Modernizing Legacy Codes for Next-Generation Storage Infrastructures

A Case Study of PALM on Intel DAOS

Steffen Christgau

Supercomputing Department
Zuse Institute Berlin

IXPUG Annual Conference
Geneva, Switzerland, September 25, 2019

Motivation: DAOS Software Stack

- approach: applications don't use DAOS API directly → **high level libraries**
- candidates for HPC: netCDF, HDF5, MPI, POSIX (!?)

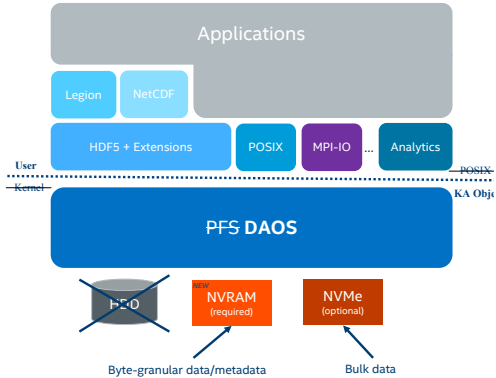
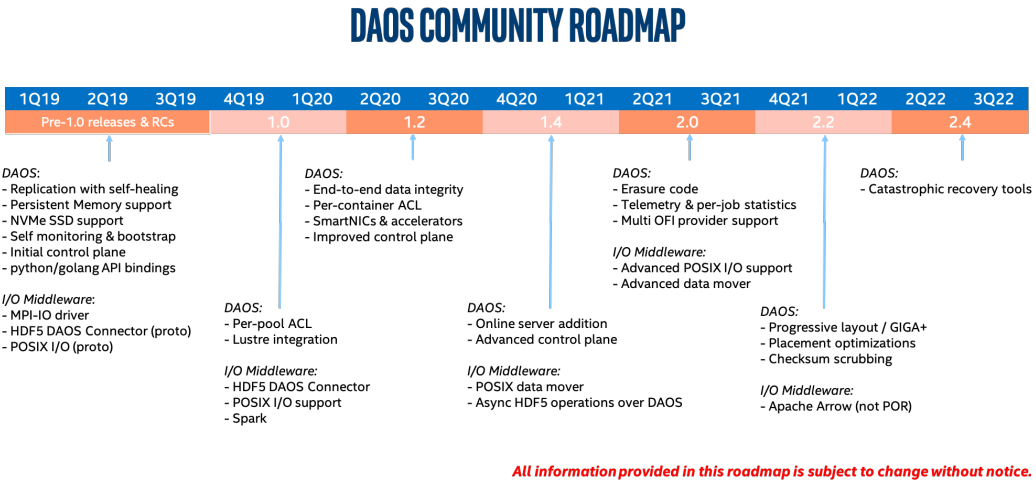


Figure: DAOS Software Stack

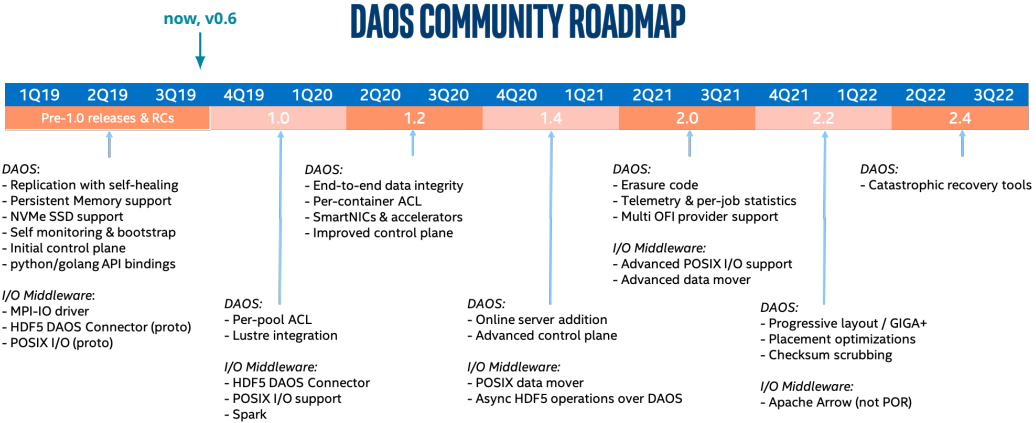
Source: A. Dilger DAOS: Scale-out Object Storage for NVRAM, Dagstuhl Workshop, May 2017

