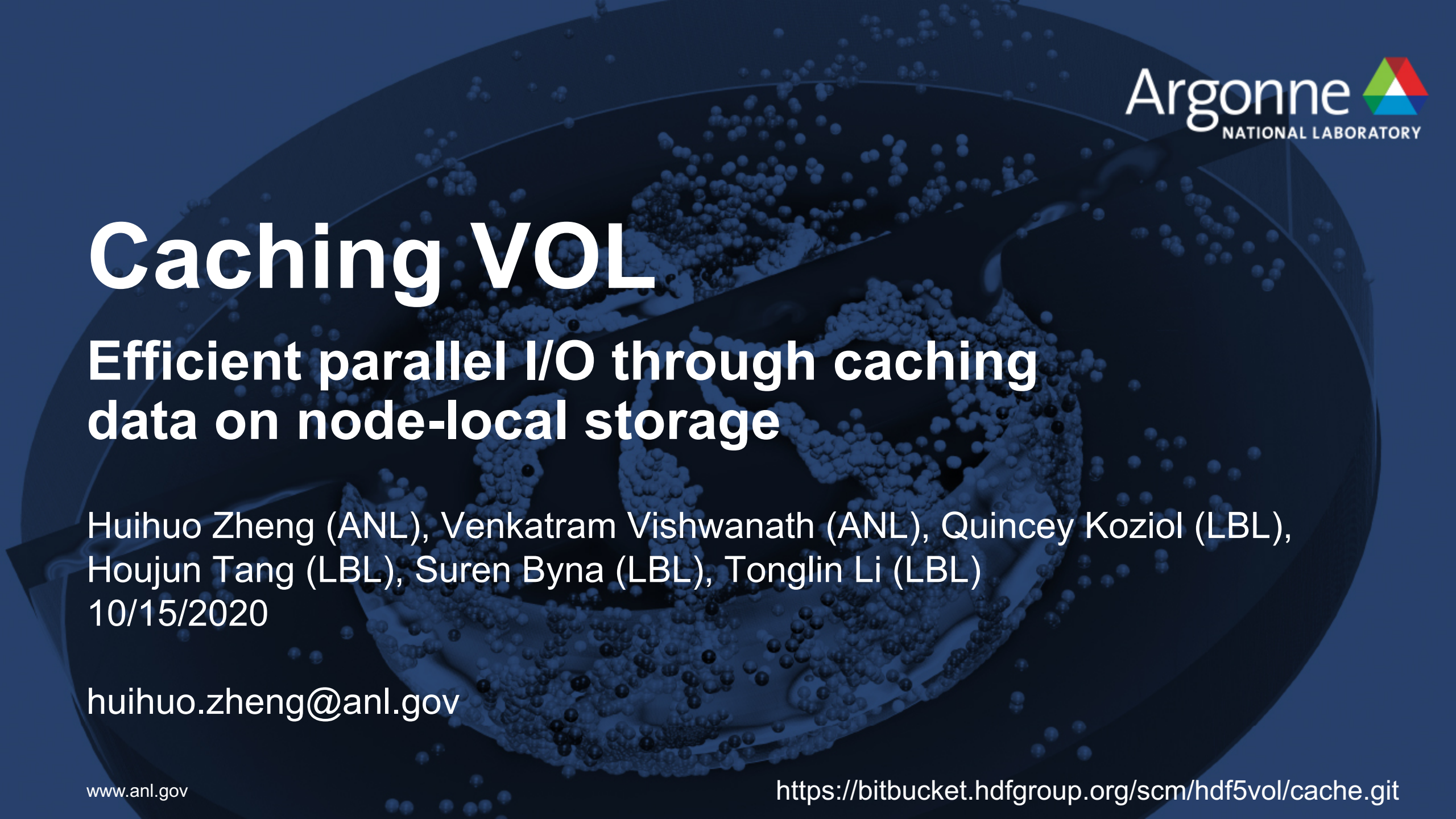


Caching VOL

Efficient parallel I/O through caching data on node-local storage

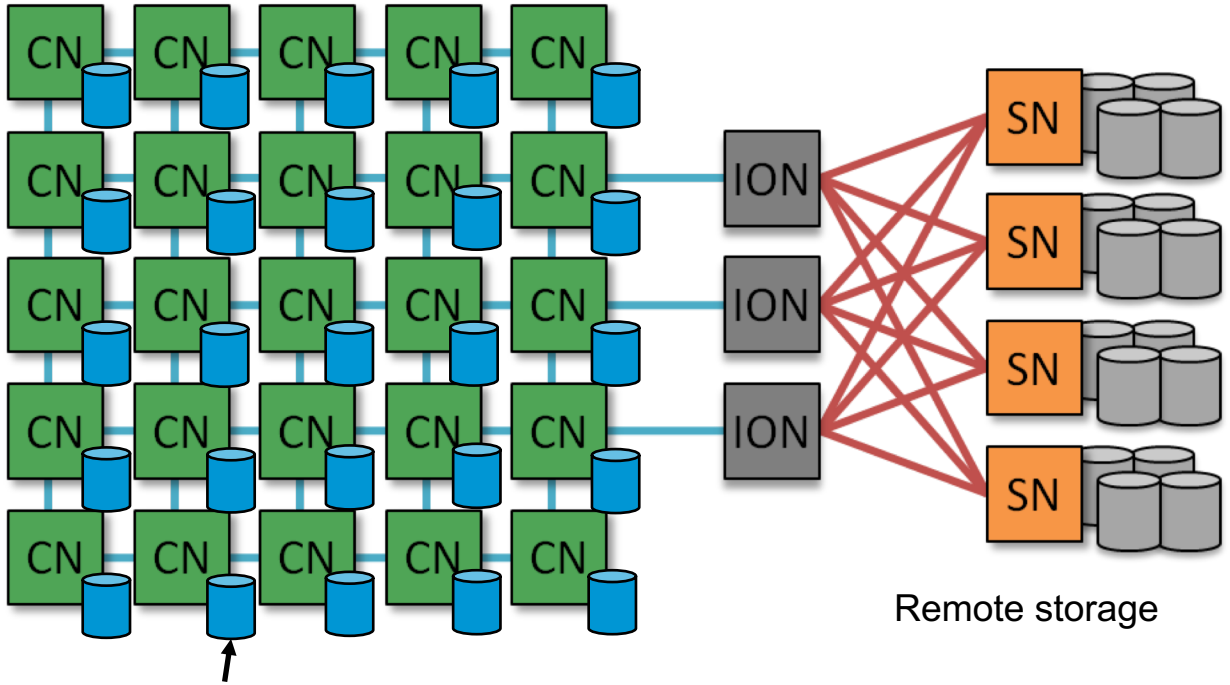
Huihuo Zheng (ANL), Venkatram Vishwanath (ANL), Quincey Koziol (LBL), Houjun Tang (LBL), Suren Byna (LBL), Tonglin Li (LBL)

10/15/2020

huihuo.zheng@anl.gov

https://bitbucket.hdfgroup.org/scm/hdf5vol/cache.git

Integrating node-local storage into parallel I/O workflow



Typical HPC storage hierarchy: node-local storage (NLS) + global parallel file system (PFS)

Theta @ ALCF: Lustre + SSD (128 GB / node),
ThetaGPU (DGX-3) @ ALCF: NVMe (15.4 TB / node)
Summit @ OLCF: GPFS + NVMe (1.6 TB / node)

Node-local storage

- Local to the compute node, does not need to go through the network
- Larger aggregate bandwidth compared to the parallel file systems

Theta (w) – Lustre: 200 GB/s, SSD: 3TB/s
Summit (w) – GPFS: 2.5 TB/s, NVMe: 9.7 TB/s

