FUTILS User's Guide

Trach-Minh Tran, Ben McMillan, Stephan Brunner, Paolo Angelino, CRPP/EPFL. v1.2, May 2014

A quick and simple way to start using HDF5 file formats for Fortran programmers.

Contents

1 Introduction

1.1 What is HDF5

HDF5 or *Hierarchical Data Format 5* is a library callable from C/C++ and Fortran to store scientific data in a portable way. Two primary objects form the basis of HDF5:

- Groups are structures for organizing objects (others groups or datasets) in a HDF5 file
- Datasets are essentially multidimensional arrays of data elements

By analogy with a filesystem, groups can be considered as *directories* and can contain others groups while datasets are simply the files.

The main advantages of HDF5 files are:

- Free, open source software
- Portability across different platforms (Unix, Windows, Mac OSX, big/little endian, ...)
- Many existing access (h5ls, h5dump, hdfview) and visualization (Matlab, Python, OpenDX, VTK, ...) tools
- Parallel IO

1.2 What is FUTILS

FUTILS is a module of Fortran *wrapper* routines which call the low level HDF5 routines. Its main purpose is to *simplify* the creation and manipulation of HDF5 files for some special types of data found in *diagnostic* or *restart* files produced by time-dependent simulation codes.

The main features of the current version of FUTILS can be summarized as follow:

- Serial and parallel modes, using serial and parallel HDF5.
- Write and read integer, single/double precision real/complex arrays of dimensions up to 4.
- Fixed and extendible (on the last dimension) arrays.
- Write and read text and binary files.
- Data can be compressed.
- In parallel mode, arrays can be partionned on a processor grid of dimension up to 4d, using MPI cartesian topology (required when the partionning is more than one dimensional).
- Local arrays with *ghost area* in parallel mode.

2 Compiling programs with FUTILS

Here are the different steps to build FUTILS.

2.1 Obtaining the HDF5 software

The pre-built binaries exist for many platforms and can be downloaded from *http://www.hdfgroup.org/HDF5/release/obtain5.html*. But most of these pre-built libraries do not include the *parallel* HDF5. Moreover, these libraries might use a different Fortran compiler than the one you are using. In such cases, it is better to grab the source code and build the library with the following `configure` and `make` commands, using the MPI `C` and `Fortran` wrappers `mpicc` and `mpif90`:

```
export F9X=mpif90
export CC=mpicc
VERSION=1.6.5
PREFIX=/usr/local/hdf5-$VERSION
./configure --prefix=$PREFIX \
      --enable-fortran \
      --enable-parallel \
      --disable-shared \
    2>&1 | tee configure.log
  make 2>&1 | tee make.log
  make install 2>&1 | tee  -a make.log
```

In addition to a MPI library with MPI-IO capability, the built of HDF5 requires the compression library *ZLIB* `<http://www.zlib.net>`.

In the case you don't have MPI installed in your system or don't use the *parallel HDF5*, set the environment variables F9X and CC to your compilers and build the serial version of HDF5 *without* the `-enable-parallel` configure option.

2.2 Building FUTILS

FUTILS can be built by checking out the source from the `crppsvn` repository and simply running `make` as follows:

```
svn co http://crppsvn.epfl.ch/repos/Utils/hdf5/futils/trunk futils
cd futils/src
make lib
```

The resulting files are the module file `futils.mod` required for the compilation of program units that contains `USE FUTILS` and the library `libfutils.a` which include others utilities in addition to the FUTILS module. If you don't have `MPI` installed in your system or if you want to use only the *serial* HDF5, you should use the file `Makefile_serial` to build the library:

```
make -f Makefile_serial lib
```

Assuming that the newly built `futils.mod, libfutils.a` (and `libmpiuni.a` required to use the serial version) are in the directory `$FUTILS` and the parallel and serial HDF5 are installed respectively in `/usr/local/hdf5/lib` and in `/usr/local/hdf5_serial/lib`, you can compile your program in one of the 3 following ways:

```
# MPI Programs using Parallel HDF5
mpif90 -I${FUTILS} -I/usr/local/hdf5/lib -c myprog.f90
mpif90 -L${FUTILS} -L/usr/local/hdf5/lib myprog.o -lfutils -lhdf5_fortran -lhdf5 -lz

# MPI Programs using serial HDF5
mpif90 -I${FUTILS} -I/usr/local/hdf5_serial/lib -c myprog.f90
mpif90 -L${FUTILS} -L/usr/local/hdf5_serial/lib myprog.o -lfutils -lhdf5_fortran -lhdf5 -lz

# Serial Programs using Serial HDF5
ifort -I${FUTILS} -I/usr/local/hdf5_serial/lib -c myprog.f90
ifort -L${FUTILS} -L/usr/local/hdf5_serial/lib myprog.o \
           -lfutils -lhdf5_fortran -lhdf5 -lz -lmpiuni
```

Note that when your MPI program calls only serial HDF5 routines from a *single* processor, you can use both versions of HDF5, compiled either using `make lib` or `make -f Makefile_serial lib`. You can find examples of `Makefile` for others platforms in `futils/src`.

3 Quick-start with examples

For the impatient, several examples are presented here to show how to quickly start to use the FUTILS routines. More examples can be found in `futils/src/`.

3.1 Example 1: Save a 2d spatial profile together with the grid coordinates

In this example, a new HDF5 file is created and contain the 1d arrays for the 2 coordinates X(1:NX) and Y(1:NY), and a 2d array for the grid values of the potential POT(1:NX,1:NY). The data are organized in the following structure:

- The dataset `/coordinateX`
- The dataset `/coordinateY`
- The group `/2D_profiles`
  - The dataset `/2D_profiles/Potential`

The group `/2D_profiles` serves to *group* any others profiles (such as density, velocity, ...) defined on the same X-Y grid! The following minimalist (and yet complete) Fortran program creates such a HDF5 file:

```
PROGRAM main
!
! Save a 2d spatial profile together with the grid coordinates
!
  USE futils
  IMPLICIT NONE
  INTEGER, PARAMETER :: NX=32, NY=20
  DOUBLE PRECISION :: x(NX), y(NY), pot(NX,NY)
  CHARACTER(len=32) :: file='ex11.h5'
  INTEGER :: i, fid
!
!   Define the arrays x, y and pot
!
  x = (/ (i-1, i=1,NX) /)
  y = 10. * (/ (i-1, i=1,NY) /) - 100
  CALL RANDOM_NUMBER(pot)
!
!   Create and fill the HDF5 file
!
  CALL creatf(file, fid)
  CALL putarr(fid, '/coordinateX', x)
  CALL putarr(fid, '/coordinateY', y)
  CALL creatg(fid, '/2D_profiles')
  CALL putarr(fid, '/2D_profiles/Potential', pot)
  CALL closef(fid)
!
END PROGRAM main
```

The code is pretty self-explanatory. Note that `creatf` returns a *file identifier* which is used subsequently to refer to the created file. It is possible to work simultaneously with several HDF5 files, each of which is referred by its `fid`.