Motivation



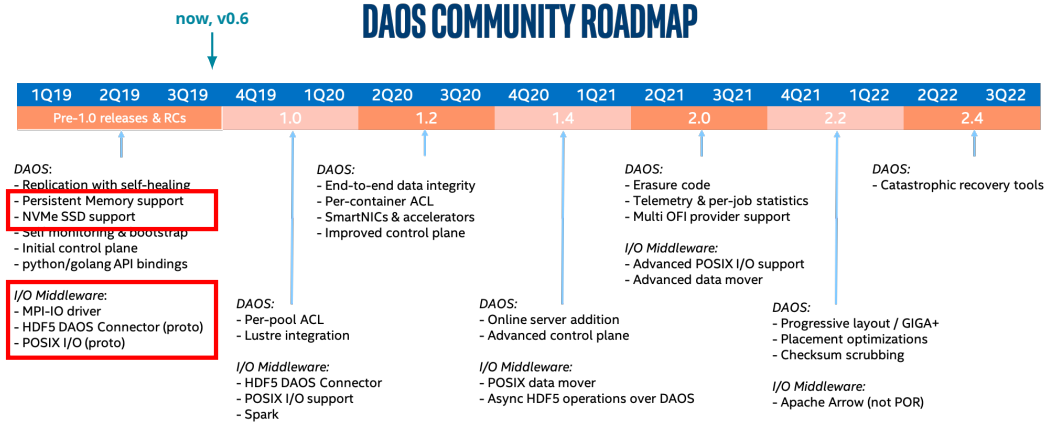
source: `https://wiki.hpdd.intel.com/display/DC/Roadmap?preview=/98342775/105317998/daos_roadmap.png`

Motivation



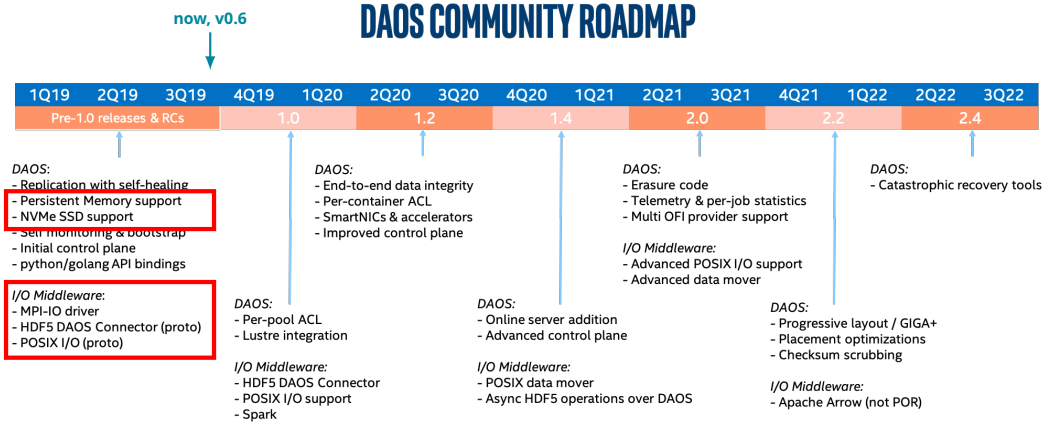
source: `https://wiki.hpdd.intel.com/display/DC/Roadmap?preview=/98342775/105317998/daos_roadmap.png`

Motivation



source: `https://wiki.hpdd.intel.com/display/DC/Roadmap?preview=/98342775/105317998/daos_roadmap.png`

Motivation



source: `https://wiki.hpdd.intel.com/display/DC/Roadmap?preview=/98342775/105317998/daos_roadmap.png`

Outline

PALM Overview

- **Pa**rallelized **L**arge-Eddy Simulation **M**odel by IMUK @ Univ. Hannover
  - computes turbulent air flows, solves imcompressible Navier-Stokes equations
  - 3D compute domain, typical dimensions O(1k) x O(1k) x O(100)
  - since 1997, 200k+ lines of code, lots of modules

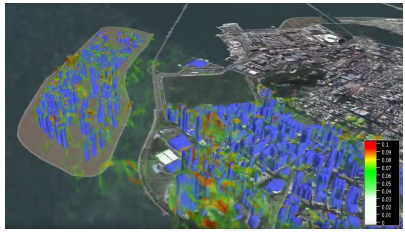


Figure: Turbulent Flow Simulation of Macau (from Knoop, Keck, Raasch: URBAN LES)

