



Argonne
NATIONAL
LABORATORY

... for a brighter future



U.S. Department
of Energy

UChicago ▶
Argonne_{LLC}



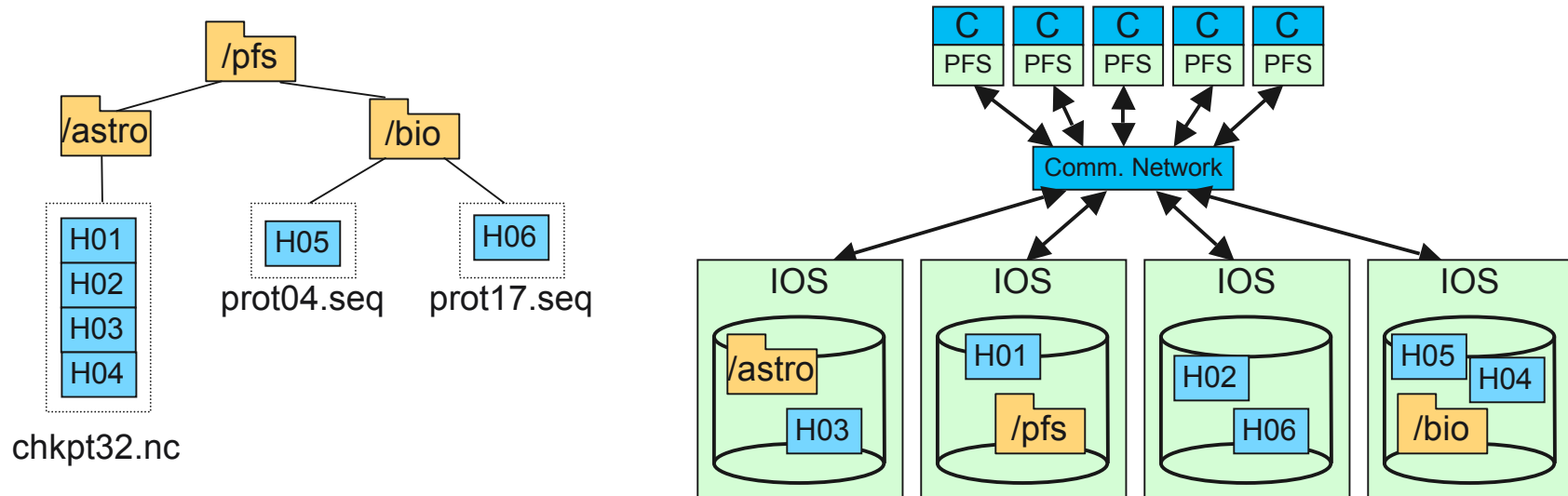
Office of
Science
U.S. DEPARTMENT OF ENERGY