The resulting HDF5 file is named `ex11.h5` and its content can be visualized quickly using the command line utilities **h5ls** and **h5dump** that are included in the HDF5 software:

```
crpppc231:src$ h5ls -r ex11.h5
/2D_profiles             Group
/2D_profiles/Potential   Dataset {20, 32}
/coordinateX             Dataset {32}
/coordinateY             Dataset {20}
crpppc231:src$
crpppc231:src$ h5dump -d /coordinateX ex11.h5
HDF5 "ex11.h5" {
DATASET "/coordinateX" {
   DATATYPE  H5T_IEEE_F32LE
   DATASPACE  SIMPLE { ( 32 ) / ( 32 ) }
   DATA {
   (0): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
   (20): 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31
   }
}
}
```

One can also use the graphical **HDFView** to browse the file more comfortably (see next example). It is available from *http://www.hdfgroup.org/hdf-java-html/hdfview* for many architectures, including Linux, Mac OSX and Windows.

It is important to note that the datasets created in this example **could not be overwritten**! Attempt to reopen the file and write to the dataset `/2D_profiles/Potential` for example will result in an error. However it is perfectly legal to add another dataset with a different name to an existing HDF5 file as follows:

```
CALL openf(file, fid)
CALL putarr(fid, '/2D_profiles/Kinetic', kin)
CALL closef(fid)
```

3.2 Example 2: Save 0d history arrays

The datasets created in Example 1 with a single **putarr** has a **fixed** array shape and could not be extended. In cases where we want to store for example, a time history of some 2d profile F(x,y,t) where the *time* dimension t can grow, the **extendible dataset** could be used. It is first created with an initial call to **creatd** where the shape of the fixed (space) dimensions is specified followed by repeated calls to **append** to insert the data. The following example shows how the time evolution of the two scalar (0d) quantities, `ekin, epot` are stored, together with the times:

```
PROGRAM main
!
!   Save 0d history arrays with buffering
!
  USE futils
  IMPLICIT NONE
  CHARACTER(len=32) :: file='ex13.h5'
  INTEGER :: fid, n, istep, ibuf, nrun=120
  INTEGER :: rank, dims(1)
  INTEGER, PARAMETER :: BUFSIZE=20
  DOUBLE PRECISION :: buf(BUFSIZE, 0:2) ! To store hist. arrays for scalars
  DOUBLE PRECISION :: time, ekin, epot
!===========================================================================
!                   1. Prologue
!
  CALL creatf(file, fid)
  CALL creatg(fid, "/var0d")
  rank = 0
  CALL creatd(fid, rank, dims, "/var0d/time")
  CALL creatd(fid, rank, dims, "/var0d/ekin")
  CALL creatd(fid, rank, dims, "/var0d/epot")
!===========================================================================
!                   2. Time loop
!
  ibuf=0
  DO istep=1,nrun
     time = istep
     ekin = COS(0.2*time)*EXP(0.01*time)
     epot = SIN(0.2*time)*(1.0-EXP(0.01*time))
!
     ibuf = ibuf+1
     buf(ibuf,0) = time
     buf(ibuf,1) = ekin
     buf(ibuf,2) = epot
     IF( ibuf.EQ.BUFSIZE .OR. istep.EQ.nrun) THEN ! Dump the buffers to file
        CALL append(fid, "/var0d/time", buf(1:ibuf,0))
        CALL append(fid, "/var0d/ekin", buf(1:ibuf,1))
        CALL append(fid, "/var0d/epot", buf(1:ibuf,2))
        ibuf = 0
     END IF
  END DO
!===========================================================================
!                   9. Epilogue
!
  CALL closef(fid)
END PROGRAM
```

Some remarks:

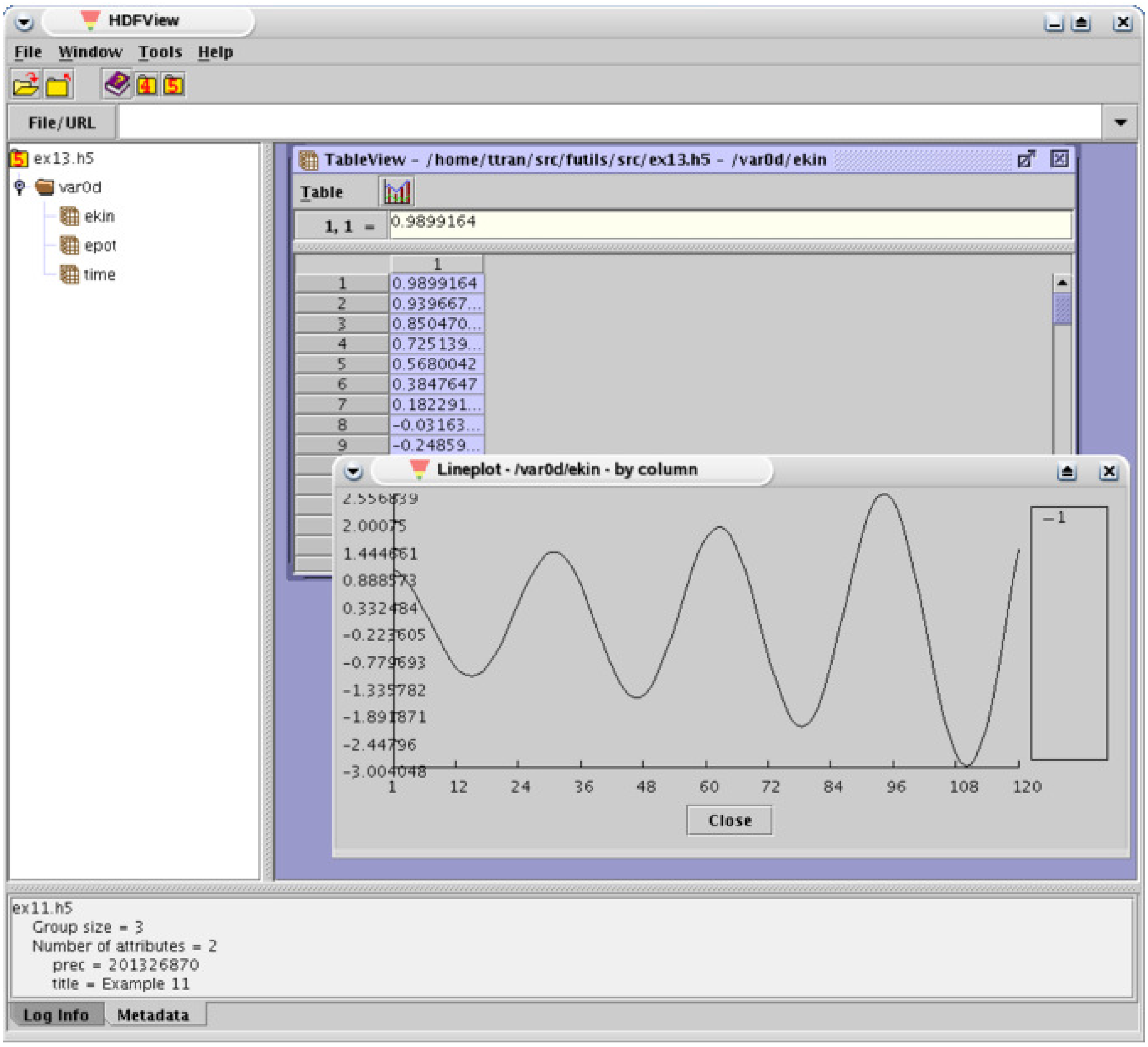
- For each extendible dataset, the call to **creatd** specifies the number of (fixed) dimensions `rank` and the shape (number of elements in each dimension) in the 1d array `dims`. Here, `rank=0` and the argument `dims` will not be referred by **creatd**.
- To minize the number of transfers to disk, the data are stored in buffers of size BUFSIZE which are flushed to disk only every BUFSIZE steps or at the last istep of the time loop.
- For 1d or higher dimensions, this buffering might not be nessary.
- In a *restart* run, the HDF5 file will be open with a call to `openf` followed by the sequence of calls to `append`, exactly as in the *time loop* in the program show above.