PALM Overview

- **Pa**rallelized **L**arge-Eddy Simulation **M**odel by IMUK @ Univ. Hannover
  - computes turbulent air flows, solves imcompressible Navier-Stokes equations
  - 3D compute domain, typical dimensions O(1k) x O(1k) x O(100)
  - since 1997, 200k+ lines of code, lots of modules
  - Fortran 2003 (95) code base, GNU licenced
  - **MPI**/OpenMP parallelization, quite memory hungry
  - **netCDF** for data output

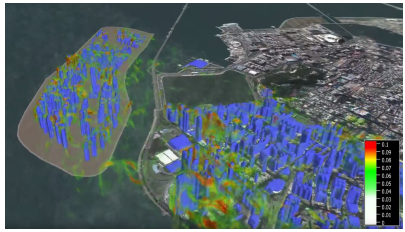
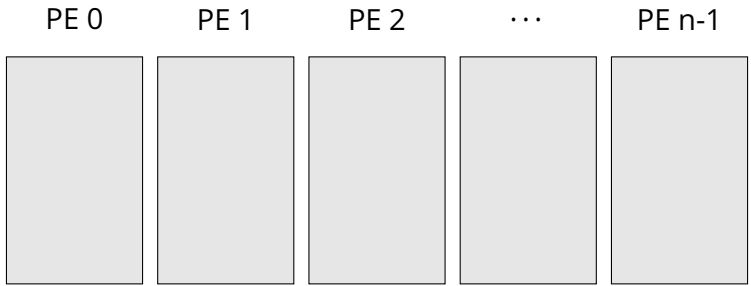


Figure: Turbulent Flow Simulation of Macau (from Knoop, Keck, Raasch: URBAN LES)

Types of Data Distribution

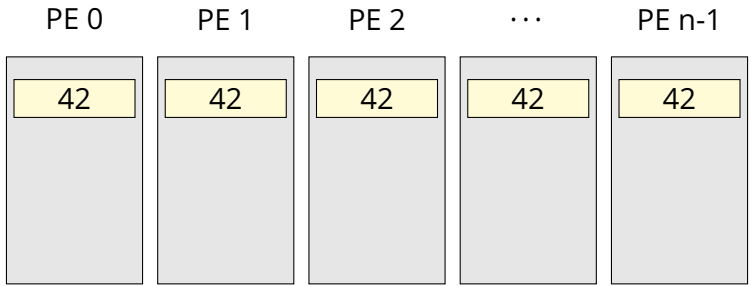
Different distributions for application data



Types of Data Distribution

Different distributions for application data

replicated same shape and data on every PE

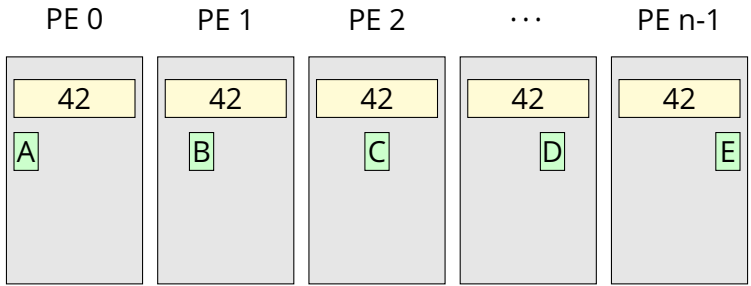


Types of Data Distribution

Different distributions for application data

replicated same shape and data on every PE

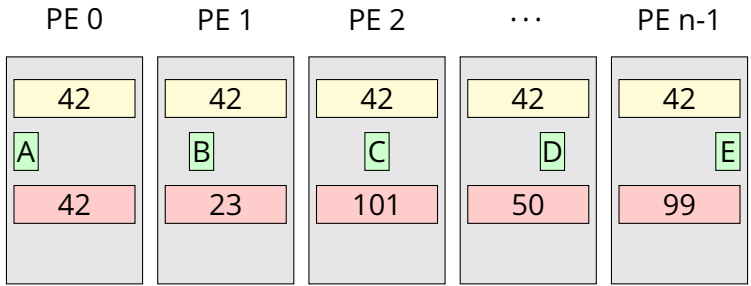
distributed portion of global array, distributed among PEs (compute domain)



