Using NETCDF with Fortran on ARCHER

Toni Collis, EPCC

acollis@epcc.ed.ac.uk

Version 1.1, January 29, 2016

1. Introduction

Current HPC systems compute tens to hundreds of petaFLOPs (floating point operations per second). Current systems are therefore capable of processing data quickly, but one of the key remaining bottlenecks on moderns systems is in the significant time required for reading and writing data. The supercomputing community often spends significant resources

Much of modern HPC usage is data driven. Analysis of large datasets, output of trajectory files sampled even infrequently but often enough to provide meaningful results for analysis all require efficient input and output, or I/O. Minimising the bottleneck present in software because of IO is key to improving software performance, thereby reducing the time and energy requirements to complete the problem under consideration and facilitating the completion of more science in overall for the same resources.

Input and output bottlenecks are not a new problem for HPC: the issue exists in all forms of computing as soon as it is necessary to transfer data between memory and a file. However, the problem is even greater on parallel architectures when the software can compute many millions of calculations per second and the I/O therefore needing to keep up with the volume of data that is required and produced. The supercomputing community often spends a lot of effort providing parallel computation, but I/O can often be left over as an after thought.

There are multiple methods of addressing this issue, and as discussed in a previous ARCHER white paper by Henty *et al.* [3], the choice depends on the specific problem under consideration. On ARCHER, a Cray XC30 supercomputer with a Lustre file system, full advantage of the architecture can only be taken when IO is performed in parallel.

This paper explains one particular approach to parallel IO based on the work completed in an ARCHER funded eCSE on the TPLS software package [2]: using NetCDF. There are multiple resources available online for using NetCDF, but the majority focus on software written in C. This guide aims to help users of ARCHER who have software written in modern Fortran (90/95 onwards) to take advantage of NetCDF using parallel file reads and writes.

2. Parallel IO structures

In its simplest form of software, written and executed in serial, a single process stores data, usually in an array, opens the file and writes the data file in a one stream. In parallel the problem becomes far more complex. Usually the variables that are required to be written to file are distributed among processes. How this data is written to file therefore requires additional steps to recombine the data either before or after the file is written. There are several ways of achieving this:

- 'Multiple files, multiple writers' Each process writes out its own portion of the data to a different file. This is often very simple, but in practice results in many files that need to be reconstructed during post processing. File management can also become an issue in this situation, particularly on massively parallel systems where thousands of files would result in this pattern.

- 'Single file, single writer' This is the simplest way to emulate the traditional serial code. Also known as 'master IO', a 'master' process coordinates the IO, receiving data from all other processes, rearranging as necessary to combine in to one global data set and then performing the file write. Despite being reasonable straightforward to implement this method has several disadvantages; other processes will be idle while the master performs I/O (or will idle after arriving early at the next synchronization point), the I/O time is a constant so will inhibit strong scaling due to Amdahl's law, and the need to store the global data limits the size of problem which can be studied to that which fits in memory on a single process. With this master I/O structure, each process may send many messages (for example, in the original TPLS software, each process sent 105 messages to the master process for each file write [2]), adding significant overhead.

- 'Single file, multiple writer' Multiple writers access the same file to produce on file, resulting in the data rearrangement occurring during the IO process itself. This becomes a collective operation, although communication between processes transferring data is not involved, instead tasks communicate to ensure that access to the

file does not clash. This facilitates the greater use of a parallel IO network, such as the Lustre File system, not possible in the single file, single writer mode.

- 'Single file, collective writers' In this format a subset of processors perform the IO, thus the data rearrangement is shared between the pre-writing and the writing sections of the code. Before the file write begins particular processors are identified as ones that will perform IO, gathering the required data from a small number of other processors (in modern architectures this is best done from processors on the same node sharing memory to minimise the communication overhead), and then performing a parallel file write as in the multiple writer method above.

In massively parallel systems such as ARCHER the first two methods result in either poor performance (single file, single writer), sometimes with the the inability to write all data out as the data cannot fit in the memory of one process, or large numbers of files that are difficult to manage and need significant work during post processing to be reassembled. The second two methods are ideal for massively parallel machines. Collective writers can often produce the best performance but often needs to be tuned to a particular architecture thus for ease of use, maintainability and portability we consider 'single file, collective writers' in the following example.

3. Using NetCDF in Fortran

The following provides commands for writing files in parallel using NetCDF in Fortran software using the 'single file, multiple writer' arrangement as described in section 2.. For full details please refer to the NetCDF manual [1].