- An alternative to the *extendible* dataset (for higher dimensions, `rank` > 0) is to create a sequence of *fixed* dimension datasets with names suffixed by the time step as in the following example:

  ```
  WRITE(name,'(a,i3.3)') 'pot.', istep
  CALL putarr(fid, name, pot)
  ```

  which will create the datasets `pot.001, pot.002, ...` for `istep=1, 2, ....`

The resulting HDF5 file can be viewed using HDFView as shown in the following figure.



TableView - /home/ttran/src/futils/src/ex13.h5 - /var0d/ekin

1, 1 = 0.9899164

	1
1	0.9899164
2	0.939667...
3	0.850470...
4	0.725139...
5	0.5680042
6	0.3847647
7	0.182291...
8	-0.03163...
9	-0.24859...

```
ex11.h5
  Group size = 3
  Number of attributes = 2
    prec = 201326870
    title = Example 11
```

Figure 1: The HDFView window

3.3 Example 3: Files in datasets

Files can be stored as a HDF5 dataset by calling **putfile**. It is thus possible to store all the input files and even the program sources together with the results inside a single self-contained HDF5 file! Note that *binary* can also be stored with **putfile**. Here is a simple example which stores its own source file in a dataset which has the same name:

```
PROGRAM main
!
!   Store myself in a HDF5 file
!
  USE futils
  IMPLICIT NONE
  CHARACTER(len=256) :: file='ex14.h5'
  INTEGER :: fid
!
  CALL creatf(file, fid)
  CALL putfile(fid, '/ex14.f90', 'ex14.f90')
  CALL closef(fid)
END PROGRAM main
```

To retrieve the file, use the program in `getfile.f90` which is included in the FUTILS source tree.

3.4 Example 4: Writing a distributed matrix in a MPI program

Assume that a matrix is distributed by columns across P MPI processes. The following example shows the **collective** creation of the (single) HDF5 file by all the P processes:

```
PROGRAM main
!
!   Parallel write a 2d array
!
  USE futils
  IMPLICIT NONE
  INCLUDE "mpif.h"
  CHARACTER(len=32) :: file='pex10.h5'
  INTEGER, PARAMETER:: nx=5, nyp=2
  INTEGER :: ierr, fid, me, start, i, j
  DOUBLE PRECISION :: array(nx,nyp)
!
!   Init MPI
  CALL mpi_init(ierr)
  CALL mpi_comm_rank(MPI_COMM_WORLD, me, ierr)
!
!   Define the local array
  start = me*nyp
  DO i=1,nx
     DO j=1,nyp
        array(i,j) = 10*i + (start+j)
     END DO
  END DO
!
!   Create file collectively
  CALL creatf(file, fid, mpicomm=MPI_COMM_WORLD)
  CALL putarr(fid, '/matrix', array, pardim=2)
  CALL closef(fid)
!
```

```
  CALL mpi_finalize(ierr)
END PROGRAM main
```

Note that the **parallel** version uses the same FUTILS routines, but with the additional arguments `mpicomm` and `pardim` in **creatf** and **putarr** respectively.

With 4 MPI processes, the resulting file `pex10.h5` should contain the global 5x8 matrix. This can be easily checked using **MATLAB-7.3**:

```
>> file = 'pex10.h5';
>> mat = hdf5read(file, '/matrix');
>> mat

mat =

    11    12    13    14    15    16    17    18
    21    22    23    24    25    26    27    28
    31    32    33    34    35    36    37    38
    41    42    43    44    45    46    47    48
    51    52    53    54    55    56    57    58

>>
```

Note that with **MATLAB-7.1**, `hdf5read` *transposes* the original matrix, which is incorrect!

In this example it was assumed that the total number of columns is a multiple of the number of processes, with constant nyp. This is however **not** required by FUTILS: each process can have different nyp and FUTILS will automatically detect it through the size along the dimension **pardim** of the input array.

In the next example, a 3d array is distributed on a 2d cartesian 2x4 processor grid. To write *collectively* this partionned array, a communicator with a *cartesian topology* defined on it should be passed to the file creation routine and the "nd" version of `putarr` should be used.

```
PROGRAM main
!
!   Parallel write a 3d array partionned on 2d processor grid:
!   A(n1/P1, n2/P2, n3).
!
  USE futils
  IMPLICIT NONE
  INCLUDE "mpif.h"
  CHARACTER(len=32) :: file='para.h5'
  INTEGER :: ierr, fid, me, npes
  INTEGER, PARAMETER :: ndims=2
  INTEGER, PARAMETER ::  n1p=3, n2p=2, n3=2  ! Dimension of local array
  REAL, DIMENSION(n1p,n2p,n3) :: array
  INTEGER, DIMENSION(ndims) :: dims, coords
  LOGICAL :: periods(ndims), reorder
  INTEGER :: cart, i, j, k, iglob, jglob
!
!   Init MPI
  CALL mpi_init(ierr)
  CALL mpi_comm_size(MPI_COMM_WORLD, npes, ierr)
  CALL mpi_comm_rank(MPI_COMM_WORLD, me, ierr)
```

```
!
!   Create cartesian topology
!
  dims    = (/2, 4/)
  periods = (/.FALSE., .TRUE./)
  reorder = .FALSE.
  IF( PRODUCT(dims) .NE. npes ) THEN
     IF( me .EQ. 0 ) THEN
        PRINT*,  PRODUCT(dims), " processors required!"
        CALL mpi_abort(MPI_COMM_WORLD, -1, ierr)
     END IF
  END IF
  CALL mpi_cart_create(MPI_COMM_WORLD, ndims, dims, periods, reorder, cart, ierr)
  CALL mpi_cart_coords(cart, me, ndims, coords, ierr)
!
!   Define local array
!
  DO i=1,n1p
     iglob = coords(1)*n1p + i
     DO j=1,n2p
        jglob = coords(2)*n2p + j
        DO k=1,n3
           array(i,j,k) = 100*iglob + 10*jglob + k
        END DO
     END DO
  END DO
!
!   Create file collectively, passing the comm. with cartesian topology
!
  CALL creatf(file, fid, mpicomm=cart)
!
!   Write to file collectively using "nd" version of "putarr".
!
  CALL putarrnd(fid, '/parray', array, (/1,2/))
!
!   Lean up and quit
!
  CALL closef(fid)
  CALL mpi_finalize(ierr)
!
END PROGRAM main
```

