# PIOVDC

## ABOUT

PIOVDC is an extension to NCAR's Parallel IO software used by NCAR and various other organizations for easily writing massive data sets in an optimized, parallel manner. The extensions involve incorporating both extra logic and a subset of the VDF library from the VAPOR visualization package in order to allow for data written through PIO to be compressed on the fly into a Vapor Data Collection (VDC). This allows for scientists writing massive data sets to have the option of either writing less data at a lower than full resolution or write it all into files with progressive compression access that can be visualized with the VAPOR package. Compared to the offline tools provided by the VAPOR package, using PIOVDC allows data to go from a user program's memory straight into the vdf format, with no manual conversion post-process needed.

## BUILDING/INSTALLING

PIOVDC is currently comprised of two different projects that must be fetched, compiled, and installed separately.

The prerequisite software necessary to build/install PIOVDC are:

- An MPI runtime/compiler suite
- A configured and installed PNetCDF or NetCDF (>4.1) + HDF5 (>1.8.5). The latter is provided as a fallback option, but not recommended as performance with NetCDF using PIOVDC does not seem as reliable as PNetCDF and version/bug conflicts make matching up the right NetCDF and HDF5 versions much more difficult than simply using PNetCDF. **This document will assume that PNetCDF is being used to build PIOVDC.**
- An installed expat library

### PIO

The first project is the PIO library. To retrieve PIO, you must use svn to download the PIO software.

```
svn co https://parallelio.googlecode.com/svn/trunk_tags/pio1_5_7/
```

After retrieving the PIO source, the software must be configured using your installed PNetCDF location. PIO uses configure options to enable/disable the optional VDC components. Execute the commands:

```
cd pio1_5_7/pio
./configure --enable-pnetcdf=yes PNETCDF_PATH=/path/to/installed/pnetcdf --enable-netcdf=no --enable-
compression=yes
```

After configuration the software is ready to be built by running GNU make

**WARNING**: if the configuration completes and MPICC has not been detected (if you see the string "{MPICC=}" in the list of Output Variables generated by configure), you should rerun configure manually setting the environment variable to a known MPI C compiler:

./configure MPICC=mpicc --enable-pnetcdf=yes PNETCDF_PATH=/path/to/installed/pnetcdf --enable-netcdf=no --enable-compression=yes

Without the MPICC compiler there will be files that cannot be built using the make files.

Running GNU make will generate the necessary PIO files, libpio.a, and pio.mod:

```
make
```

For now you are finished with PIO

### VAPOR

Next the source for building the VDF library will be needed:

The VDF library subset can be downloaded from the PIO repository.

```
svn co https://parallelio.googlecode.com/svn/libpiovdc/trunk/ .
```

This will download a folder containing all the necessary parts to build the VDF libraries. To start, run autoreconf to generate the platform specific files, then run the configure script:

```
./configure --with-pnetcdf=/path/to/pnetcdf
```

**WARNING: If the expat header installed into the system and not detected then it is in a non-standard path. Use the --with-expat option in order to tell the configure script where expat is installed.**

After configuration is complete you may build by running GNU make. This will generate two static libraries, libpiovdc.a, and libpiocommon.a in the vdf and common directories respectfully, both of which are necessary for running PIOVDC.

## Linking

Once all of your source code has been compiled into the three static libs and the pio module file: libpio.a, libpiovdc.a, and libpiocommon.a, you can link any test code to the libs. The PIO library is Fortran, while the VDF libraries are C++ which makes linking a delicate operation. The process for doing it changes depending on what compiler suite you are using, and possibly what version of compiler. For example, to use the intel compiler suite to link to a test_lib.F90 user program:

### Intel Compiler Suite

Build your user program into an object file using ifort through the mpi wrapper script

```
mpif90 -c test_lib.F90
```

Link your user program to the libraries (static libs and pio.mod are located in current dir):

```
mpif90 test_lib.o -o TestLib -cxxlib -L. -L/path/to/pnetcdf/lib -lpio -lpiovdc -lpiocommon -lpnetcdf -lexpat
```

### GNU Compiler Suite

Build your user program into an object file

```
mpif90 -c test_lib.F90 -ffree-line-length-none
```

Link your user program to the librares, using either gfortran or g++ through the mpi wrappers:

GFORTRAN

```
mpif90 test_lib.o -o TestLib -L. -L/path/to/pnetcdf/lib -lpio -lpiovdc -lpiocommon -lpnetcdf -lexpat -lstdc++
```

G++

```
mpiCC test_lib.o -o TestLib -L. -L/path/to/pnetcdf/lib -lpio -lpiovdc -lpiocommon -lpnetcdf -lexpat -lgfortran
```

**WARNING: Depending on your installation the default mpi library might not contain the appropriate methods. If you get warnings about undefined mpi symbols in either the Fortran libpio or the C++ libpiovdc, then the mpi compiler script is not providing the symbols for said language and those MPI functions must be imported separately. For the OpenMPI implementation, the specific libraries needed are libmpi_cxx and libmpi_f77 when linking with gfortran and g++, respectively.**

# USING PIOVDC

PIOVDC functions as a well integrated extension to PIO, all that is required to use PIOVDC is overloading a few normal PIO api calls and omitting a few unnecessary PIO steps if you do not plan on using non compressed data. If the user is familiar with PIO, then using PIOVDC will take almost no additional effort. For those unfamiliar with PIO, I will explain the basic workflow. Complete PIO documentation is available here.