A U.S. Department of Energy laboratory
managed by UChicago Argonne, LLC

Storage Mini-Workshop



Parallel File Systems

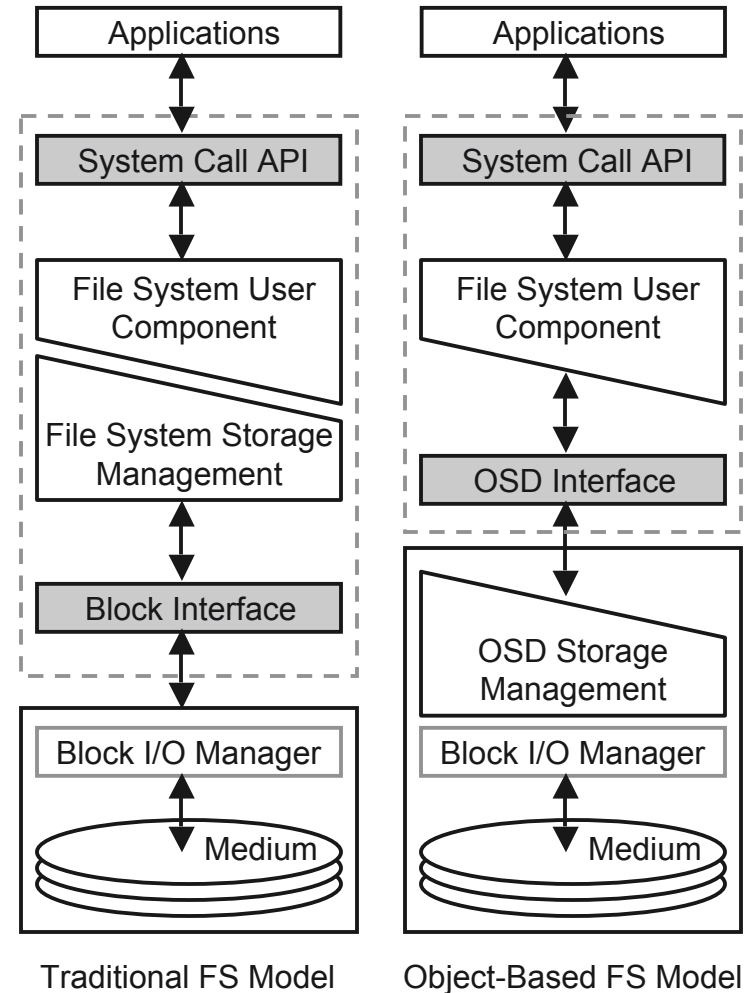


An example parallel file system, with large astrophysics checkpoints distributed across multiple I/O servers (IOS), while small bioinformatics files are each stored on a single IOS.

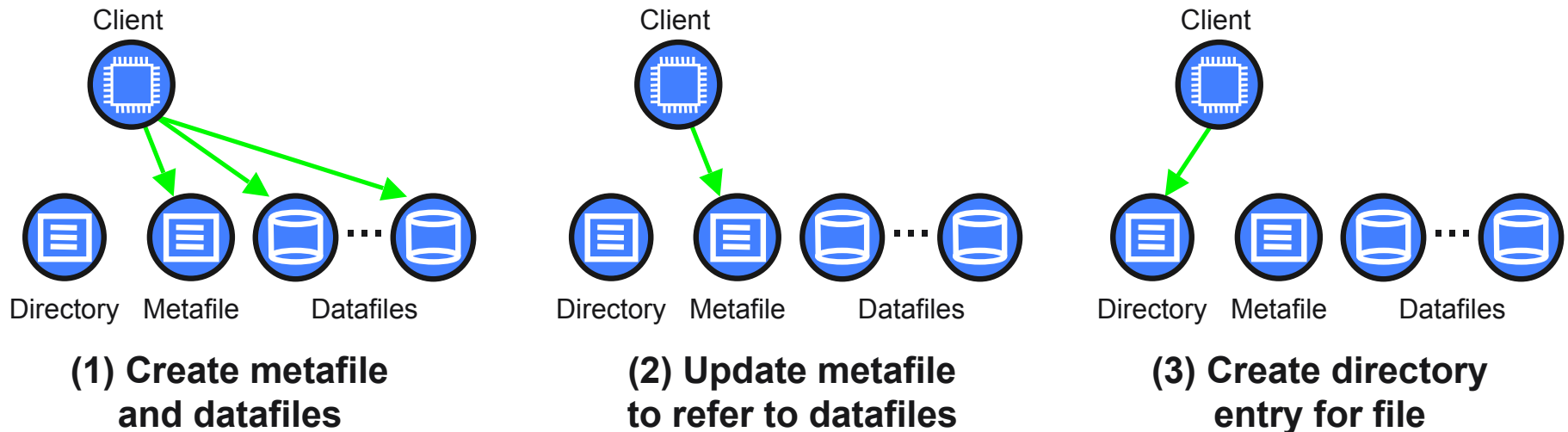
- File system information (data and metadata) spread across multiple servers
- Data for large files is striped across multiple resources
 - Simultaneous use of multiple servers, disks, and network links
- Two data models/APIs are exposed:
 - Directories, with entries
 - Files, with a linear array of bytes (and perhaps extended attributes)

Object Based Parallel File Systems

- Underneath, many parallel file systems are using an object-based storage organization
 - I/O servers provide access to named “objects”
 - *Hold byte streams*
 - *May have associated attributes*
 - I/O servers responsible for allocation of local storage space
- Separate metadata servers may be used to store file system name space and map file data onto objects
- These structures are hidden from applications and libraries outside the file system