It was assumed that the size of the partitionned local dimensions is a multiple of the number of processors along the corresponding processor grid dimension. As in the non "nd" version, this is **not** required. For local arrays with *ghost area*, see example `pex11.f90`. The resulting array `A(6,8,2)` stored in the HDF5 file can be checked using MATLAB:

```
>> hdf5read('para.h5','/parray')

ans(:,:,1) =
     111   121   131   141   151   161   171   181
     211   221   231   241   251   261   271   281
     311   321   331   341   351   361   371   381
     411   421   431   441   451   461   471   481
     511   521   531   541   551   561   571   581
     611   621   631   641   651   661   671   681


  ans(:,:,2) =

     112   122   132   142   152   162   172   182
     212   222   232   242   252   262   272   282
     312   322   332   342   352   362   372   382
     412   422   432   442   452   462   472   482
     512   522   532   542   552   562   572   582
     612   622   632   642   652   662   672   682

  >>
```

3.5 Example 5: Writing a section of a timeslice

It is possible to write less than a full timeslice to a dataset using the routine `append` with the argument offset. This may be useful when it is impractical to store the whole timeslice of a diagnostic in memory as an array. In this case, the dataset is not extended automatically and `extend` must be called before data may be written. Also, for optimum performance, when `creatd` is called, the optional argument `chunking` should be set equal to the size of the arrays to be written.

```
PROGRAM main
!
! Save a large 2d array without buffering the whole array.
!
  USE futils
  INTEGER, PARAMETER :: NX=10000,NY=10000, NT = 5
  DOUBLE PRECISION   :: pot_column(1,NY)
!=====================================================================
!                  1. Prologue
!
  CALL creatf(file, fid)
  CALL creatd(fid, 2, (NX,NY), "potential",chunking=(1,NY) )
!=====================================================================
!                  2. Time loop
!
  DO t =1,NT
     extend(fid, "potential", 1)
     DO ix=1,NX
        DO iy=1,NY
           pot_column(iy) = (ix + iy*iy)*t
        END DO
        append(fid, "potential", pot_column, offset = (ix-1,0) )
     END DO
  END DO
!=====================================================================
!                   9. Epilogue
!
  CALL closef(fid)
END PROGRAM
```

3.6 Miscellanies

We list briefly here the features which are not mentionned in the examples above. Their detailed description will be given in the Reference Manual section.

- Attributes (or properties) can be *attached* to groups and datasets. This can be an character argument added to the creation routines (creatf, creatg, creatd and putarr) to give a short description or by calling **attach** on existing objects.
- Routines to read datasets (getarr) and attributes (getatt).
- In the present version, the maximum rank of arrays is 3. This can be increased in a future version.
- By default, the real type is in **single precision** with 32 bits. This can be changed to 64 bits at the file *creation* by adding `real_prec='d'` in `creatf`.
- **Complex type** is implemented using the HDF5 *compound type*. The MATLAB high-level function `hdf5read` reads such dataset into arrays of *cells*. The script `src/h5Complex.m` shows how to read and convert these arrays into MATLAB `complex` arrays.

4 The HASHTABLE module

The HASHTABLE module provides a convenient means to buffer 0d quantities before they are written into an HDF5 file. Such buffering might be necessary if collecting and writing these 0d quantities at each timestep was found to be a performance bottleneck. The HASHTABLE module also allows collective sums to be performed on data to be output from an MPI code. The module was designed to maintain the elegant interface of FUTILS, especially the concise keyword-based single line of code calling sequence.

As an example, we recode the example of section 3.2 (Save 0d history arrays) using the HASHTABLE module:

```
PROGRAM main
!
! Save 0d history arrays with buffering using the HASHTABLE module
!
USE futils
USE hashtable
IMPLICIT NONE
  include 'mpif.h'
  CHARACTER(len=32) :: file='ex13.h5'
  INTEGER :: fid, n, istep, nrun = 120, me_world,ierr
  DOUBLE PRECISION :: time, ekin, vel
  INTEGER, PARAMETER :: BUFSIZE = 20
  TYPE(BUFFER_TYPE) :: hbuf
!=======================================================================
!                   1. Prologue
```

```
!
  CALL MPI_INIT(ierr)
  CALL mpi_comm_rank(MPI_COMM_WORLD, me_world, ierr)
  CALL htable_init(hbuf,BUFSIZE)
  IF(me_world==0) THEN
     CALL creatf(file, fid)
     CALL creatg(fid, "/var0d")
     CALL set_htable_fileid(hbuf,fid)
  END IF
!===========================================================================
!                  2. Time loop
!
  DO istep=1,nrun
     time = istep
     ekin = COS(0.2*time)*EXP(0.01*time)
     vel  = SIN(0.2*time)*(1.0-EXP(0.01*time))
!
     CALL add_record(hbuf,"time",  "simulation time", time)
     CALL add_record(hbuf,"ekin",  "kinetic energy",  ekin, MPI_COMM_WORLD)
     CALL add_record(hbuf,"maxvel","maximum velocity",vel,  MPI_COMM_WORLD,MPI_MAX)
     CALL htable_endstep()
  END DO
!===========================================================================
!                  9. Epilogue
!
  CALL htable_hdf5_flush()
  IF (me_world==0) CALL closef(fid)
END PROGRAM
```

5 The vis3d module

The vis3d module enables writing 3D data in HDF5 based formats readable from 3D visualization softwares as Paraview and VisIt.