Challenges

- Distributed
- Accessible only during job running

Typical usage

- Temporal storing data of the compute node

Out goal: using node-local storage for caching / staging data to improve the parallel I/O

How to use the caching VOL

1) Inserting compute work between write/read and close.

```
H5Dopen()
H5Dread()
…# compute
H5Dclose()
```

```
H5Dcreate()
H5Dwrite()
… # compute
H5Dclose()
```

MPI_Init_thread(…, MPI_THREAD_MULTIPLE…)

Same public HDF5 API

Compatible with h5py 2.10.0

2) Setting VOL path

```
export HDF5_PLUGIN_PATH=$HDF5_ROOT/../vol/lib
export HDF5_VOL_CONNECTOR="cache_ext under_vol=0;under_info={};"
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HDF5_PLUGIN_PATH
```

3) Enabling caching VOL

Opt. 1 Through global environmental variable

```
export HDF5_CACHE_RD=yes
export
HDF5_LOCAL_STORAGE_PATH=/local/scratch
export HDF5_LOCAL_STORAGE_TYPE=SSD
```

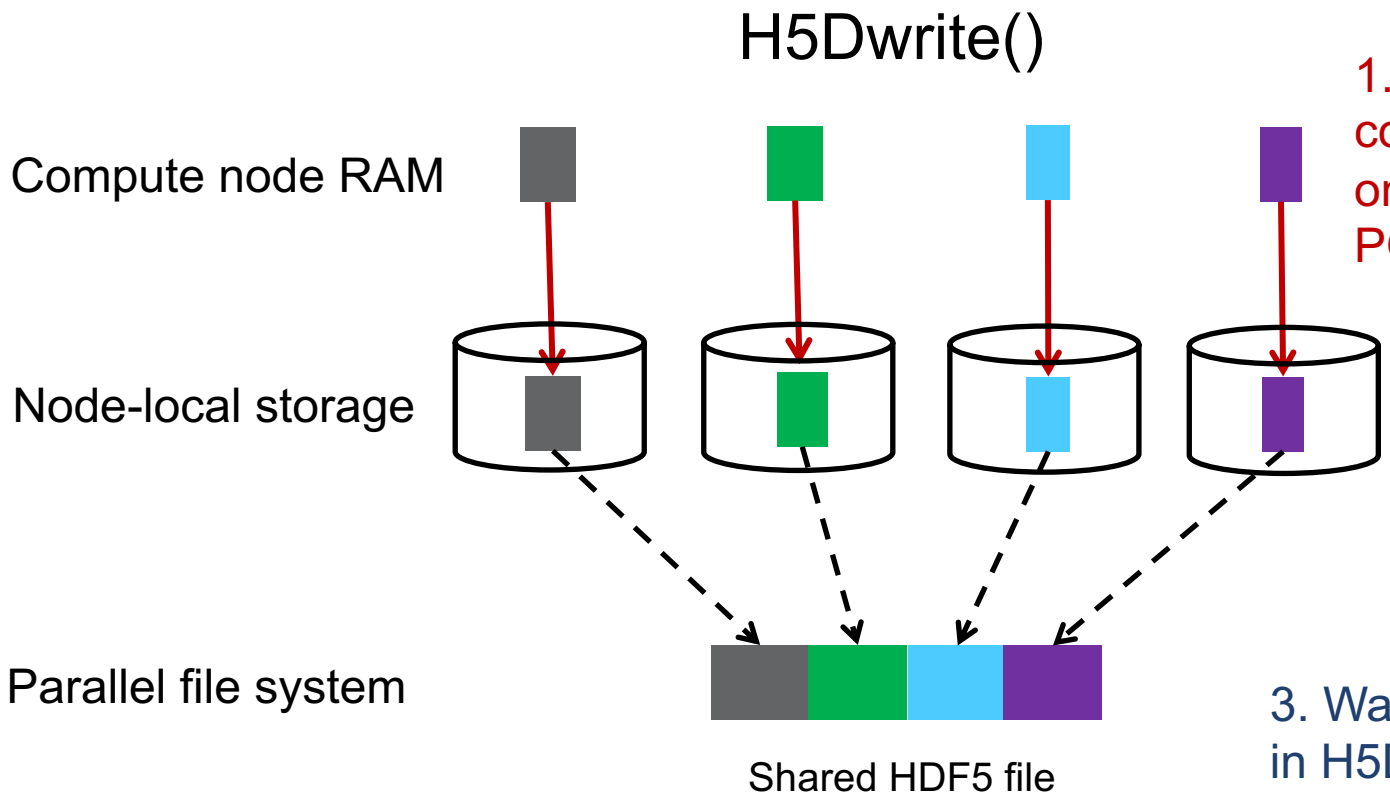
Opt. 2 Through explicit APIs

```
H5Pset_fapl_plist('HDF5_CACHE_RD', true)
…
H5Fcreate_cache()
H5Dcreate_cache()
```