Types of Data Distribution

Different distributions for application data

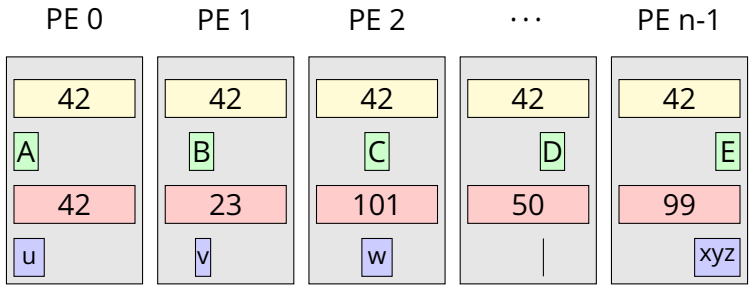
- **replicated** same shape and data on every PE
- **distributed** portion of global array, distributed among PEs (compute domain)
- **individual** same shape, but different data on each PE



Types of Data Distribution

Different distributions for application data

- **replicated** same shape and data on every PE
- **distributed** portion of global array, distributed among PEs (compute domain)
- **individual** same shape, but different data on each PE
- **sgv** scatter/gatherv, like distributed but different shapes, globally indexed



Checkpointing in PALM: Current State

- motivation: save state when job reaches wall time limit, continue at next run

Checkpointing in PALM: Current State

- motivation: save state when job reaches wall time limit, continue at next run
- Fortran unformatted I/O

Checkpointing in PALM: Current State

- motivation: save state when job reaches wall time limit, continue at next run
- Fortran unformatted I/O
- checkpoint creation: write name and raw data to hard-coded unit

```
CALL wrd_write_string( 'topography' )
WRITE ( 14 )  topography
```

Checkpointing in PALM: Current State

- motivation: save state when job reaches wall time limit, continue at next run
- Fortran unformatted I/O
- checkpoint creation: write name and raw data to hard-coded unit

```
CALL wrd_write_string( 'topography' )
WRITE ( 14 )  topography
```

- restore: read name from other hard-coded unit + large select statement

```
SELECT CASE ( restart_string(1:length) )
  ...
  CASE ( 'topography' )
    READ ( 13 )  topography
```

Checkpointing: Challenges

- raw binary data: hard to postprocess (useful for debugging)
- unformatted Fortran IO: "hidden" additional small writes
- one file per process
- PE-independent $\rightarrow$ complicated restore code
- interface to DAOS? Via POSIX!?
- **no abstraction**
- **no expressed parallelism**

Rewrite of PALM's Checkpointing

- new checkpointing abstraction layer

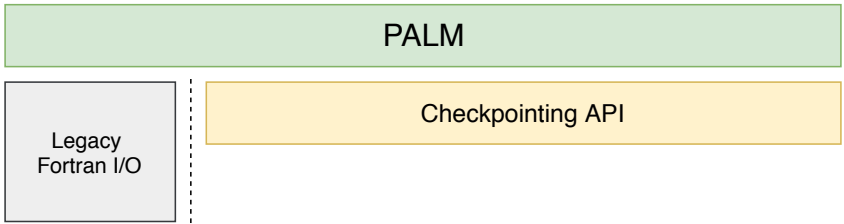
```
!-- optional call to define variable
CALL checkpoint_define('topography', topography, dt_replicated)

CALL checkpoint_write('topography', topography, dt_replicated)
CALL checkpoint_read('topography', topography, dt_replicated)
```

- enables to abstract from actual storage API
  - remove Fortran unformatted IO
  - use of *generic* functions
  - replace storage backend with modern technologies without application changes

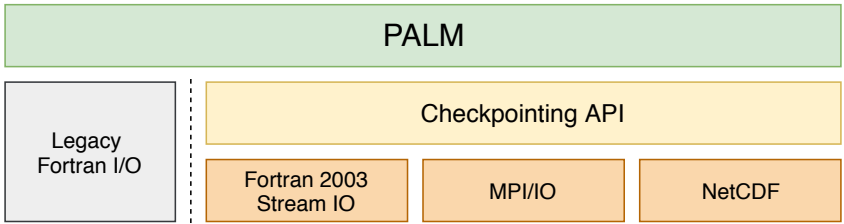
Checkpointing Abstraction Layer

- new abstraction layer for PALM



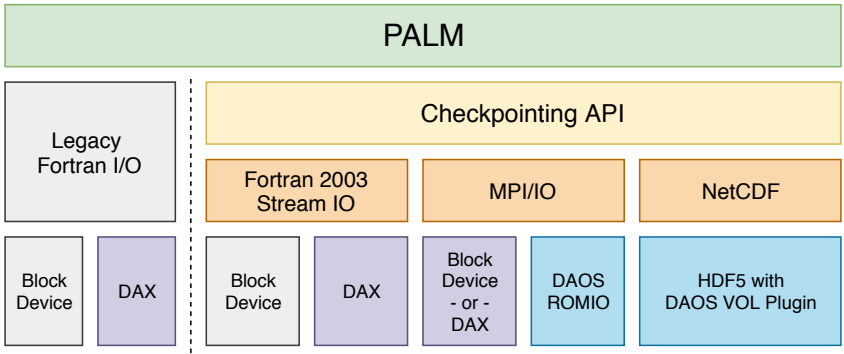
Checkpointing Abstraction Layer

- new abstraction layer for PALM
- access target hardware via **high-level libraries**
  - **ROMIO**: implementation of MPI's IO chapter
  - **netCDF**: IO for named and typed arrays; self-describing files; existing ecosystem