5.1 Features

- parallel HDF5
- Meshes:
  - Toroidal meshes:
    - s, chi, phi mesh
    - field aligned mesh
  - volume selection (select one, custom resolution, 3D slice to output)
  - custom mesh size
- Data type (4Bytes, 8Bytes)
- Output formats:

- XDMF
  - 1 file x time step, x output field
    - 1 file x output field
  - PIXIE (parallel read in VisIT)
    - 1 file x time step, x output field
    - 1 file x output field
    - 1 file x time step, all output fields
    - 1 file, all time steps, all fields

5.2 Module vis3d API

The interface between the main code and the vis3d module is defined by three subroutine calls:

- module initialization
- write a 3D field
- check out of the 3D files

The following code exemplifies the three steps:

```
PROGRAM main
  !
  !   Parallel write a 2d array
  !
  USE futils
  USE vis3D
  IMPLICIT NONE
  !
  ! Main program variables
  ! ...
  !
  ! Derived type containing the parameter for the 3D module
  TYPE(DIAG3D_PARAMETERS) :: param3d
  !
  ! Pointer to the function returning the 3D field (optional)
  PROCEDURE(get_field_func), POINTER :: get_field_ptr
  !
  !============================================================================
  !                             1.  Prologue
  ! ...
  !
  !============================================================================
  !                             2.  Initialize 3d module
  !
  ! set parameters for 3d output
  param3d = ...
  !
  CALL init3Ddiagnostics(param3d)
  !
  ! ...
  !
  ! Main loop
  DO time = 0, end_time
    !
    ! ...
    !
    !==========================================================================
    !                             3.  Write 3d output
    !
    get_field_ptr => get_field_function
    CALL write3d('field_name', time, field_function_ptr=get_field_ptr)
    !
    ! ...
    !
  END DO
  !
  ! ...
  !
  !==========================================================================
  !                             4.  Epilogue
  !
  CALL checkout3dfile('debug')
  !
END PROGRAM main
```

5.2.1 Module initialization

The subroutine init3Ddiagnostics initializes the module:

```
SUBROUTINE init3Ddiagnostics(parameters)
TYPE(DIAG3D_PARAMETERS), INTENT(IN) :: parameters
```

The parameters to the vis3d module are passed through the derived type DIAG3D_PARAMETERS:

```
 TYPE DIAG3D_PARAMETERS
     CHARACTER(LEN=2) :: mesh_type        ! sc: s, chi, phi; fa: field aligned
     CHARACTER :: data_type               ! s: single precision; d: double precision
     CHARACTER(LEN=16) :: output_format   ! format of the output files
     INTEGER, DIMENSION(3) :: mesh_size   ! size of the output mesh
     DOUBLE PRECISION, DIMENSION(2) :: slim   ! slice in s direction
     DOUBLE PRECISION, DIMENSION(2) :: chilim ! slice in chi direction
     DOUBLE PRECISION, DIMENSION(2) :: philim ! slice in phi direction
     LOGICAL :: forcelocal            ! grid is forced to be local to the processors
     PROCEDURE(coordf), POINTER, NOPASS :: coordf_ptr ! Pointer to the
! coordinate transform function
     INTEGER :: comm_loc              ! Communicator of distributed 3d data
     INTEGER :: nphip                 ! # of grid points in the distributed dimension per procs
     LOGICAL :: nlres                 ! Flag for restart runs
     LOGICAL :: verbose               ! Write 3D module messages to std output?
  END TYPE DIAG3D_PARAMETERS
```

5.2.2 Write 3D output

The subroutine write3D is called to write a 3D field into the formatted file specified in the parameters:

```
SUBROUTINE write3D(out_field, ztime, field_array_3d, field_function_ptr, filename)
    CHARACTER(*), INTENT(IN) :: out_field
    INTEGER, INTENT(IN) :: ztime
    DOUBLE PRECISION, DIMENSION(:,:,:), INTENT(IN), TARGET, OPTIONAL :: field_array_3d
    PROCEDURE(get_field_func), POINTER, OPTIONAL :: field_function_ptr
    CHARACTER(len=128), INTENT(OUT), OPTIONAL :: filename     !! Output file name
```

out_field: the name of the field to be written.

ztime: the time at which the field is evaluated.

field_array_3d: a 3D array containing the output field, in the curvilinear coordinates.

field_function_ptr: a function returning the field value at a given point (specified by curvilinear coordinates).

filename: the name of the file is returned by the subroutine.

The field to be written can be passed to the vis3d module as a 3D array or as a function returning the value of the field at point given in curvilinear coordinates. The function must have the following interface:

```
  INTERFACE
     SUBROUTINE get_field_func(x, y, z, field)
       DOUBLE PRECISION, INTENT(IN) :: x, y, z
       DOUBLE PRECISION, INTENT(INOUT) :: field
     END SUBROUTINE get_field_func
  END INTERFACE
```

5.2.3 3D files checkout

The routine *checkout3dfile* must be called for each written field, at the end of the run, in order to correctly close the 3D files.

```
  SUBROUTINE checkout3dfile(out_field)
  CHARACTER(*), INTENT(IN) :: out_field
```

out_field: the name of the 3D field written, as in the call to the subroutine write3D.

5.3 Parameter description

CHARACTER(LEN=2) :: mesh_type

This parameter control the type of mesh of the 3D output. Possible values are: 'sc', for s, χ, φ mesh; 'fa', for a field aligned mesh.

NB: the field aligned mesh results in very poor visualization quality. This is due to the stretch of the mesh cells for high q and s values, which makes very hard for the 3D visualization software to interpolate the values between the mesh point.

The choice of this type of mesh is recommended for off-line data analysis only.

CHARACTER :: data_type

This parameter selects single ('s') or double ('d') precision output. For visualization pourposes single precision is enough and it allows to reduce the file size by a factor of 2.

CHARACTER(LEN=16) :: output_format

This parameter selects the format of the 3D output files. The main format types are PIXIE and XDMF. Both are based on parallel HDF5 write. The PIXIE format is the most versatile and compact and should be preferred.

NB: XDMF does not support the writing of fluid moments. XDMF_one does not support restarts.

The PIXIE format

PIXIE files are generated by software at Los Alamos (related to Polar Ionospheric X-Ray Imaging Experiment, PIXIE). Data format information is stored as attributes in the HDF5 file.

VisIt's Pixie reader was written with a high degree of generality in mind. It makes few assumptions regarding the names and/or locations of HDF5 objects in the file and can often be used successfully to read arbitrary, HDF5 arrays of data.

Within the PIXIE format, you can actually choose four different output schemes, according to the data size and type of analysis. For example, having all fields groupped in the same file enable VisIt to combine them with several expressions, ex. sum, correlation, etc... On the other hand, in the case of a large grid size, it may be interesting to save the different fields or time steps in separate files, to keep the files to a reasonable size.

'PIXIE_many': output fields are written in separate files, the time steps are also written in separate files. The full time sequence is read in VisIt with the option *File grouping: smart,* in the open file dialog.