This workflow assumes that your program is running in an MPI environment, with multiple MPI tasks.

## Prerequisites

prepared user data - as PIOVDC only works with in-memory data, the program must already have loaded the data in memory. PIO provides facilities for reading data into memory, but the file formats are not guaranteed to support the user program data (PNetCDF, NetCDF +HDF5, MPI-IO, and raw binary are supported by PIO) and PIOVDC is not coded to use these facilities to get the data into memory as part of the regular operation.

available space - PIO uses special data rearrangement in order to ensure that IO gets good performance. As a result, the memory requirements for a PIOVDC using program can be 2-3X the size of the data set you intend to write. Please ensure that all of the MPI tasks together have enough memory to run your data set beforehand, as the performance can be slow and unreliable when not enough memory is supplied.

WARNING: PIO is highly optimized IO software, and as a result is very dependent on the underlying performance of the machine it is running on. Depending on the architecture and the way that the machine is setup it is also possible that you may see non-linear scaling. For example, on the Janus super computing cluster a 1024^3 sample data set can be written using 64 computational and 64 IO tasks, but it takes 4x the amount of tasks (256 comp, 256 IO) to safely write a 2048^3 data set.

## Workflow

An example program can be found here. This is the program used to test the PIOVDC compression with in data held in memory. A quick summary of the general usage of PIOVDC and PIO follows:

1. Initialize PIO
    a. PIO requires the use of MPI, so MPI must first be started
    b. call MPI_Comm_rank to get the rank of the current process
    c. call PIO_Init - requires setting arguments to certain values to get the expected behavior. See the file pio1_4_0_vdc/pio/test_lib.F90 that comes in the repository for an example
2. Set up compdof
    a. IO in PIO is accomplished by converting from a computational decomposition, which is the arrangement of data used by a user program, to an IO decomposition used by PIO to write out data using either a subset of or the entire set of computational tasks.
    b. A compdof is simply an array that maps the local data held by a single MPI task to the global array being written by PIO. For example, for two MPI tasks writing an 2x2x2 cube of data, the compdof for the first task can map the 4 values contained by task 0 to the global indices (1, 2, 3, 4), and the 4 values contained by task 1 to the global indices(5, 6, 7, 8). It does not necessary have to be a linear relationship, using the compdof allows one to map a local element of data to any global index. As long as there are no conflicts PIO is content to take whatever mapping that is handed to it
3. Initialize PIO decomposition
    a. Once you have your comp->global io mapping PIO takes that information, and an IO start/count for each MPI task, and creates the decomposition necessary to support the IO that the user desires. For PIOVDC, the iostart/counts are automatically calculated and the user need not worry about setting them, simply call the initdecomp method with your compdof and no start/counts like in the test_lib example.
4. Define files to output
    a. PIO behaves as a higher level abstraction that strongly resembles the NetCDF conventions. As such there is a process for defining files. This process differs depending on if you plan on writing compressed VDC files or regular uncompressed PIO files.
        i. Uncompressed files
            1. Create file
            2. Define dimensions of the data set
            3. Define variables that will be stored in the file
            4. End file definition, file is now out of define mode and can be written to.
        ii. Compressed files
            1. Create file (file is not valid until define mode is left)
            2. Define variables that will be stored in file (VDC collections always have three dimensions, x/y/z, of a set range)
            3. End file definition, file is now out of define mode and can be written to.
5. Write/Read Data
    a. Once all the setup is accomplished, the user program can call writedarray to output the local data held by the MPI task. There is a slight difference between calling the method for compressed vs uncompressed data
        i. Due to the design of a VDC compressed files need the timestep data for every write and read of data. A VDC is a capable of storing the data from multiple time steps together, hence the requirement to specify which time step a write belongs to.
        ii. Optionally, one can change the level of detail and refinement level for each write. The way that compression works in a VDC is that the data is run through a wavelet transformation, and the coefficients of the transform are then stored inside netcdf files. The default compression levels are 1:1, 10:1, 100:1, and 500:1. Progressive compression comes from the fact that each of these compression levels corresponds to a single netcdf file. The level of detail functions as selector that allows you to choose which of these compression levels you want to be outputted. For example, an LOD of -1 defaults to outputting all compression files, a LOD of 0 outputs only the most compressed 500:1 level, and 1 outputs 500:1 and 100:1 and so on.
    b. Due to the way PIO is setup, it is possible to interleave calls to write compressed and uncompressed data, the user simply has to manually keep track of the files being written to and their variables.
6. Close PIO
    a. Before calling PIO's close function all non-compressed files must be closed
    b. After calling PIO's finalize function remember to close MPI by calling it's finalize function