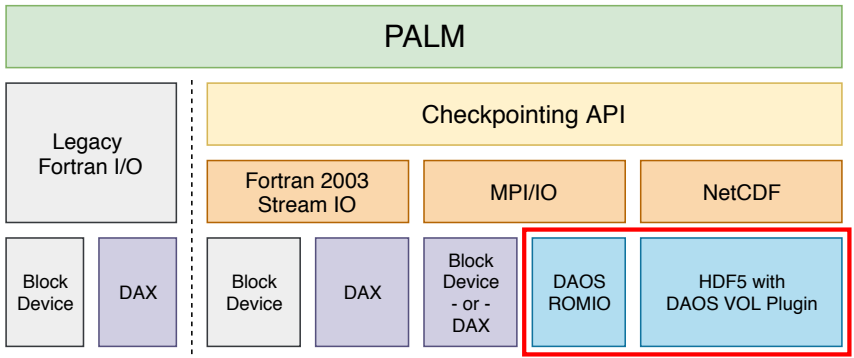
Checkpointing Abstraction Layer

- new abstraction layer for PALM
- access target hardware via **high-level libraries**
  - **ROMIO**: implementation of MPI's IO chapter
  - **netCDF**: IO for named and typed arrays; self-describing files; existing ecosystem
- target hardware: **NVRAM**, via DAX or **DAOS**



Checkpointing Abstraction Layer

- new abstraction layer for PALM
- access target hardware via **high-level libraries**
  - **ROMIO**: implementation of MPI's IO chapter
  - **netCDF**: IO for named and typed arrays; self-describing files; existing ecosystem
- target hardware: **NVRAM**, via DAX or **DAOS**



Backend Details

stream

- writes raw binary data
- offset table and end of file, no PE-independence, PoC

Backend Details

stream

- writes raw binary data
- offset table and end of file, no PE-independence, PoC

MPI/IO

- table for scalar values $\rightarrow$ avoid small writes
- use of MPI derived datatypes for multi-dimensional arrays
- collective write operations where possible, no shared file pointers
- derived from existing prototype code

Backend Details

stream
- writes raw binary data
- offset table and end of file, no PE-independence, PoC

MPI/IO
- table for scalar values $\rightarrow$ avoid small writes
- use of MPI derived datatypes for multi-dimensional arrays
- collective write operations where possible, no shared file pointers
- derived from existing prototype code

netCDF
- use netCDF4 format with parallel MPI/IO (built on top of HDF5)
- easy definition + data access

Backend Details

stream
- writes raw binary data
- offset table and end of file, no PE-independence, PoC

MPI/IO
- table for scalar values → avoid small writes
- use of MPI derived datatypes for multi-dimensional arrays
- collective write operations where possible, no shared file pointers
- derived from existing prototype code

netCDF
- use netCDF4 format with parallel MPI/IO (built on top of HDF5)
- easy definition + data access
- **Danger**: performance pitfall

```
CALL NF90_REDEF( ncid )
CALL NF90_DEFVAR( ncid, label, datatype, id )
CALL NF90_ENDDEF( ncid )
```

Avoid transitions between definition and data mode!
Build batches of variable definitions, write in batch afterwards

Checkpoint Composition

- not all application data written to checkpoint
- typical composition (simplified) for $4096 \times 4096 \times 256$ compute domain

type	count	data (approx)
scalars	192	907 B
1D arrays	57	1.00 GB
2D arrays	21	1.25 GB
3D arrays	9	258.00 GB
4D arrays	1	806.25 KB
total	280	261 GB

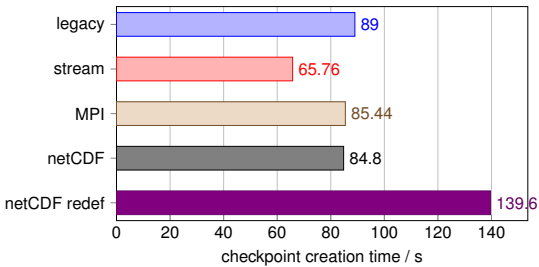
Figure: share of checkpoint size per variable type

