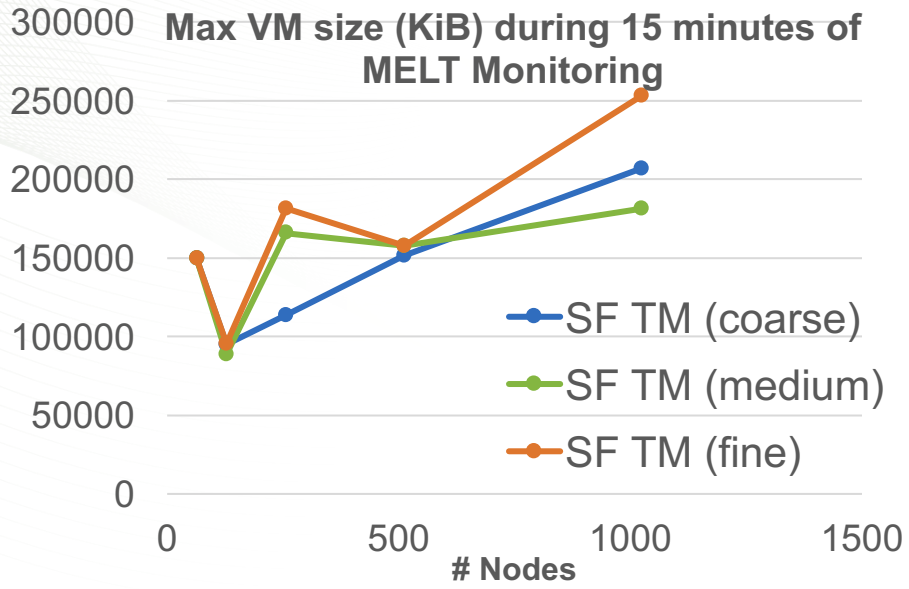


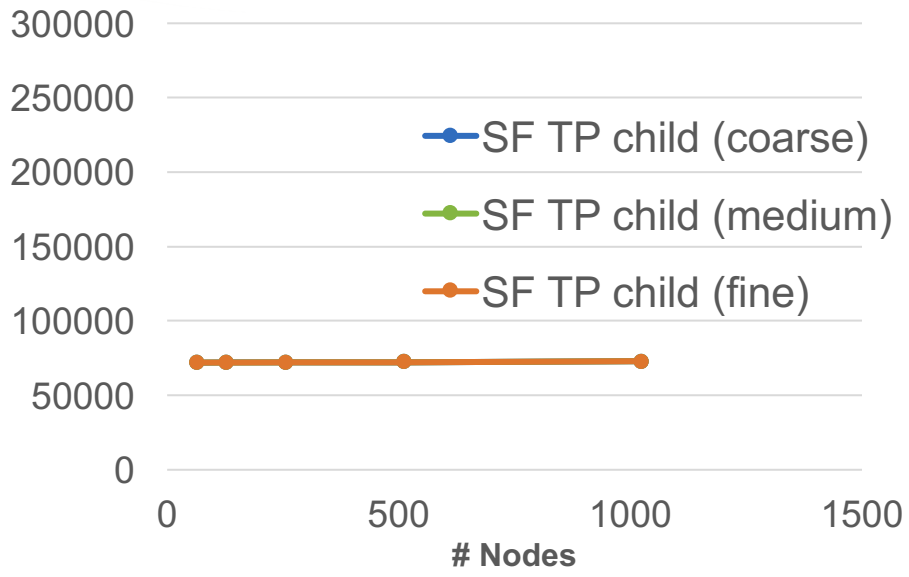


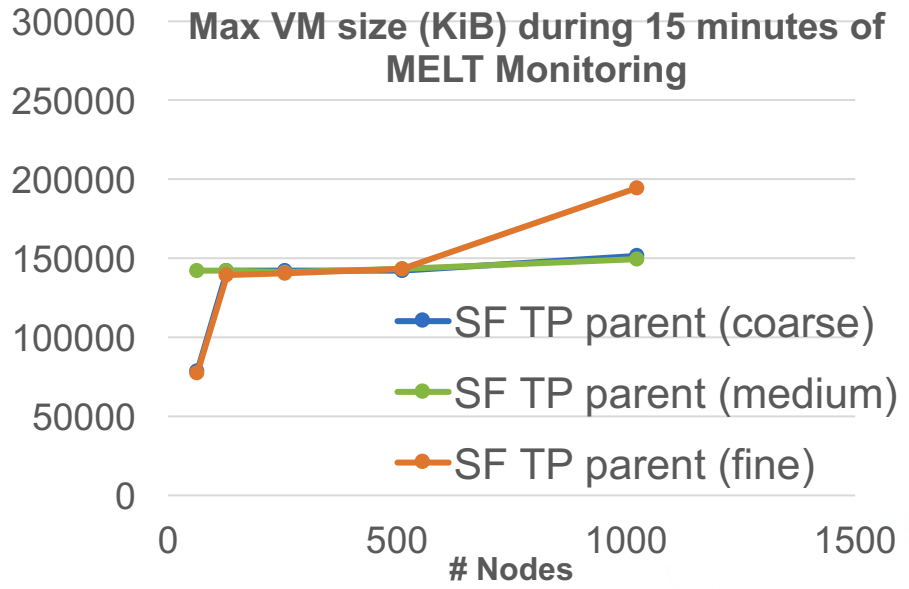


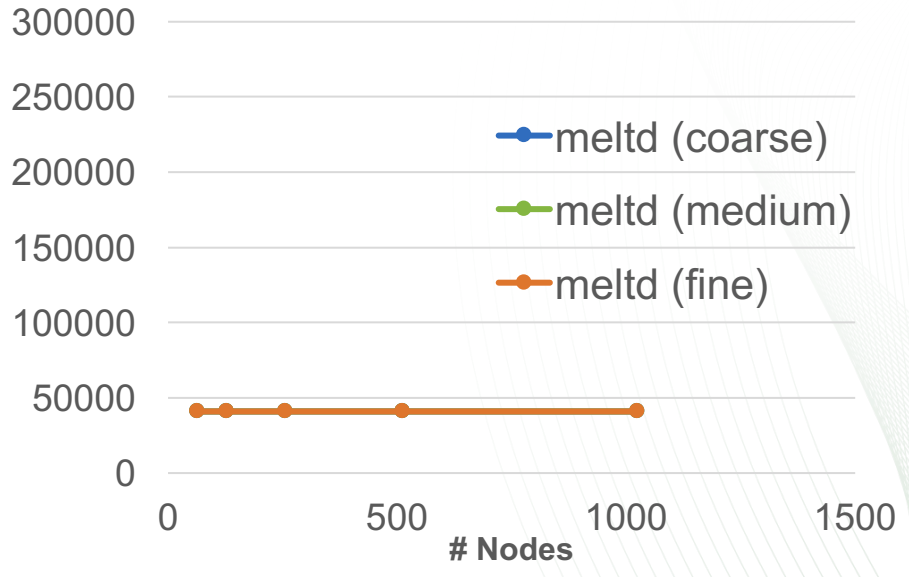


MELT Overhead Analysis – Memory VM

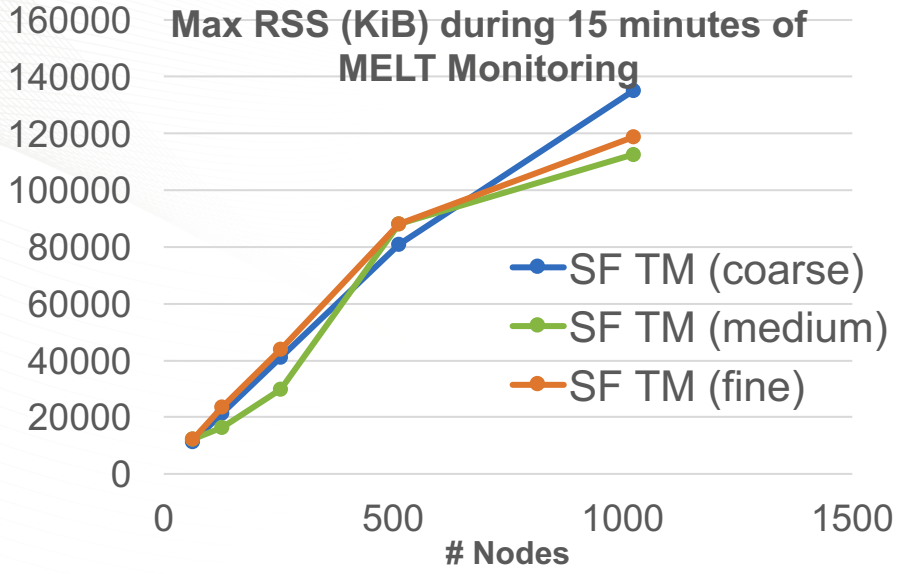


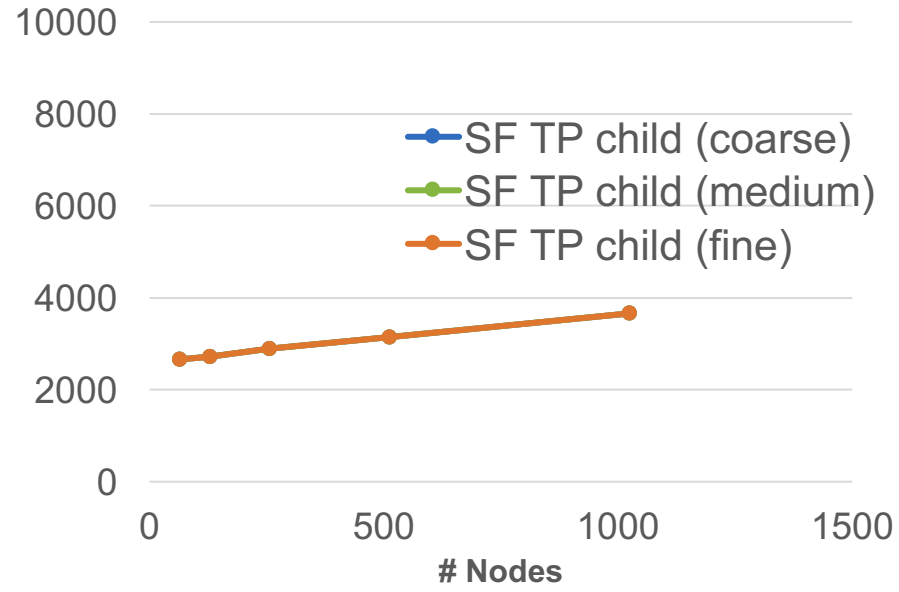


