Date: 16 June 2016	# M4.2 ACME Migration to FFIO Stack

Government Purpose Rights

Prime Contract No.: DE-AC52-07NA27344
LLNL Subcontract No.: B613306
Subcontractor Name: Intel Federal LLC, on behalf of itself, its parent and AffiliatesSubcontractor Address: 4100 Monument Corner Dr, Ste 540, Fairfax, VA 22030

The Government's rights to use, modify, reproduce, release, perform, display, or disclose this technical data are restricted by the above agreement.

Limited Rights

Prime Contract No.: DE-AC52-07NA27344
LLNL Subcontract No.: B613306
Subcontractor Name: Intel Federal LLC, on behalf of itself, its parent and AffiliatesSubcontractor Address: 4100 Monument Corner Dr, Ste 540, Fairfax, VA 22030

The Government's rights to use, modify, reproduce, release, perform, display, or disclose this technical data are restricted by the above agreement.

NOTICES

Acknowledgment: This material is based upon work supported by Lawrence Livermore National Laboratory subcontract B613306.

ACME Migration to FFIO Stack

M. Scot Breitenfeld[1], Neil Fortner[2]

[1] brtnfld@hdfgroup.org, [2]nfortne2@hdfgroup.org

Nomenclature

BB	Burst Buffer
CN	Compute Node
DAOS	Distributed Application Object Storage
EFF	Exascale FastForward
ION	I/O Node

I. NetCDF-4 Overview

NetCDF is a set of software libraries used to facilitate the creation, access, and sharing of array-oriented scientific data in self-describing, machine-independent data formats. This project uses the EFF version of netCDF, meaning a netCDF version built using the EFF HDF5 implementation, found at

https://hdfgit.hdfgroup.org/scm/ffwd2/netcdf-c.git

II. ACME/PIO Overview

The end goal of implementing an EFF version of netCDF is to demonstrate an application which uses netCDF. The netCDF application identified for this quarter is software associated with the Accelerated Climate Modeling for Energy (ACME) program. Their software uses the package Parallel I/O (PIO) to perform I/O which, in turn, uses as its backend the netCDF file format. PIO uses a large subset (211) of the netCDF functions (not all of which would need EFF versions), and complete list of those functions is given in Appendix A.

The PIO performance testing program *pioperformance.F90* is an ACME I/O stand-alone driver program which closely duplicates the I/O pattern from an actual ACME application. Therefore, this project will implement the program *pioperformance.F90* within the EFF stack framework via an EFF version of PIO. *Pioperformance.F90* uses the following PIO functions,

- PIO_init
- PIO_Readdof
- **PIO_CreateFile**
- PIO_InitDecomp
- PIO_setframe
- **PIO_write_darray**
- **PIO_read_darray**
- PIO_freedecomp
- **PIO_OpenFile**
- **PIO_CloseFile**
- PIO_def_var
- PIO_def_dim
- PIO_def_att
- **PIO_enddef**

The PIO APIs in **blue** utilize EFF netCDF APIs and therefore have an EFF equivalent API. All new EFF PIO C APIs are indicated by appending a "*_ff*" to the C function names. The Fortran EFF PIO APIs are implemented by overloading the current Fortran PIO APIs. For example, pio_write_darray would be,

```
CALL PIO_setframe(File, vard(nv), recnum)
CALL pio_write_darray(File, vard(nv), iodesc_r8, dfld(:,nv)    , ierr, fillval= PIO_FILL_DOUBLE, &
     h5ff_tr_num=h5ff_tr_num, h5ff_tr_id=h5ff_tr_id, h5ff_rc_id=h5ff_rc_id, h5ff_e_stack=e_stack)
```

where the last four arguments are optional EFF parameters. If these optional parameters are not present, then PIO well automatically default to the non-EFF netCDF APIs.

The EFF PIO source files and test code can be downloaded from

https://hdfgit.hdfgroup.org/scm/ffwd2/parallelio.git

PIO expects as input from the application the partitioned data arrays for each process. Additionally, PIO has the option for requesting a subset of the CN that will perform the IO. Hence, PIO aggregates the IO from each process to only a subset of processes for IO. The IO processes then uses netCDF APIs to carry out the IO. PIO implements two methods for aggregating the IO from all the processes to the subset of IO processes. In the *box* method, each compute task will transfer data to one or more of the IO processes. For the the *subset* method, each IO process is associated with a unique subset of compute processes for which each compute process transfers data to only one IO process [1]. In general, the *subset* method reduces the overall communication cost when compared to the *box* method. All the demonstrations use the *box* method.