Experimental Results: HLRN IV

- evaluate new checkpointing framework on production system
- HLRN IV computer system, Göttingen
- **no DAOS** installation, but **Lustre** file system (4 MDT, 10 OSS, 100 OST)
- used 8 nodes: 2x Xeon Gold SKL 6148, 192 GB RAM, OmniPath, Intel MPI 19u4
- 256 MPI procs (32/node), domain size = $4096 \times 4096 \times 256$

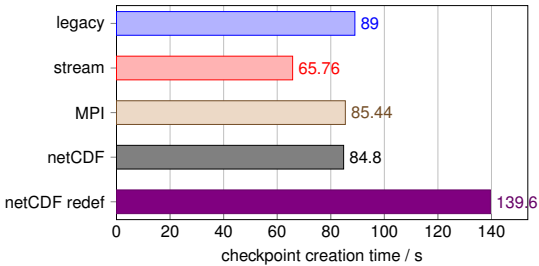
Experimental Results: HLRN IV

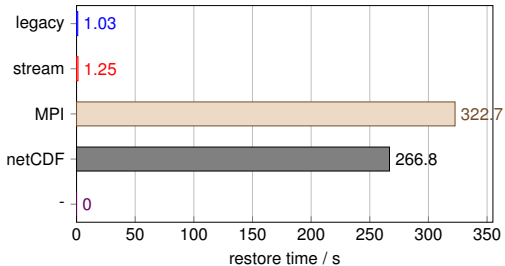
- evaluate new checkpointing framework on production system
- HLRN IV computer system, Göttingen
- **no DAOS** installation, but **Lustre** file system (4 MDT, 10 OSS, 100 OST)
- used 8 nodes: 2x Xeon Gold SKL 6148, 192 GB RAM, OmniPath, Intel MPI 19u4
- 256 MPI procs (32/node), domain size = $4096 \times 4096 \times 256$



Experimental Results: HLRN IV

- evaluate new checkpointing framework on production system
- HLRN IV computer system, Göttingen
- **no DAOS** installation, but **Lustre** file system (4 MDT, 10 OSS, 100 OST)
- used 8 nodes: 2x Xeon Gold SKL 6148, 192 GB RAM, OmniPath, Intel MPI 19u4
- 256 MPI procs (32/node), domain size = $4096 \times 4096 \times 256$