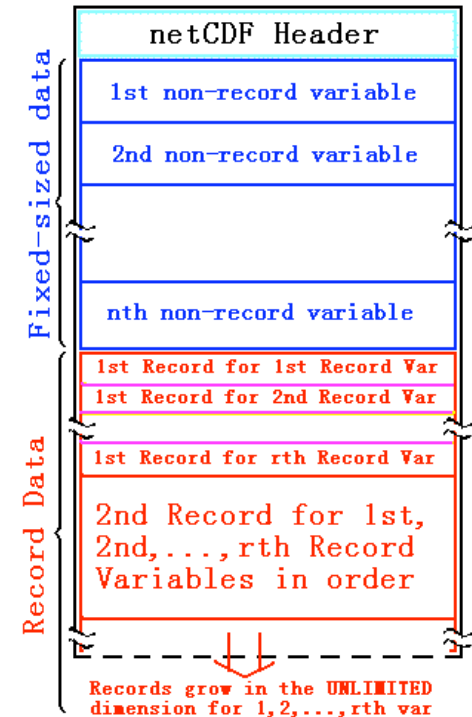
Adding Files to a Coherent Name Space



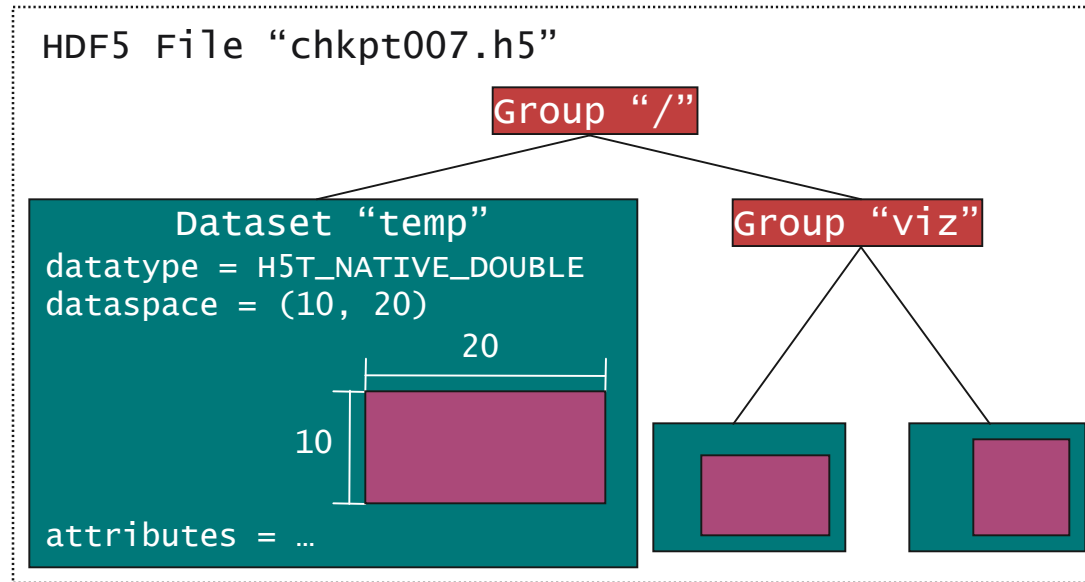
- Parallel file systems build up constructs in multiple steps, kept coherent through locks or atomic operations
 - e.g. clients orchestrate new file creation in a series of steps
- Again, the building-block operations are hidden from applications and other code outside the file system

netCDF/PnetCDF Files

- High-level I/O libraries try to present data structures that are more useful to scientists
- PnetCDF files consist of three regions
 - Header
 - Non-record variables (all dimensions specified)
 - Record variables (ones with an unlimited dimension)
- Record variables are interleaved, so using more than one in a file is likely to result in poor performance due to noncontiguous accesses
- All this is built on top of the linear array of bytes presented by the file system, because that's the only interface available



HDF5 Files



- HDF5 has a similar structure, only it looks even more like a file system inside a file because of the hierarchical nature
- Again, this structure is mapped onto a linear array of bytes

Defining Dimensions (in PnetCDF)