File naming convention: *nemorb_3d_fieldname_timestep.h5*

'PIXIE_one': output fields are written in separate files, all time steps are groupped in the same file.

File naming convention: *nemorb_3d_fieldname.h5*

'PIXIE_only': output fields are all written in a single file, all time steps are groupped in the same file.

File naming convention: *nemorb_3d.h5*

'PIXIE_step': output fields are all written in a single file, the time steps are split in separate files. The full time sequence is read in VisIt with the option *File grouping: smart,* in the open file dialog.

File naming convention: *nemorb_3d_timestep.h5*

The XDMF format

XDMF uses XML to describe the data format. HDF5 is used to store heavy data. This allows tools to parse XML to determine the resources that will be required to access the heavy data.

When you choose this format, in the output directory you will find two kind of files: *.xmf and *.h5. the xmf file is the one that has to be actually opened by the visualization tool (either VisIt or Paraview).

Within the XDMF format, you can actually choose two different output schemes:

'XDMF_many': output fields are written in separate files, the time steps are also written in separate files. The full time sequence is read in VisIt with the option *File grouping: smart,* in the open file dialog.

File naming convention: *nemorb_3d_fieldname_timestep.**xmf***,
*nemorb_3d_fieldname_timestep.**h5***

'XDMF_one': output fields are written in separate files, all time steps are groupped in the same file. *NB: XDMF_one does not support restarts.*

File naming convention: *nemorb_3d_fieldname.**xmf***,
*nemorb_3d_fieldname.**h5***

INTEGER, DIMENSION(3) :: mesh_size

An arbitrary mesh size can be given, independently from the solver grid size.

All quantities are defined on the grid nodes, which are located in the middle of the intervals used for particles binning.

WARNING: when the number of toroidal grid points (φ direction) is not consistent with the parallelization (size of the *cart* communicator), a global sum is used to gather all data to the process of rank 0, which then writes all the output. The global sum and no parallel I/O result in a much slower execution time. A warning is issued to the std output. Example:

nvp_cart = 8, mesh_size = 32 64 21

```
 ###################################################################
 WARNING: 3D phi mesh is not consistent with cartesian communicator
 WARNING: 3D ouput will not be parallel
 Check on mesh size:  F
     Mesh size:           21 ; Number of domains:            8
 Chech on mesh lower bound:  T
     Mesh lower bound:   0.000000000000000E+000 ; Required: 0
 Chech on mesh upper bound:  T
     Mesh upper bound:    6.28318530717959      ; Required: 2PI
 ###################################################################
```

An exception is when the number of toroidal grid points is an integer fraction of *nvp_cart.* In this case, a new communicator is defined, which is a subset of the *cart* communicator. The output is parallelized on this new communicator. Example:

nvp_cart = 16, mesh_size = 32 64 8

A warning is also issued to the std output:

```
 ###################################################################
 WARNING: 3D phi mesh is a subset of code grid
 WARNING: 3D communicator redefined
 3D communicator domains:            8
Selected ranks (from cart communicator) :    0    2    4    6    8
10   12   14
 ###################################################################
```

DOUBLE PRECISION, DIMENSION(2) :: slim, chilim, philim

These parameters set lower and upper limits of the mesh grid, in order to output only a given 3D slice of the torus.

Example:

```
philim = 0 1.57
chilim = 0 4.73
slim = 0.4 0.8
```

The number of mesh nodes is still provided by **mesh_size**, but all the nodes lay now withing the given limits.

WARNING: setting philim other than $[0, 2\pi]$ is not consistent with the parallelization. As stated previously, this case is handled by gathering all data to rank 0 and writing a serial output, thus slowing down the code execution. A warning is issued to the std output:

```
 ###################################################################
 WARNING: 3D phi mesh is not consistent with cartesian communicator
 WARNING: 3D ouput will not be parallel
 Check on mesh size:  T
     Mesh size:           32 ; Number of domains:            8
 Chech on mesh lower bound:  T
     Mesh lower bound:   0.000000000000000E+000 ; Required: 0
 Chech on mesh upper bound:  F
     Mesh upper bound:    1.57000000000000      ; Required: 2PI
 ###################################################################
```

LOGICAL :: forcelocal

If *forcelocal* = *TRUE*, *philim* and φ mesh size are adjusted to be consistent with the parallelization.

philim is adjuted to the closer toroidal slice consistent with parallelization and a new communicator is defined for the parallel output.

From the previous example:

philim = [0 1.57] => philim [0.000 1.374]

mesh_size(φ) 32 => 8

A WARNING is issued to std output:

```
###################################################################
WARNING: 3D phi mesh is a subset of code grid
WARNING: 3D communicator redefined
3D communicator domains: 2
Selected ranks (from cart communicator) : 0 1
###################################################################
```

PROCEDURE(coordf), POINTER, NOPASS :: coordf_ptr

A function must be privided which transform from the curvilinear coordinates of the mesh to the cartesian coordinates used by the visualization software.

The function receives the three curvilinear coordinates as input and returns the corresponding cartesian coordinaters as from the following interface:

```
  INTERFACE
     SUBROUTINE coordf(st, chit, phit, xt, yt, zt)
       DOUBLE PRECISION, INTENT(IN)    :: st, chit
       DOUBLE PRECISION, INTENT(INOUT) :: phit
       DOUBLE PRECISION, INTENT(OUT)   :: xt, yt, zt
     END SUBROUTINE coordf
  END INTERFACE
```

INTEGER :: comm_loc

Communicator of the distributed 3d data.

INTEGER :: nphip

Number of grid points in the distributed dimension per procs.

LOGICAL :: nlres

Restart information is stored in the 3d files for the PIXIE format. This parameter indicate to the module if the actual run output must be appended after the previus run data.

Restart is not supported in the XDMF format.

LOGICAL :: verbose

If *verbose = .TRUE.*, extra output is added to the standard output. The value .TRUE. is usually passed only to the processor of rank 0.

5.4 3D module options summary

A summary of the parameters and options set for the 3D output is printed on the standard output, just after the diagnostic initialization.

It looks like:

```
##################################################################
3D output parameters
Output format: PIXIE_only
Output format type: PIXI
Mesh type(sc: s, chi, phi; fa: field aligned): sc
Data type( s: single precision; d: double precision): s
Mesh size: 32x64x4
s mesh boundaries: 0.400 - 0.800
chi mesh boundaries: 0.000 - 4.730
phi mesh boundaries: 0.000 - 1.178
Force parallelized grid: T
The grid is parallelized: T
```

```
Number of slices per procs: 1
#####################################################################
```

5.5 Example

The file examples/pex3D.f90 tests the module by writing a 3D field in a simplified toroidal geometry. The expected output is shown in fig. 2.