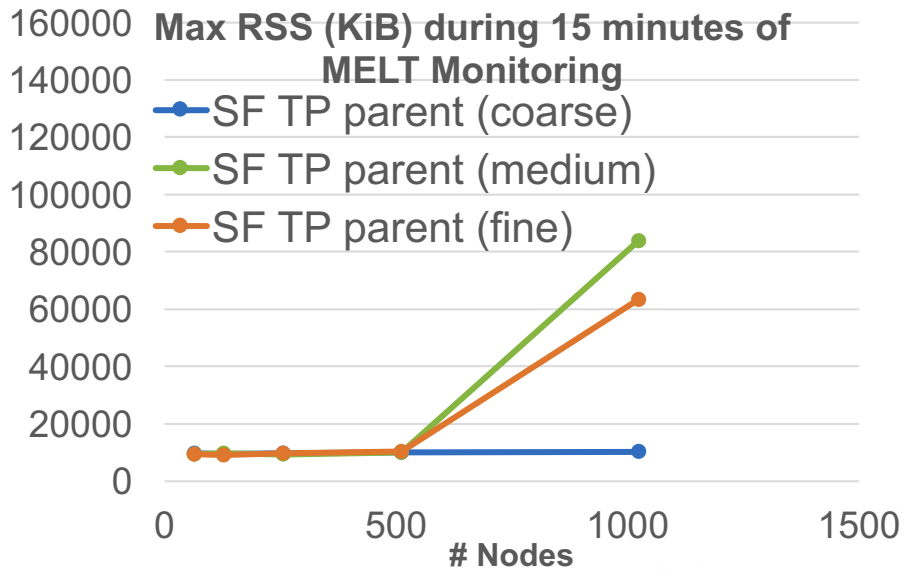


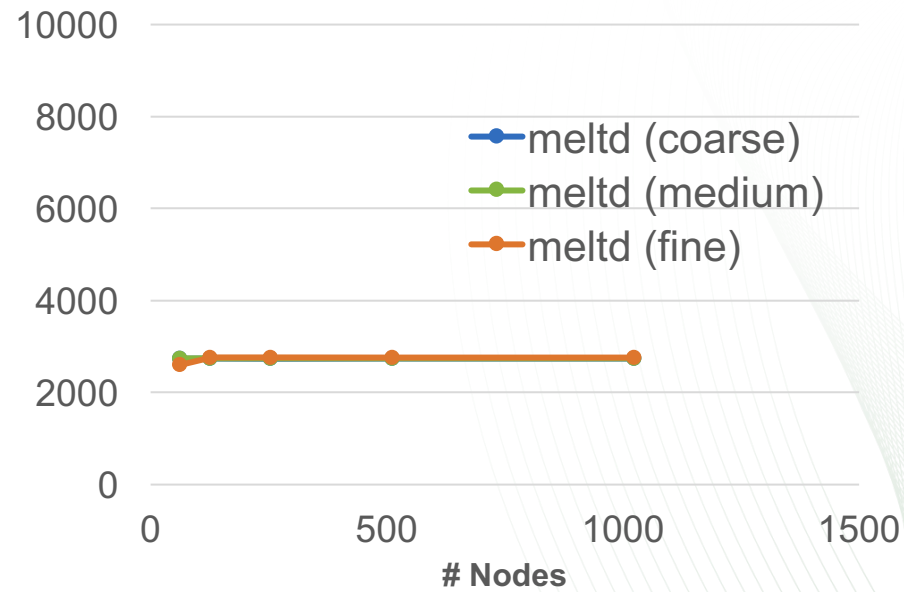


MELT Overhead Analysis – Memory RSS









Acknowledgements

OAK RIDGE National Laboratory | OAK RIDGE LEADERSHIP COMPUTING FACILITY

This work was supported by the Laboratory Directed Research and Development (LDRD) program at Oak Ridge National Laboratory.

This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

This work was supported by the United States Department of Defense (DoD) & used resources of the Computational Research and Development Programs at Oak Ridge National Laboratory.