Repository: https://bitbucket.hdfgroup.org/scm/hdf5vol/cache.git

Cache VOL Design Details

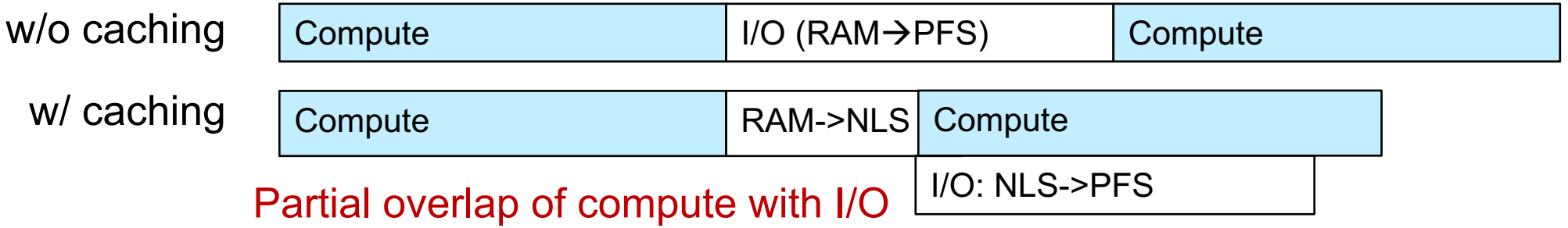
Parallel Write



1. Buffer is synchronously copied to memory mapped files on the node-local storage using POSIX I/O.

2. Move data from memory mapped file to the parallel file system asynchronously using background thread with HDF5 dataset write function (***not sacrificing extra memory***)