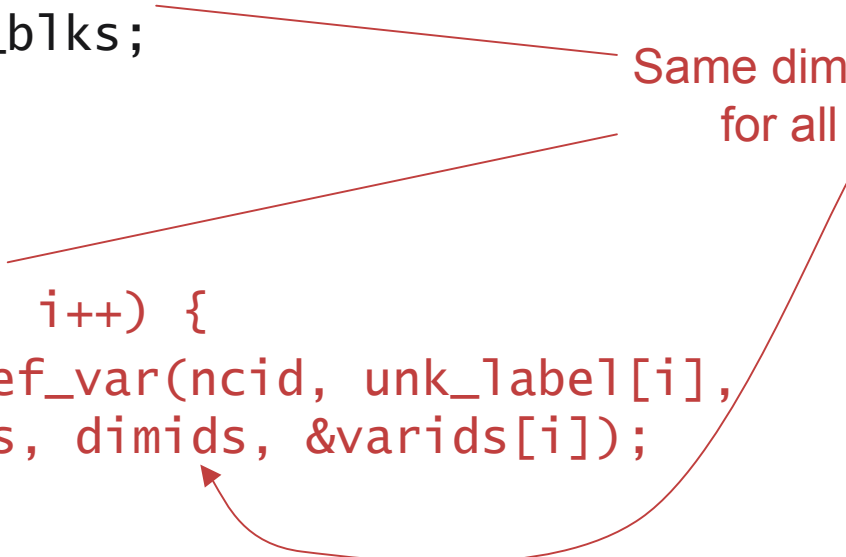
```
int status, ncid, dim_tot_blks, dim_nxb,  
    dim_nyb, dim_nzb;  
MPI_Info hints;  
/* create dataset (file) */  
status = ncmpi_create(MPI_COMM_WORLD, filename,  
    NC_CLOBBER, hints, &file_id);  
/* define dimensions */  
status = ncmpi_def_dim(ncid, "dim_tot_blks",  
    tot_blks, &dim_tot_blks);  
status = ncmpi_def_dim(ncid, "dim_nxb",  
    nzones_block[0], &dim_nxb);  
status = ncmpi_def_dim(ncid, "dim_nyb",  
    nzones_block[1], &dim_nyb);  
status = ncmpi_def_dim(ncid, "dim_nzb",  
    nzones_block[2], &dim_nzb);
```

Each dimension gets
a unique reference

Creating Variables (in PnetCDF)

```
int dims = 4, dimids[4];
int varids[NVARS];
/* define variables (X changes most quickly) */
dimids[0] = dim_tot_blks;
dimids[1] = dim_nzb;
dimids[2] = dim_nyb;
dimids[3] = dim_nxb;
for (i=0; i < NVARS; i++) {
    status = ncmpi_def_var(ncid, unk_label[i],
        NC_DOUBLE, dims, dimids, &varids[i]);
}
```

Same dimensions used
for all variables



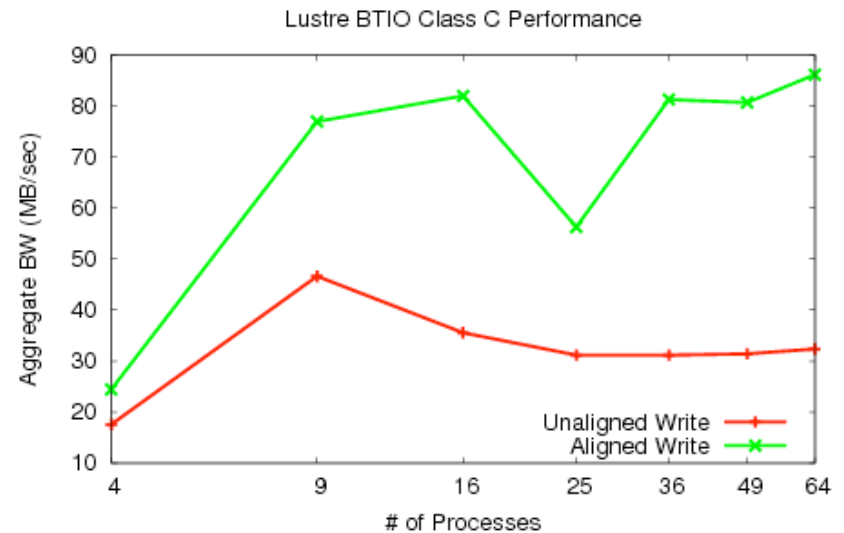
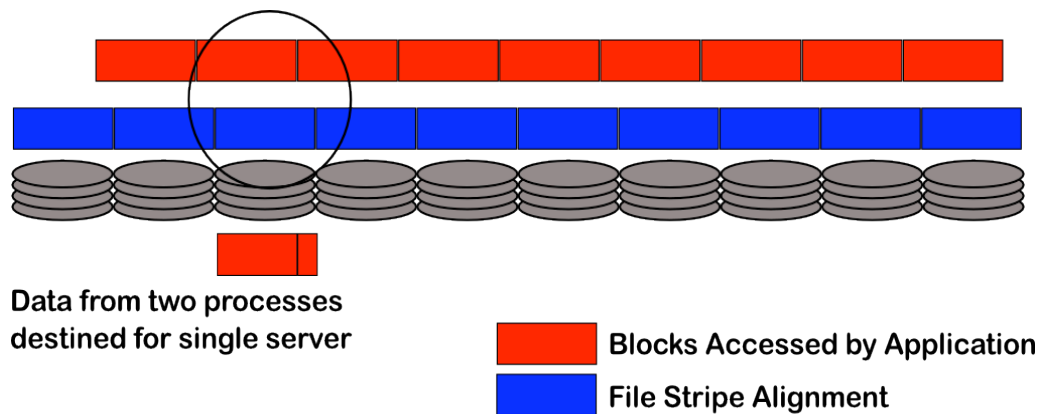
Writing Variables (in PnetCDF)

```
double *unknowns; /* unknowns[blk][nzb][nyb][nxb] */
size_t start_4d[4], count_4d[4];
start_4d[0] = global_offset; /* different for each process */
start_4d[1] = start_4d[2] = start_4d[3] = 0;
count_4d[0] = local_blocks;
count_4d[1] = nzb;  count_4d[2] = nyb;  count_4d[3] = nxb;
for (i=0; i < NVAR; i++) {
    /* ... build datatype "mpi_type" describing values of a
       single variable ... */
    /* collectively write out all values of a single variable
       */
    ncmpi_put_vara_all(ncid, varids[i], start_4d, count_4d,
        unknowns, 1, mpi_type);
}
status = ncmpi_close(file_id);
```