To take advantage of parallel I/O features in NetCDF-4, netCDF-4/HDF5 files need to be produced and this is the method laid out below. While HDF5 does not support writing to compressed files in general, when using HDF5 behind the NetCDF interface, data is written with a defined 'chunk size' enabling the HDF5 compression algorithm to work correctly in parallel. We do not discuss changing 'chunk size' in this guide, but this can also be investigated to further improve performance.

By default parallel file access using NetCDF is ‘independent’, in that any process may access the file without waiting for others, also known as ‘single file, multiple writers’. In this mode, files are created by each process individually writing the appropriate section of a file, according to the defined data distribution. In this setup data is not transferred between processes, but synchronization is required. The alternative is a ‘collective’ file access pattern where each process must communicate with all other processes to receive and rearrange data but the data is written all at once. The ‘independent’ I/O model results in file locking when each process writes and therefore it is expected that this method does not scale, although the global processor synchronisation overhead is minimised. With the ‘collective’ method, i.e. the I/O system knows that all processes are writing, enabling optimization of the I/O operation specific to the I/O pattern and potentially increasing bandwidth. Therefore on massively parallel architectures such as ARCHER, the collective file writing pattern for I/O should result in the best scaling for file writes and is the choice given in the example below.

3.1. Writing files using NetCDF

The following code snippet provides the necessary netCDF calls for Fortran codes in 90 onwards to write a 3D array distributed, distributed in three dimensions across processors.

When distributing the array a reference must be maintained that identifies the location in the global array of the local arrays data. In this example this is done by using lower bounds (sx, sy, sz) and upper bounds (ex, ey, ez) in each of the three dimensions that is passed to the I/O routine.

It is recommended that each call to netCDF is handled for error and as such this code calls ’check’ on the returned function value. An example for the error handling code is in section 3.2. below.

NetCDF uses a creation mode (cmode) flag to open a file with the correct access type. For full details please read the netCDF manual [1]. In this example the following command from the NetCDF Fortran 90 interface to create the new I/O files:

```
nf90\_create(filename, cmode, ncid,
```

```
    comm=MPI\_COMM\_WORLD, info=MPI\_INFO\_NULL)
```

The use of the comm and info parameters ensure that parallel I/O is enabled. The variable ‘ ncid’ is an identifier set during this created command to enable future identification of the file.

The variable `cmode` is set to:

```
cmode = IOR(NF90\_NETCDF4, NF90\_MPIIO)
```

which results in the creation of a parallel HDF5/NetCDF-4 file, which will be uses MPI-IO to utilise the parallel file system.

Before starting to write into the file netCDF needs to understand the nature of the data to be written to enable parallel file access by calculating how much space to allocate to each process, thus once the file has been opened, the user must define the size of data in each dimension as defined in the example below in the array `gsizes` and the type of data by using:

```
nf90_def_var(ncid, dataname, NF90_DOUBLE, dimid, varid)
```

where NF90_DOUBLE in this instance defines the variables to be of type double. Multiple variables can be defined for each file with each call to this command returning a different `varid`.

The writing is performed by command `nf90_put_var`, that is provided with the file id (`ncid`), the id of the variables to be written (`varid`). The values to be written (`data`) may be of any type but a start position and the number need to be specified in this command.