3. Wait for all the tasks to finish in H5Dclose() / H5Fclose()



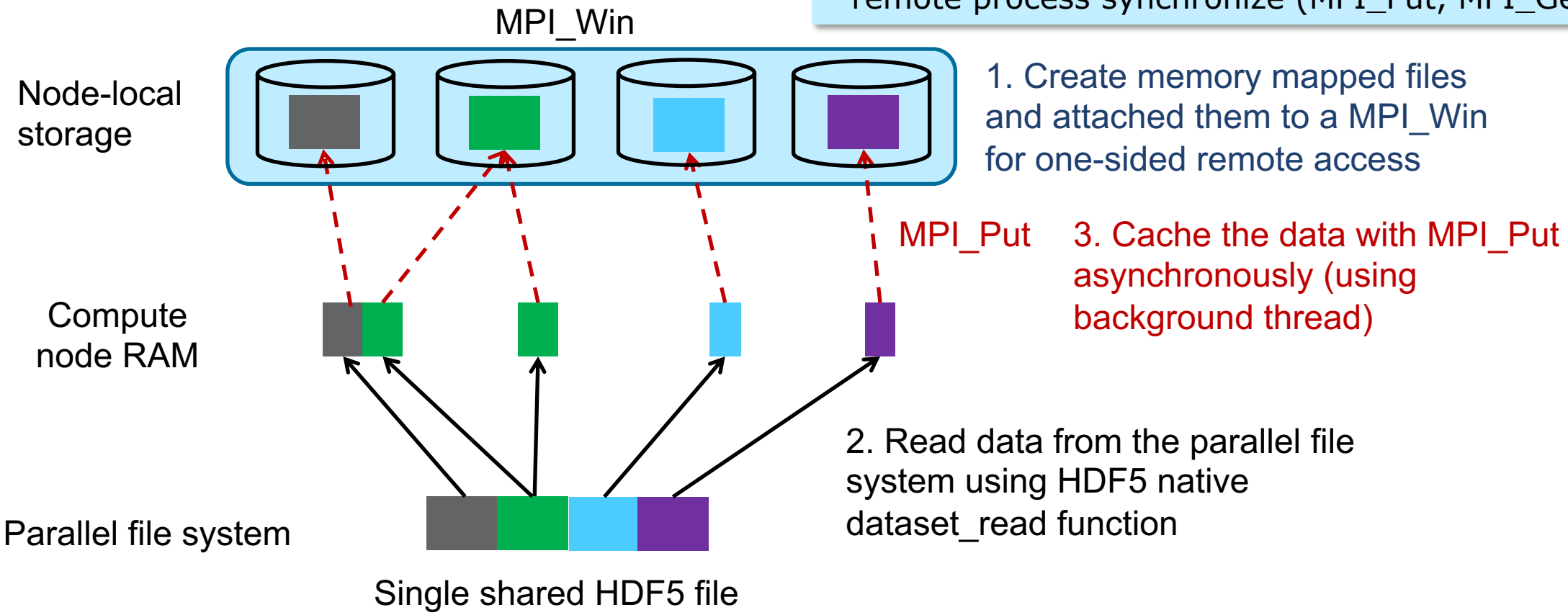
Partial overlap of compute with I/O

Details are hidden from the application developers.

Parallel Read

First iteration: on-the-fly prefetching data

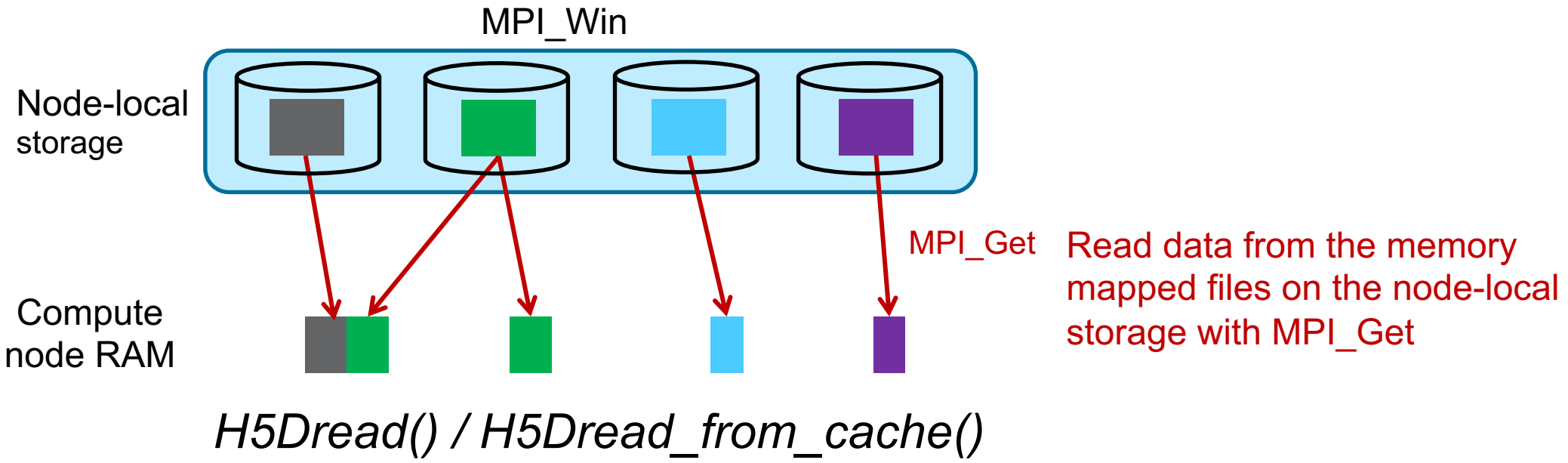
One-sided communication
- Each process exposes a part of its memory to other processes (MPI Window)
- Other processes can directly read from or write to this memory, without requiring that the remote process synchronize (MPI_Put, MPI_Get)