Additionally, since PIO has the capability of using a subset of processes for IO, *EFF_init* (an EFF HDF5 API used to start the EFF stack) was extended to handle a MPI sub-communicator group. Hence, in the current EFF PIO implementation, only those processes involved in IO will initialize the EFF stack. This initialization of the EFF stack happens automatically when the IO MPI sub-communicator is created in PIO and it is finalized when this same sub-communicator is freed in PIO.

II.a. Updates to the EFF NetCDF schema

Additional NetCDF EFF APIs not implemented last quarter, but needed by PIO were:

- nc_sync
- nc_put_vara_double
- nc_get_vara_double

Firstly, in the initial implementation of EFF NetCDF, the only EFF stack variables exposed to the calling program were the transaction and version number. Additionally, the EFF stack variables were globally accessible by the inclusion of *nc_h5ff.h* in the application. In the current implementation, the global EFF stack variables are passed as arguments to the NetCDF and PIO APIs. The current NetCDF implementation allows access to all the EFF stack variables,

- Read context identifier,
- Version number,
- Event stack identifier,
- Transaction identifier

from PIO. Furthermore, the Fortran PIO APIs are overloaded by having the EFF stack variables being optional arguments, eliminating the need to have additional Fortran EFF PIO APIs.

Secondly, in the initial implementation, the NetCDF APIs managed the transactions internally, from creation to closing, with no control of the stack given to the calling program. In the current implementation, stack control is now shifted to within PIO. This allows for a PIO API to call multiple NetCDF APIs and to use the same or different transactions depending on which set of NetCDF calls are made within the PIO API. The transaction number is initialized in the application code and is then automatically incremented as needed within the EFF PIO APIs. Hence, transaction management is handle in the PIO library, but it is still accessible to the application.

Exposing the event stack identifier allows the application to use asynchronous I/O. To do this, the application creates an event stack and passes it to an operation that can be asynchronous. At the point in the future when the application needs that operation to be finished, it can use *H5ESwait* or *H5ESwait_all* to block until it is completed. Even if no event stack is passed to some metadata operations, NetCDF will use an internal event stack to issue asynchronous operations that can run concurrently, improving performance. In this case, NetCDF will wait on all operations before returning.

Other improvements to NetCDF include:

- Support for unlimited dimensions (supported only with collective access, and only for the slowest changing dimension)
- "Links" from variables to their dimensions, allowing the variables to be queried about their dimensions
- Support for multiple server processes and nodes
- Fixed several bugs, including some that affect mainline NetCDF4 (these have been or will soon be forwarded to NetCDF4 developers)

III Demonstration Example

The test program uses two input files; the first file contains namelist settings for the testing parameters and the second file contains the decomposition information from a PIO program (e.g. CESM, ACME). Decomposition files are available at,

https://svn-ccsm-piodecomps.cgd.ucar.edu/trunk.

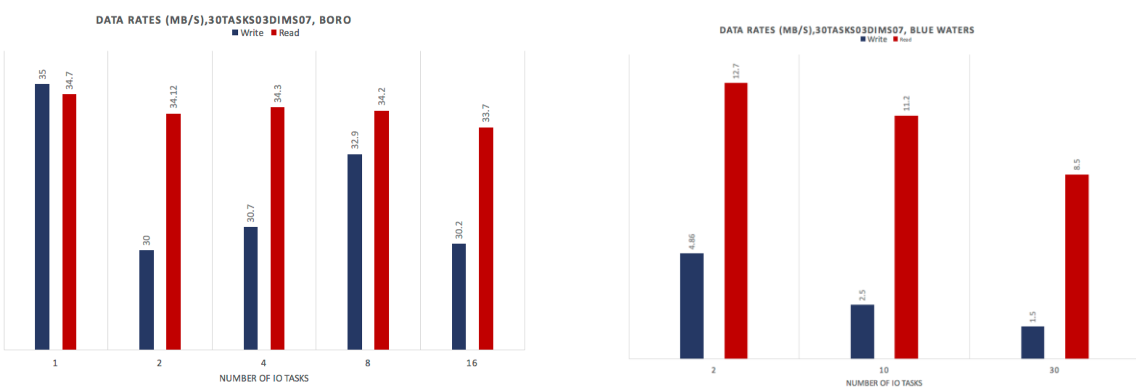
The format of the decomposition file name is,

piodecomp<***NUM_MPI_PROCESSES***>tasks<***NUM_DIMENSIONS***>dims<***COUNTER***>.dat

where *NUM_MPI_TASKS* is the number of MPI tasks/ranks (30, 1024, 2048 and 16384 are available), *NUM_DIMENSIONS* is the number of dimensions in the decomposition (typically corresponding to the variable for which the decomposition was created), and *COUNTER* is a file identifier counter. [2]