Typical MPI buffer-
count-type tuple

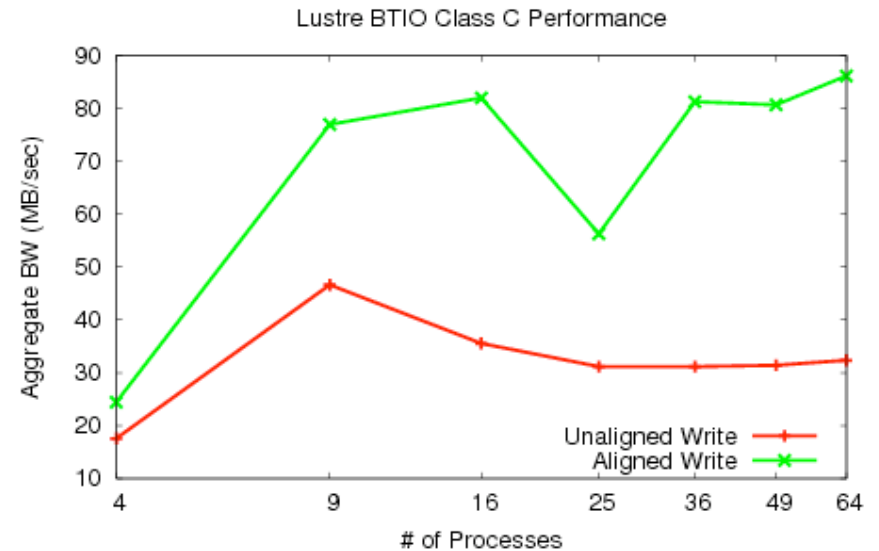
Stripe Alignment and I/O Performance

- Mapping onto linear array of bytes can lead to performance problems
- One reason is unaligned access
 - More than one process sending data to a single server simultaneously
 - *Alignment is with respect to file striping by the file system*
 - *Also impacts behavior of locking subsystem (false sharing)*
 - Can be managed by MPI-IO in some cases (data from W.-K. Liao)



BTIO and Stripe Alignment

- BTIO class C I/O Benchmark
 - 3D array access:
162 x 162 x 162 40-byte records
 - 40 writes/reads
 - Collective I/O (3D subarrays)
 - Total I/O amount = 12974.63 MB
- Results here from Tungsten machine at NCSA running Lustre
 - 8 I/O servers, 1MB stripe size
 - Up to 11GB/s peak I/O to independent files!
- “Aligned” results use MPI-IO research prototype with file system alignment awareness (from NWU)
 - Augmentation to two-phase optimization



Data from Wei-Keng Liao (NWU)

Idea: Expose File System Constructs

- What if we allowed high-level libraries access to the underlying FS structures?
- Variables could be split across FS objects as appropriate
- Alignment could be explicitly managed
- Metadata could be stored in separate objects, keeping it out of data path
- Hierarchical structures could be easily constructed and managed
- Also helps with active storage, because data can be placed to maintain necessary locality



(Some of the) Challenges

- How do we present resulting data structures back to users?
 - Big “file”?
 - Directory hierarchy?
 - How do existing UNIX tools work on them? Or do they?
- How do we serialize the resulting data structures?
 - What is the right order?
 - How do we preserve the structure we created, so we can recreate if placed on this FS again?
- What is the right API?
 - Can we somehow retain the ability to use POSIX and/or MPI-IO interfaces?