H5Dread() / H5Dread_to_cache / H5Dprefetch

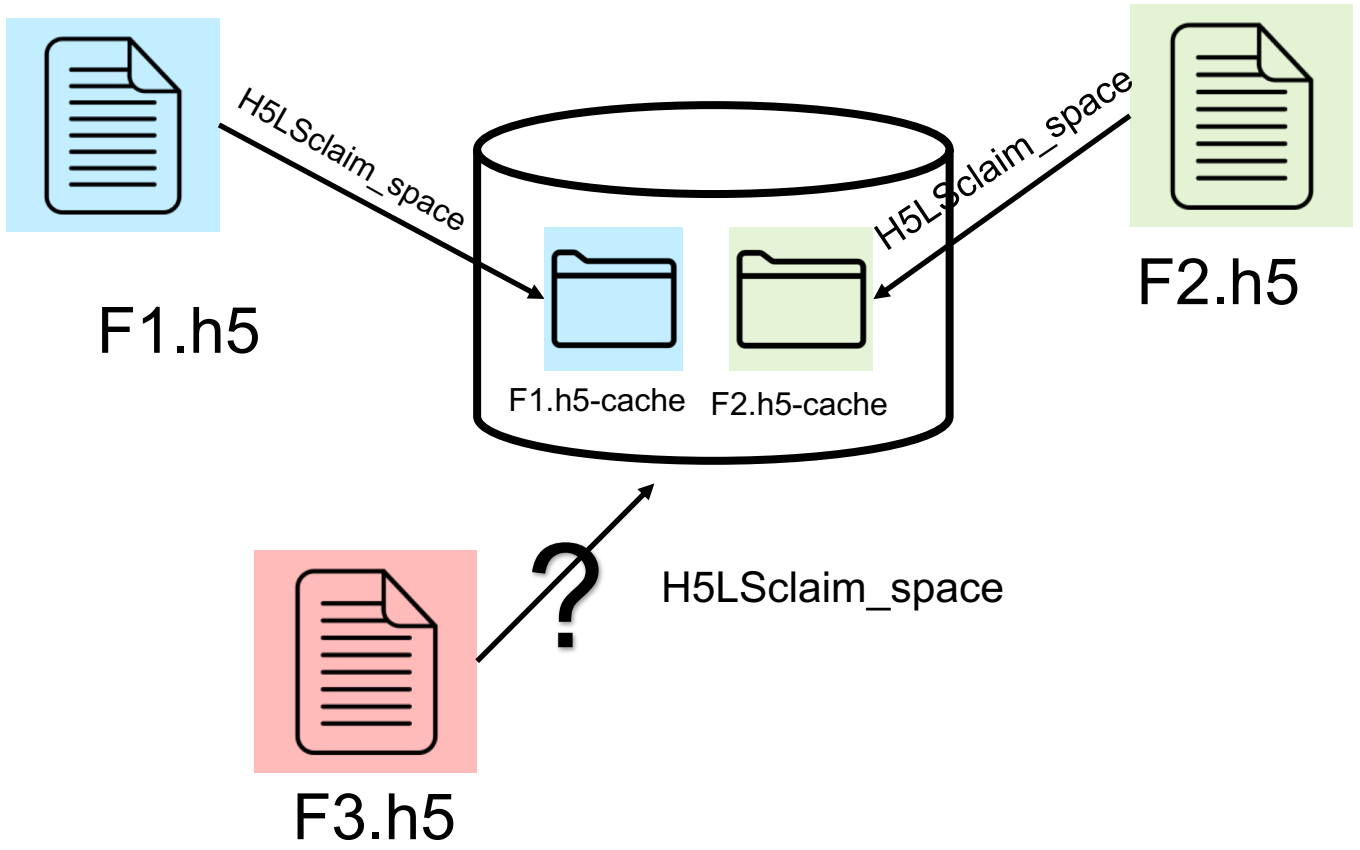
Parallel Read

Reading data to the node-local storage



H5Dread() / H5Dread_from_cache()

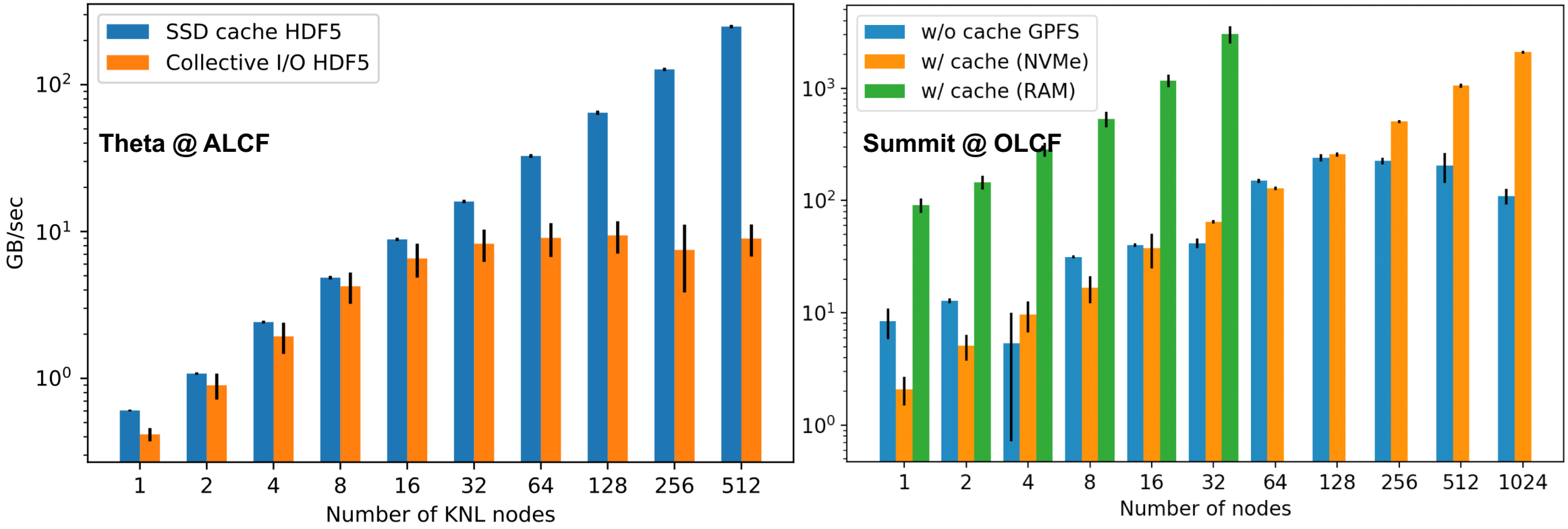
Storage management: cache replacement policy



- Each file claims a certain space on the node-local storage. If successful, a folder is created to contain the cached data.
- If the space is full, free up space for the new file based on certain cache replacement policy (LRU, FIFO, LFU). If not able to free up enough space, no caching will be turned on for that file.
- Cache is removed at H5Dclose / H5Fclose