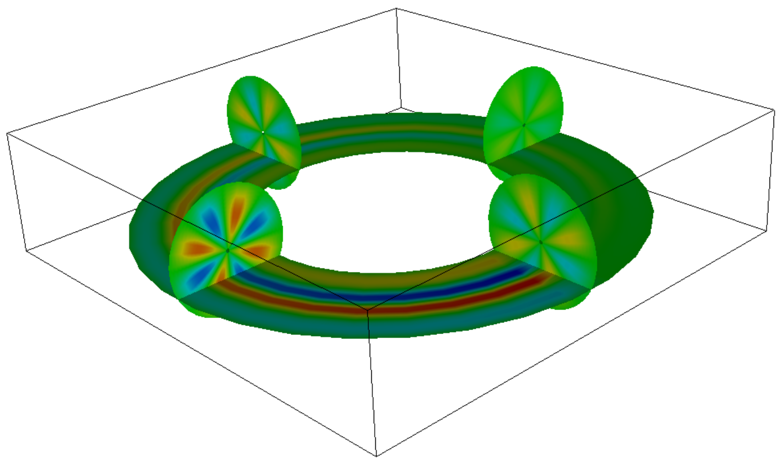


Figure 2: 3D debug field from pex3d.f90 example. Plotted with VisIt.

6 Reference Manual for FUTILS

The following conventions are adopted in the following routine description:

- <TYPE> in the declaration of an argument means that the argument can be a scalar or an array of rank 1 to 4 and
- of types integer (I), single precision real (SP), double precision real (DP), complex (C), double complex (Z), logical (L) or character (S).

6.1 allatts

```
  SUBROUTINE allatts(fid, name, attnames, atttypes, attsizes)
    IMPLICIT NONE
    INTEGER, INTENT(in) :: fid
    CHARACTER(len=*), INTENT(in) :: name
    CHARACTER(len=*), DIMENSION(:), INTENT(out) :: attnames
    CHARACTER(len=1), DIMENSION(:), INTENT(out) :: atttypes
    INTEGER(SIZE_T), DIMENSION(:), INTENT(out) :: attsizes
```

Purpose:

Get all attributes in a group or dataset. Used together with `getatt` to extract all attributes from a group or a dataset, see `ex8.f90`.

Arguments:

```
fid          IN: file identifier
name         IN: name of group or dataset
attnames    OUT: array of attribute name
atttypes    OUT: array of attribute type
attsizes    OUT: array of attribute size
```

6.2 append

```
  SUBROUTINE append(fid, name, array, pardim, ionode, offset)
    IMPLICIT NONE
    INTEGER, INTENT(in) :: fid
    CHARACTER(len=*), INTENT(in) :: name
    <TYPE>, INTENT(in) :: array
    INTEGER, INTENT(in), OPTIONAL :: pardim
    INTEGER, INTENT(in), OPTIONAL :: ionode
    INTEGER, DIMENSION(:), INTENT(in), OPTIONAL :: offset
```

Purpose:

Add array at the end of an extensible dataset. If argument offset is not present, the dataset is extended before it is written to. If argument offset is present, the dataset is not extended, and the vector `offset` specifies the position (relative to the first element) of the write in the non-extensible coordinates. In a **collective** call, only the MPI rank `ionode` will perform the output if `ionode` is specified.

Arguments:

```
fid          IN: file identifier
name         IN: name of group or dataset
array        IN: array of type DP or Z and rank 0, 1, 2 or 3
pardim       IN: dimension which is partitioned
ionode       IN: the node (MPI rank) which does the writing
offset       IN: offset in the dataset (default is at the end)
```

6.3 attach

```
  SUBROUTINE attach(fid, name, attr, val)
    IMPLICIT NONE
    INTEGER, INTENT(in) :: fid
    CHARACTER(len=*), INTENT(in) :: name, attr
    <TYPE> INTENT(in) :: val
```

Purpose:

Attach a scalar attribute to group or dataset.

Arguments:

```
fid          IN: file identifier
name         IN: name of group or dataset
attr         IN: name of attribute
val          IN: attribute value of type I, SP, DP, L, S
```

6.4 closeall

```
SUBROUTINE closeall(ierr)
  IMPLICIT NONE
  INTEGER, intent(OUT) :: ierr
```

Purpose:

Flushes all data to disk, closes all file identifiers, and cleans up memory.

Arguments:

```
ierr        OUT: Error status
```

6.5 closef

```
SUBROUTINE closef(fid)
  IMPLICIT NONE
  INTEGER, INTENT(in) :: fid
```

Purpose:

Close the hdf5 file and release fid for reuse.

Arguments:

```
fid          IN: file identifier
```

6.6 creatd

```
SUBROUTINE creatd(fid, r, d, name, desc, compress, pardim, chunking, iscomplex)
  IMPLICIT NONE
  INTEGER, INTENT(in) :: fid, r, d(:)
  CHARACTER(len=*), INTENT(in):: name
  CHARACTER(len=*), INTENT(in), OPTIONAL :: desc
  LOGICAL, INTENT(in), OPTIONAL :: compress
  LOGICAL, INTENT(in), OPTIONAL :: iscomplex
  INTEGER, INTENT(in), OPTIONAL :: pardim
  INTEGER, INTENT(in),DIMENSION(:), OPTIONAL :: chunking
```

Purpose:

Create a dataset for arrays of rank r and shape d, with UNLIMITED size for the r+1 dimension. d is **unused** when r=0.

Arguments:

```
fid          IN: file identifier
r            IN: number of fixed dimensions, not larger than 3
d            IN: shape of the r dimensions
name         IN: name of group or dataset
desc         IN: description of dataset
compress     IN: compress the array  elements. Default: not compressed
pardim       IN: dimension which is partitioned
attr         IN: name of attribute
chunking     IN: chunking size for fixed dimensions
iscomplex    IN: type of dataset is Z (DP by default)
```

6.7 creatf

```
SUBROUTINE creatf(file, fid, desc, real_prec, mpicomm, mpiposix)
  IMPLICIT NONE
  INCLUDE 'mpif.h'
  CHARACTER(len=*), INTENT(in) :: file
  INTEGER, INTENT(out) :: fid
  CHARACTER(len=*), INTENT(in), OPTIONAL :: desc
  CHARACTER(len=1), INTENT(in), OPTIONAL :: real_prec
  INTEGER, INTENT(in), OPTIONAL :: mpicomm
  LOGICAL, INTENT(in), OPTIONAL :: mpiposix
```

Purpose:

Creates a new HDF5 file and returns a *file identifier*. Notes: mpiposix is not supported in hdf5-1.8.13 and later!