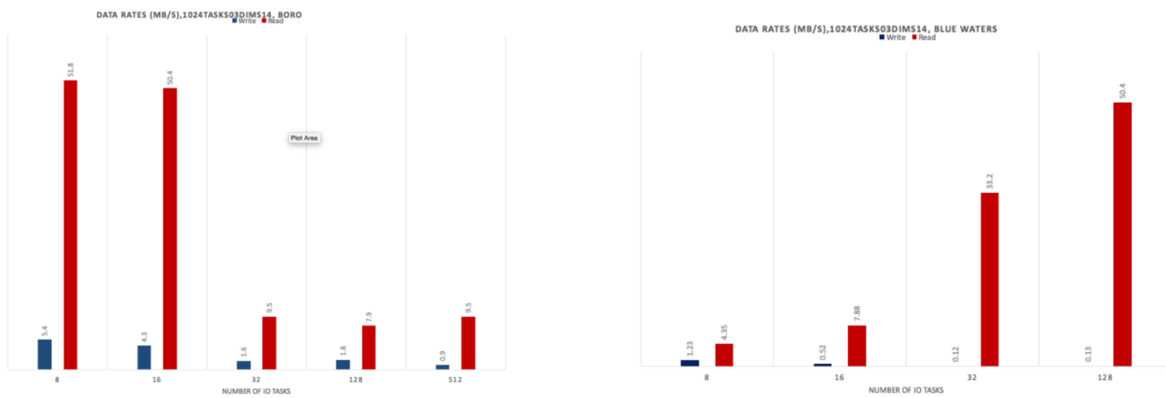
III.a Initial benchmarks for PIO on boro

Several testing runs were made on Intel's *boro* cluster in order to verify the EFF PIO implementation, and a sampling of those test's data rates are presented in Figures 1-4. Furthermore, for qualitative purposes, the performance of a standard implementation of PIO is also presented for 30 and 1024 processes runs on Blue Waters at NCSA at the University of Illinois at Urbana-Champaign.



(a) (b)

Figure 1 – 30 process data rates for various numbers of IO tasks for boro (a) and Blue Waters at NCSA (b).



(a) (b)

Figure 2 – 1024 process data rates for various numbers of IO tasks for boro (a) and Blue Waters at NCSA (b).

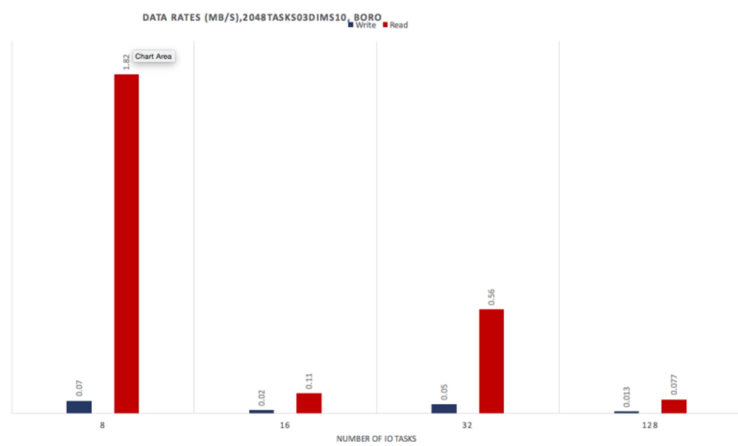


Figure 3 – 2048 process data rates for various numbers of IO tasks for boro.