Code example:

```
!> Output 3D array with hdf5
subroutine output_3D_hdf5(dataname,sx,ex,sy,ey,sz,ez,&
    data,nprocs_x,nprocs_y,nprocs_z,rank)
  implicit none
```

```
!Variables for file writing
character(len=*), intent(in) :: dataname !< File name.
integer,          intent(in) :: sx
integer,          intent(in) :: ex
integer,          intent(in) :: sy
integer,          intent(in) :: ey
integer,          intent(in) :: sz
integer,          intent(in) :: ez

!Array to be written (3D)
double precision, intent(in) :: data(1:(ex-sx+1),1:(ey-sy+1),1:(ez-sz+1))

integer,          intent(in) :: rank !< process ID

!Number of processors in each dimension:
integer,          intent(in) :: nprocs_x
integer,          intent(in) :: nprocs_y
integer,          intent(in) :: nprocs_z

!Netcdf variables
integer cmode, ncid, varid, dimid(3)
integer psizes(3), gsizes(3), start(3), count(3)
character(len=60) :: filename
character(len=256) ::strg
character(len=60) :: filename
character(len=256) ::strg

write(strg,*) iteration
filename = trim(dataname)//'_'//trim(strg)//'.nc'
```

```
! indicate to use  parallel 'PnetCDF' to carry out I/O
cmode = IOR(NF90_NETCDF4, NF90_MPIIO)

! Create output file
call check(nf90_create(filename, cmode, ncid, comm=MPI_COMM_WORLD,&
     info=MPI_INFO_NULL), 'creating file: ')

!Define sizes of data to write out
gsizes(1) = (ex-sx+1)*nprocs_x
gsizes(2) = (ey-sy+1)*nprocs_y
gsizes(3) = (ez-sz+1)
psizes(1) = nprocs_x
psizes(2) = nprocs_y
psizes(3) = nprocs_z

! define dimensions x, y  and z
! for dimension 'x' gsizes contains length of dimension
! and dimid is returned with a dimension ID
call check(nf90_def_dim(ncid, "x", gsizes(1), dimid(1)), &
 'In nf_def_dim X: ')
call check(nf90_def_dim(ncid, "y", gsizes(2), dimid(2)), &
 'In nf_def_dim Y: ')
call check(nf90_def_dim(ncid, "z", gsizes(3), dimid(3)), &
 'In nf_def_dim Z: ')
! define a 3D variable of type double
call check(nf90_def_var(ncid, dataname, NF90_DOUBLE, dimid, varid), &
 'In nf_def_var: ')

! exit define mode
call check(nf90_enddef(ncid), 'In nf_enddef: ')
```

```
  !Now in NETCDF Data Mode
  ! set to use MPI/PnetCDF collective I/O
  call check(nf90_var_par_access(ncid, varid, NF90_COLLECTIVE),&
       'In nf_var_par_access: ')

  !Set up start and end points for each process' data
  start(1) = sx
  start(2) = sy
  start(3) = sz+1
  count(1) = (ex-sx)+1
  count(2) = (ey-sy)+1
  count(3) = ez-sz+1

  call check(nf90_put_var(ncid, varid, data, start=start, count=count),&
       'In nf_put_vara_int: ')
  ! close the file
  call check(nf90_close(ncid), 'In nf_close: ')
  return
end subroutine output_3D_hdf5
```

3.2. Error checking

It is recommended that error handling is integrated into your netCDF calls. `NF90_CREATE` returns the value `NF90_NOERR` if no errors occurred. Possible causes of errors include:

- Passing a dataset name that includes a directory that does not exist.
- Specifying a dataset name of a file that exists and also specifying that the file should not be overwritten using `NF90_NOCLOBBER`.

- Specifying a meaningless value for the creation mode.
- Attempting to create a netCDF dataset in a directory where you don?t have permission to create files.

For this purpose the following routine can be used to check each function.

```
subroutine check(err, message)
  use netcdf
  use mpi
  implicit none

  integer err
  character(len=*) message

  ! It is a good idea to check returned value for possible error
  if (err .NE. NF90_NOERR) then
     write(*,*) "Aborting: ",trim(message), trim(nf90_strerror(err))
     call MPI_Abort(MPI_COMM_WORLD, -1, err)
  end if
```

4. Conclusions

IO performance is often an overlooked bottleneck in modern HPC software. Using netCDF is often a simple route to introduce parallel file writing into software that is portable across platforms with good performance. NetCDF provides various routines for parallel I/O, allowing all processes to write simultaneously to a single file, and stores data in a binary file format with optional compression using the HDF5 library [4].

In the work that led to writing this guide, introducing parallel IO using collective parallel HDF5 NetCDF file writes introduced an order of magnitude reduction in I/O time. Further performance improvements can be obtained by investigating the optimal striping

and chunk size, details of which are available from the ARCHER IO white paper [3] and the the NetCDF manual [1].

References

[1] NetCDF web page. http://www.unidata.ucar.edu/software/netcdf.

[2] TPLS: High Resolution Direct Numerical Simulation of Two-Phase Flows. http://sourceforge.net/projects/tpls/.

[3] David Henty, Adrian Jackson, Charles Moulinec, and Vendel Szeremi. Performance of parallel io on archer. https://www.archer.ac.uk/documentation/white-papers/parallelIO/ARCHER_wp_parallelIO.pdf.

[4] The HDF Group. Hierarchical Data Format, version 5. http://www.hdfgroup.org/HDF5/, 1997-2015.