VTune Profile for Restore on HLRN IV

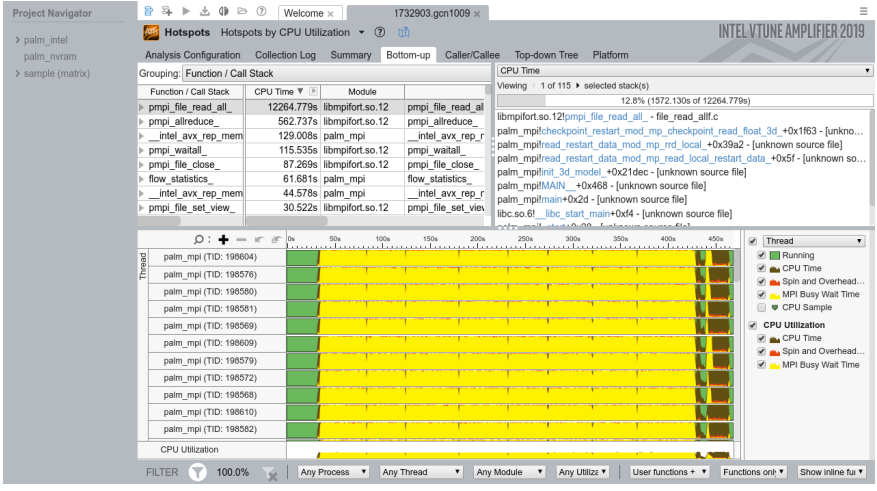


Figure: Profile of MPI backend performance for restore from checkpoint

VTune Profile for Restore on HLRN IV

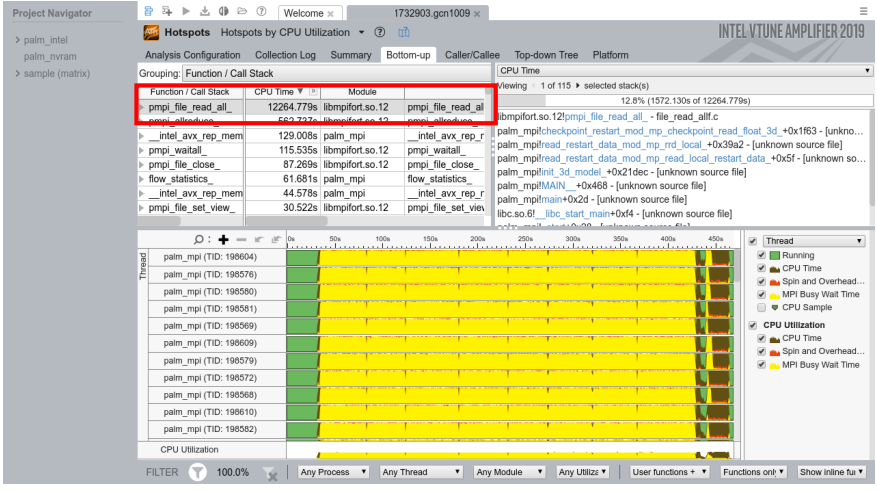


Figure: Profile of MPI backend performance for restore from checkpoint

VTune Profile for Restore on HLRN IV

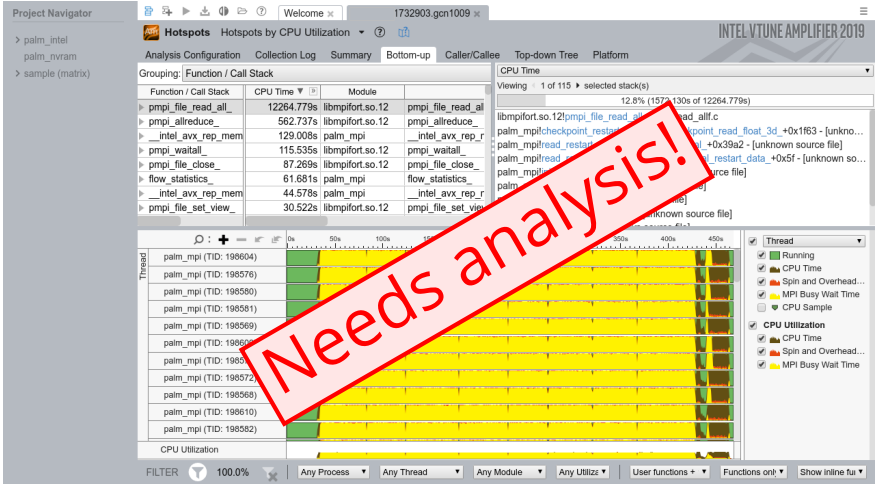


Figure: Profile of MPI backend performance for restore from checkpoint

Using NVRAM and DAOS

motivation recap: use high level libraries to interface DAOS

Using NVRAM and DAOS

motivation recap: use high level libraries to interface DAOS

stream

- **no interface to DAOS**, maybe FUSE $\rightarrow$ not investigated
- use NVRAM via **DAX-mounted file system**

Using NVRAM and DAOS

motivation recap: use high level libraries to interface DAOS

stream

- **no interface to DAOS**, maybe FUSE → not investigated
- use NVRAM via **DAX-mounted file system**

MPI/IO

- **interfaces DAOS'** low level API
- ROMIO implementation from DAOS team, based on MPICH 3.3
- `configure`, `make`, `make install` and **just run your MPI IO code**
- some intended limitations; do not apply to PALM
- alternative: use **DAX-mounted file system**

Using NVRAM and DAOS

motivation recap: use high level libraries to interface DAOS

stream

- **no interface to DAOS**, maybe FUSE → not investigated
- use NVRAM via **DAX-mounted file system**

MPI/IO

- **interfaces DAOS'** low level API
- ROMIO implementation from DAOS team, based on MPICH 3.3
- `configure`, `make`, `make install` and **just run your MPI IO code**
- some intended limitations; do not apply to PALM
- alternative: use **DAX-mounted file system**

netCDF

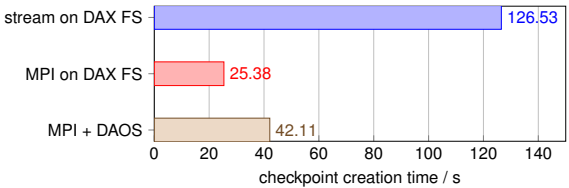
- our assumption: just built netCDF on top of DAOS-aware HDF5
- HDF5 VOL plugin for DAOS is work in progress, said to become stable this year, netCDF likely to need adjustments
- **DAOS currenty not useable via netCDF**