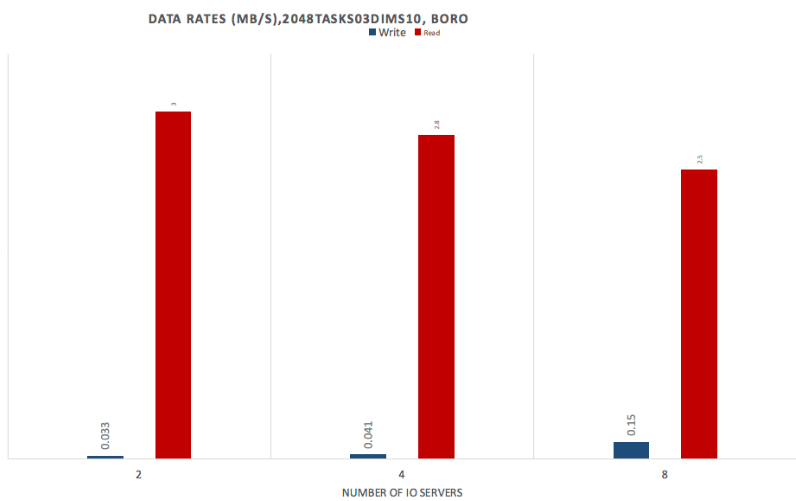


Figure 4 –Effect of the number of EFF servers on the data rate for 8 PIO IO tasks.

III.b Overview of Demonstration

The test program *pioperformance.F90* reads namelist and then generates test data consisting of integers, 4-byte reals and 8-byte reals. It then writes the data using EFF netCDF via PIO APIs. It then reads the data back using EFF PIO, checks for correctness, and outputs the data rate in reading and writing the data.

III.c Summary of the Demonstration on June 16th, 2016

CASE I – This demonstration is for 30 processes, it creates a **limited** dataset, and uses the following parameters:

Decomposition File	*Number of IO tasks*	*Number of variables*	*Number of frames*
piodecomp30tasks03dims07.dat	8	3	1

CASE II – This demonstration is for 30 processes, creates an **unlimited** dataset, and uses the following parameters:

Decomposition File	*Number of IO tasks*	*Number of variables*	*Number of frames*
piodecomp30tasks03dims07.dat	16	1	3

The steps for starting the servers were as follows,

(1) Clean-up any files created by previous runs:

cleanup-all.sh; ssh boro-9 /scratch/ESSIO/opt/sbin/cleanup-all.sh;

(2) Start two servers

mpirun -n 2 --hostfile host_srv ./h5ff_server

Once the servers are started, then the PIO application can be run,

mpirun -n 30 --hostfile hosts -iface ib0 pioperf pioperf.nl.30.caseII

where the "*hosts*" file is the list of nodes on boro to use, option *-iface* instructs MPI to use InfiniBand, and the input argument for *pioperf* is the namelist control file. In the demonstration, both cases I and II completed successfully and the correctness of each program was verified.

References

[1] http://ncar.github.io/ParallelIO/decomp.html
[2] https://groups.google.com/forum/#!topic/parallelio/vtvOXP-sjZE

Appendix A

List of NetCDF APIs called by PIO,

- nc_function
- nc_var_par_access
- nc_put_vara_double
- nc_put_vara_int
- nc_put_vara_float
- nc_get_vara_double
- nc_get_vara_int
- nc_get_vara_float
- nc_open
- nc_open_par
- nc_create_par
- nc_create
- nc_close
- nc_delete
- nc_sync
- nc_get_var1_schar
- nc_get_vars_ulonglong
- nc_get_varm_uchar
- nc_get_varm_schar
- nc_get_vars_short
- nc_get_var_double
- nc_get_var_int
- nc_get_var_ushort
- nc_get_vara_text
- nc_get_var1_float
- nc_get_var1_short
- nc_get_vars_int
- nc_get_var_text
- nc_get_varm_double
- nc_get_vars_schar
- nc_get_vara_ushort
- nc_get_var1_ushort
- nc_get_var_float
- nc_get_vars_uchar
- nc_get_var
- nc_get_var1_longlong
- nc_get_vars_ushort
- nc_get_var_long
- nc_get_var1_double
- nc_get_vara_uint
- nc_get_vars_longlong
- nc_get_var_longlong
- nc_get_vara_short
- nc_get_vara_long
- nc_get_var1_int
- nc_get_var1_ulonglong
- nc_get_var_uchar
- nc_get_vara_uchar
- nc_get_vars_float
- nc_get_vars_long
- nc_get_var1
- nc_get_var_uint
- nc_get_vara
- nc_get_vara_schar
- nc_get_var1_uint
- nc_get_vars_uint
- nc_get_varm_text
- nc_get_var1_text
- nc_get_varm_int
- nc_get_varm_uint
- nc_get_varm
- nc_get_vars_double
- nc_get_vara_longlong
- nc_get_var_ulonglong
- nc_get_vara_ulonglong
- nc_get_var_short
- nc_get_varm_float
- nc_get_var1_long
- nc_get_varm_long
- nc_get_varm_ushort
- nc_get_varm_longlong
- nc_get_vars_text
- nc_get_var1_uchar
- nc_get_vars
- nc_get_varm_short
- nc_get_varm_ulonglong
- nc_get_var_schar
- nc_inq
- nc_inq_dimname
- nc_put_att_short
- nc_rename_dim

