

Introduction

Parallel netCDF (often abbreviated PnetCDF) is a library for parallel I/O providing higher-level data structures (e.g. multi-dimensional arrays of typed data). PnetCDF produces and consumes the same file format as the serial netCDF library, meaning PnetCDF can operate on existing datasets, and existing serial analysis tools can process PnetCDF-generated files.

The most distinguishing feature of both netCDF and PnetCDF is the *bi-modal* programming interface. An application creating a file will first enter *define mode*, in which it can describe all attributes, dimensions, types and structures of variables. The program will then exit "define mode" and enter *data mode*, in which it actually performs I/O. We'll see that often in the following examples. This "declaration-before-use" model can be a little restrictive, but does allow for some aggressive optimization when carrying out I/O.

This brief tutorial was written with the assumption the reader has some familiarity with serial netcdf. If serial netcdf concepts like attributes, dimensions, and variables are not familiar, start with the [NetCDF Users Guide](#).

I/O from Master

In the old days, there were no parallel I/O facilities for netCDF. Applications would run in parallel, but when it came time to do I/O, a master process (typically MPI rank 0) would handle all the I/O. This approach can work for small amounts of I/O, but will not scale as the number of processes increases:

- When writing, the master process needs to allocate enough memory to hold the data from all the other processes.
- Again when writing, the master process becomes a serialization point as it receives data from all other processes.
- Reading requires the master process to send a message for how much data to expect to all processes, then send the actual data.
- I/O from one process means no parallel I/O can happen, limiting performance to one (serial) stream.

[Example writer](#) and [Example reader](#) demonstrate this less than ideal approach.

Separate files

We present the "one-file-per-process" approach not to recommend it, but rather because it is commonly seen.

This approach has some significant drawbacks. What if the number of writers differs from the number of readers? What if there are a million processes? What contextual information about the application data is lost in such an approach?

On some platforms, however, the "one-file-per-process" approach is the most straightforward way to achieve acceptable I/O performance. Observe the trade-offs made in this example: [Write one file per process](#) and [Read one file per process](#).

Real parallel I/O on shared files

The previous approaches either bypass parallel I/O entirely or hide a great deal of application context from PnetCDF. We now present a more natural way of carrying out I/O in a parallel program: operating on a shared file.

Shared-file I/O provides several benefits. First of all, because all processes can participate in collective I/O, the underlying MPI-IO library can make use of several powerful optimizations, such as file access alignment and collective buffering. Opening or creating a dataset (file) is a collective operation as well, meaning processes store and query metadata (the number, size, and location in file of attributes and variables) in an efficient manner. Data decomposition may be more sophisticated, but it is also more likely to match how the scientific application has already split up the data among processes.

Examples: [Writing data with standard API](#) and [Reading data via standard API](#)

Flexible interface

The standard netCDF and PnetCDF APIs explicitly specify the type of the application data (an array of integers, double precision, or floating point values, e.g. `ncmpi_put_vara_float_all`). We have further extended the PnetCDF API to accept arbitrary MPI datatypes. Say an application's data structures are more complex than a multidimensional array of a basic type, or if the application needs to write a non-contiguous selection of a given memory region to the dataset. In these situations, an MPI datatype can describe the desired data.

In these examples, we use `ncmpi_put_vara_all`, even though the datatype is a basic `MPI_INT` type:

Examples: [Writing data with Flexible API](#) and [Reading data via Flexible API](#)

Non-blocking interface

A set of "non-blocking" APIs is available in PnetCDF. They can aggregate multiple smaller requests into larger ones for better I/O performance. These routines follow the MPI model of posting operations, then waiting for completion of those operations.

The PnetCDF and netCDF APIs are variable oriented: if an application writes 50 variables to a dataset (file), it must make 50 calls where each call is carried out by a separate MPI-IO write call. With the PnetCDF non-blocking API, however, the library can take this collection of pending operations and then stitch them together into one larger, more efficient MPI-IO request.

Examples: [Non-blocking write](#) and [Non-blocking read](#)

Buffered write interface

The non-blocking APIs require users not to access any part of the I/O buffer after a non-blocking API is called, until a call to the `wait` API completes. This requirement may result in changes to the application programs if they use the same intermediate buffers for multiple write operations. Using intermediate buffers to convert/average the raw data into data for visualization and data analysis purposes is a common practice in many applications. In this case, the programs must allocate separate intermediate buffers for each non-blocking write operations.

PnetCDF includes a set of "buffered write" APIs that makes a copy of write request data into a user attached buffer. Hence, once the buffered write call returns, the contents of user's write buffers are free to change. However, users are required to explicitly call `ncmpi_buffer_attach` to allow PnetCDF to allocate an internal buffer for accommodating the write requests. In addition, a call to `ncmpi_buffer_detach` is required when this buffer is no longer needed.

Examples: Non-blocking buffered write in C and Non-blocking buffered write in Fortran