Experimental Results: DAOS Testbed

- single node:
  - dual Xeon Platinum 8260L (CL-SP, 24C/48T)
  - 6 TB Apache Pass (2 $\times$ 6 DCPMM), 768 GB DRAM
  - CentOS 7, gcc/gfortran 9.1
  - 32 MPI procs, domain size = 4096 $\times$ 4096 $\times$ 256
- mimics "DAOS on every compute node" scenario (vs burst buffer-like setup)
- compare backend stream, MPI on DAX FS, and MPI on top of DAOS

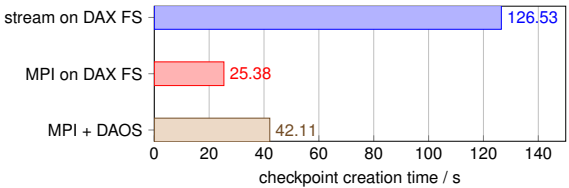
Experimental Results: DAOS Testbed

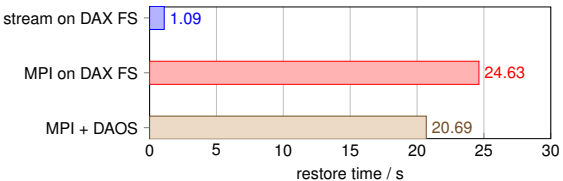
- single node:
  - dual Xeon Platinum 8260L (CL-SP, 24C/48T)
  - 6 TB Apache Pass (2 $\times$ 6 DCPMM), 768 GB DRAM
  - CentOS 7, gcc/gfortran 9.1
  - 32 MPI procs, domain size = 4096 $\times$ 4096 $\times$ 256
- mimics "DAOS on every compute node" scenario (vs burst buffer-like setup)
- compare backend stream, MPI on DAX FS, and MPI on top of DAOS



Experimental Results: DAOS Testbed

- single node:
  - dual Xeon Platinum 8260L (CL-SP, 24C/48T)
  - 6 TB Apache Pass (2 $\times$ 6 DCPMM), 768 GB DRAM
  - CentOS 7, gcc/gfortran 9.1
  - 32 MPI procs, domain size = 4096 $\times$ 4096 $\times$ 256
- mimics "DAOS on every compute node" scenario (vs burst buffer-like setup)
- compare backend stream, MPI on DAX FS, and MPI on top of DAOS





Experiences with using DAOS (v0.6)

- Installation:
  - scons

Experiences with using DAOS (v0.6)

- Installation:
  - scons
  - use "download dependencies" feature
  - no packages,. . . yet

Experiences with using DAOS (v0.6)

- Installation:
  - scons
  - use "download dependencies" feature
  - no packages,. . . yet
- Setup:
  - currently cannot use all DCPMMs on NUMA system
  - sometimes confusing configuration:
    - values for unimplemented features in examples
    - leave defaults where unsure
    - sufficient to define what contributes to pool.
  - notice *immutable* notes

Experiences with using DAOS (v0.6)

- Installation:
  - scons
  - use "download dependencies" feature
  - no packages,. . . yet
- Setup:
  - currently cannot use all DCPMMs on NUMA system
  - sometimes confusing configuration:
    - values for unimplemented features in examples
    - leave defaults where unsure
    - sufficient to define what contributes to pool.
  - notice *immutable* notes
- Usage:
  - permissions: required to use DAOS account to get access to pool
  - unmature tools: documented commands return *not implemented*
  - (1+x) threads per target, i.e. DCPMM, hog CPU cores $\rightarrow$ energy consumption!?
  - `first_core` option helpful

Summary

Summary

- Avoid raw (POSIX) IO, although it seems to be easy.
- Use established high-level libraries.

Summary

- Avoid raw (POSIX) IO, although it seems to be easy.
- Use established high-level libraries.
- DAOS is under heavy development! Expect some trouble, give feedback!

Summary

- Avoid raw (POSIX) IO, although it seems to be easy.
- Use established high-level libraries.
- DAOS is under heavy development! Expect some trouble, give feedback!
- Early DAOS performance numbers promising, but room for improvements.

Summary

- Avoid raw (POSIX) IO, although it seems to be easy.
- Use established high-level libraries.
- DAOS is under heavy development! Expect some trouble, give feedback!
- Early DAOS performance numbers promising, but room for improvements.

```
CALL NF90_CLOSE(presentation)
```

Questions!?