- nc_get_att_double
- nc_set_fill
- nc_def_var
- nc_def_var_deflate
- nc_put_att_double
- nc_inq_dim
- nc_get_att_uchar
- nc_inq_var_fill
- nc_inq_attid
- nc_inq_vartype
- nc_put_att_schar
- nc_inq_vardimid
- nc_get_att_ushort
- nc_inq_varid
- nc_inq_attlen
- nc_inq_atttype
- nc_rename_var
- nc_inq_natts
- nc_put_att_ulonglong
- nc_inq_var
- nc_rename_att
- nc_put_att_ushort
- nc_inq_dimid
- nc_put_att_text
- nc_get_att_uint
- nc_inq_format
- nc_get_att_long
- nc_inq_attname
- nc_inq_att
- nc_put_att_long
- nc_inq_unlimdim
- nc_get_att_float
- nc_inq_ndims
- nc_put_att_int
- nc_inq_nvars
- nc_enddef
- nc_put_att_uchar
- nc_put_att_longlong
- nc_inq_varnatts
- nc_get_att_ubyte
- nc_get_att_text
- nc_del_att
- nc_inq_dimlen
- nc_get_att_schar
- nc_get_att_ulonglong
- nc_inq_varndims
- nc_inq_varname
- nc_def_dim
- nc_put_att_uint
- nc_get_att_short
- nc_redef
- nc_put_att_ubyte
- nc_get_att_int
- nc_get_att_longlong
- nc_put_att_float
- nc_inq_var_deflate
- nc_inq_var_szip
- nc_def_var_fletcher32
- nc_inq_var_fletcher32
- nc_def_var_chunking
- nc_inq_var_chunking
- nc_def_var_fill
- nc_def_var_endian
- nc_inq_var_endian
- nc_set_chunk_cache
- nc_get_chunk_cache
- nc_set_var_chunk_cache
- nc_get_var_chunk_cache
- nc_put_vars_uchar
- nc_put_vars_ushort
- nc_put_vars_ulonglong
- nc_put_varm
- nc_put_vars_uint
- nc_put_varm_uchar
- nc_put_var_ushort
- nc_put_var1_longlong
- nc_put_vara_uchar
- nc_put_varm_short
- nc_put_var1_long
- nc_put_vars_long
- nc_put_var_short
- nc_put_var1_ushort
- nc_put_vara_text
- nc_put_varm_text
- nc_put_varm_ushort
- nc_put_var_ulonglong
- nc_put_var_int
- nc_put_var_longlong
- nc_put_var_schar
- nc_put_var_uint
- nc_put_var
- nc_put_vara_ushort
- nc_put_vars_short
- nc_put_vara_uint
- nc_put_vara_schar
- nc_put_varm_ulonglong
- nc_put_var1_uchar
- nc_put_varm_int
- nc_put_vars_schar
- nc_put_var1
- nc_put_var1_float
- nc_put_varm_float
- nc_put_var1_text
- nc_put_vars_text
- nc_put_varm_long
- nc_put_vars_double
- nc_put_vara_longlong
- nc_put_var_double
- nc_put_var_float
- nc_put_var1_ulonglong
- nc_put_varm_uint
- nc_put_var1_uint
- nc_put_var1_int
- nc_put_vars_float
- nc_put_vara_short
- nc_put_var1_schar
- nc_put_vara_ulonglong
- nc_put_varm_double
- nc_put_vara
- nc_put_vara_long
- nc_put_var1_double
- nc_put_varm_schar
- nc_put_var_text
- nc_put_vars_int
- nc_put_var1_short
- nc_put_vars_longlong
- nc_put_vars
- nc_put_var_uchar
- nc_put_var_long
- nc_put_varm_longlong