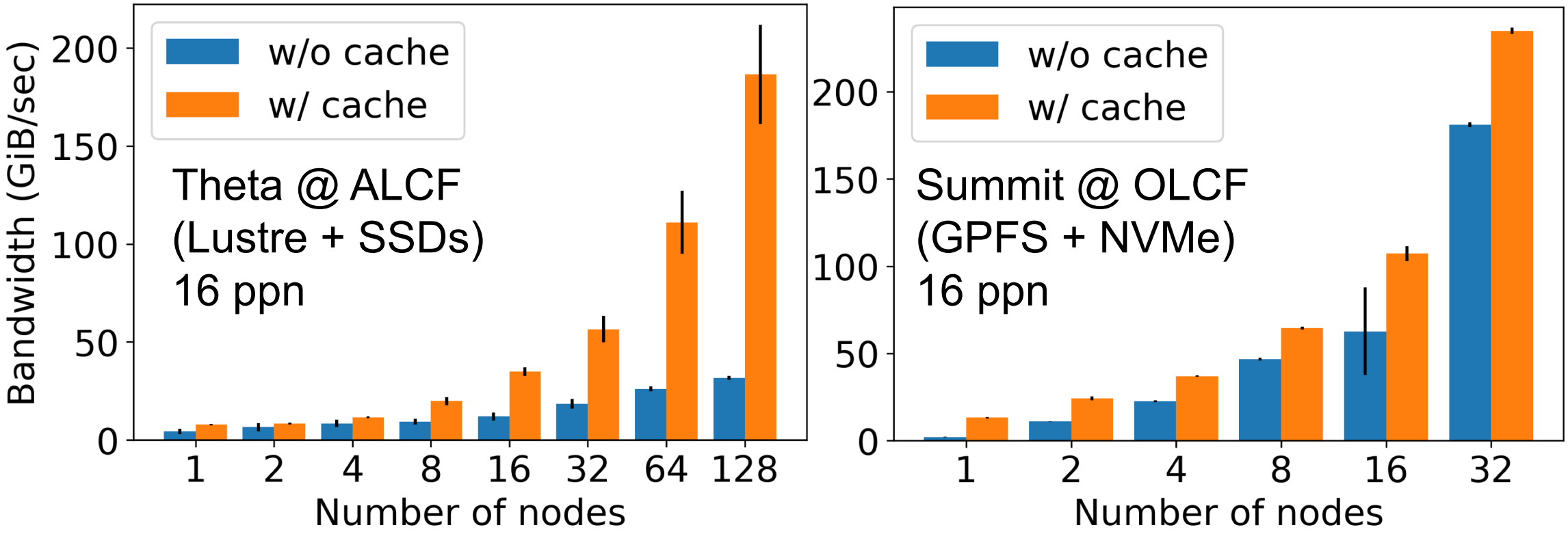
Initial Performance Evaluation

Initial performance evaluation – parallel write



Parallel write performance: each process writes 16MB of data to a shared HDF5 file. The type of node-local storage is either SSD/NVMe or RAM. With caching, the write bandwidth scale linearly with a larger aggregate bandwidth surpassing the Lustre / GPFS write bandwidth.

Initial performance evaluation – parallel read



Parallel read performance. The bandwidth is averaged over four iterations. At each step, the application reads a random batch (32) of samples (224x224x3) with shuffling. The application reads through the entire dataset in one iteration.

Conclusion

- Node-local storage caching / staging improves the scalability and achieves higher aggregate bandwidth over direct I/O to parallel file system.
- VOL implementation makes it easy to integrate into existing HPC applications and python workloads with minimal code change.

Future works

- Integrating with other ExaIO / ExaHDF5 developments, such as Async VOL (stacking), Subfiling VFD, Topology aware VFD.

Git Repo: https://bitbucket.hdfgroup.org/scm/hdf5vol/cache.git
huihuo.zheng@anl.gov

Acknowledgment

- This work was supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under contract number DE-AC02-05CH11231 (Project: Exascale Computing Project [ECP] - ExaHDF5 project).

- This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02- 06CH11357.

- 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.