Arguments:

```
file         IN: file name
fid          IN: file identifier
desc         IN: description of file
real_prec    IN: kind of reals: 'd' or 'D' for DP datasets, default is SP.
mpicomm      IN: MPI communicator
mpiposix     IN: use MPI Posix if true. Default is MPI-IO.
```

6.8 creatg

```
SUBROUTINE creatg(fid, name, desc)
  IMPLICIT NONE
  INTEGER, INTENT(in) :: fid
```

```
    CHARACTER(len=*), INTENT(in):: name
    CHARACTER(len=*), INTENT(in), OPTIONAL :: desc
```

Purpose:

Creates a group

Arguments:

```
fid         IN: file identifier
name        IN: name of group or dataset
desc        IN: description of group
```

6.9 extend

```
  SUBROUTINE extend(fid, name, length)
    IMPLICIT NONE
    INTEGER, INTENT(in) :: fid
    CHARACTER(len=*), INTENT(in):: name
    INTEGER, INTENT(in) :: length
```

Purpose:

Extends a dataset

Arguments:

```
fid         IN: file identifier
name        IN: name of group or dataset
length      IN: number of elements to extend group by
```

6.10 getarr

```
  SUBROUTINE getarr(fid, name, array, pardim, ionode)
    IMPLICIT NONE
    INTEGER, INTENT(in) :: fid
    CHARACTER(len=*), INTENT(in) :: name
    <TYPE>, INTENT(out) :: array
    INTEGER, INTENT(in), OPTIONAL :: pardim
    INTEGER, INTENT(in), OPTIONAL :: ionode
```

Purpose:

Read array from dataset. In a **collective** call, only the MPI rank `ionode` will read the dataset if `ionode` is specified.

Arguments:

```
fid          IN: file identifier
name         IN: name of group or dataset
array       OUT: array of rank=1,..,6 and of types I, SP, DP, C and Z
pardim       IN: dimension which is partitioned
ionode       IN: the node (MPI rank) which does the writing
```

6.11 getarrnd

```
SUBROUTINE getarrnd(fid, name, array, pardim, garea)
  IMPLICIT NONE
  INTEGER, INTENT(in) :: fid
  CHARACTER(len=*), INTENT(in) :: name
  <TYPE>, INTENT(out) :: array
  INTEGER, INTENT(in) :: pardim(:)
  INTEGER, INTENT(in), OPTIONAL :: garea(:)
```

Purpose:

Read from dataset into the local array. This is a collective call from ALL processors in the communicator (with a defined cartesian topology) passed to the file open/creation routine

Arguments:

```
fid          IN: file identifier
name         IN: name of group or dataset
array       OUT: array of rank=1,..,6 and of types I, SP, DP, C and Z
pardim       IN: dimensions which are partitioned. Its size should be the
                 same as the number of dimensions of the toplogy associated
                 with the communicator.
garea        IN: ghost area of local array. For example, for 2d partition
                 a ghost area of 2 elements on each side in each dimension
                 is defined by garea=(/2,2/). Default is no ghost area.
```

6.12 getatt

```
SUBROUTINE getatt(fid, name, attr, val, err)
  IMPLICIT NONE
  INTEGER, INTENT(in) :: fid
  CHARACTER(len=*), INTENT(in) :: name, attr
  <TYPE>, INTENT(out) :: val
  INTEGER, INTENT(out), OPTIONAL :: err
```

Purpose:

Get attribute from group or dataset

Arguments:

```
fid          IN: file identifier
name         IN: name of group or dataset
attr         IN: name of attribute
val         OUT: attribute value of types I, SP, DP, L, S
err         OUT: Attribute not found (-1) or of wrong type (-2)
```

6.13 getdims

```
SUBROUTINE getdims(fid, name, rank, dims)
  IMPLICIT NONE
  INTEGER, INTENT(in) :: fid
  CHARACTER(len=*), INTENT(in) :: name
  INTEGER, INTENT(out) :: rank, dims(:)
```

Purpose:

Get rank and dimensions of dataset (cf. ex14.f90).

Arguments:

`fid`	`IN: file identifier`
`name`	`IN: name of dataset`
`rank`	`IN: rank of dataset`
`dims`	`IN: dimensions of dataset`

6.14 getfile

```
SUBROUTINE getfile(fid, name, path)
  IMPLICIT NONE
  INTEGER, INTENT(in) :: fid
  CHARACTER(len=*), INTENT(in) :: name
  CHARACTER(len=*), INTENT(in), OPTIONAL :: path
```

Purpose:

Get file in dataset name and put it in path or standard ouput(default).

Arguments:

`fid`	`IN: file identifier`
`name`	`IN: name of group or dataset`
`path`	`IN: pathname of output file. Default is standard output.`

6.15 geth5ver

```
SUBROUTINE geth5ver(libver, l)
  IMPLICIT NONE
  CHARACTER(len=*), INTENT(out) :: libver
  INTEGER, INTENT(out) :: l
```

Purpose:

Get HDF5 library version

Arguments:

`libver`	`OUT: HDF5 library version (for example "1.8.7")`
`l`	`OUT: lenght of the string libver`

6.16 getsize

```
SUBROUTINE getsize(fid, name, n)
  IMPLICIT NONE
  INTEGER, INTENT(in) :: fid
  CHARACTER(len=*), INTENT(in) :: name
  INTEGER, INTENT(out) :: n
```

Purpose:

Get the size of last dimension of dataset, mainly used when extending an existing "extendible" dataset (cf. ex2.f90).

Arguments:

fid	IN: file identifier
name	IN: name of group or dataset
n	IN: size of the last dimension of dataset

6.17 isdataset

```
LOGICAL FUNCTION isdataset(fid, name)
  IMPLICIT NONE
  INTEGER, INTENT(in) :: fid
  CHARACTER(len=*), INTENT(in) :: name
```

Purpose:

Is name a dataset?

Arguments:

fid	IN: file identifier
name	IN: path name of item

6.18 isgroup

```
LOGICAL FUNCTION isgroup(fid, name)
  IMPLICIT NONE
  INTEGER, INTENT(in) :: fid
  CHARACTER(len=*), INTENT(in) :: name
```

Purpose:

Is name a group?

Arguments:

fid	IN: file identifier
name	IN: path name of item

6.19 numatts

```
INTEGER FUNCTION numatts(fid, name)
  IMPLICIT NONE
  INTEGER, INTENT(in) :: fid
  CHARACTER(len=*), INTENT(in) :: name
```

Purpose:

Number of attributes in group or dataset.

Arguments:

```
fid          IN: file identifier
name         IN: name of group or dataset
```

6.20 openf

```
SUBROUTINE openf(file, fid, mode, real_prec, mpicomm, mpiposix)
  IMPLICIT NONE
  INCLUDE 'mpif.h'
  CHARACTER(len=*), INTENT(in) :: file
  INTEGER, INTENT(out) :: fid
  CHARACTER(len=1), OPTIONAL, INTENT(in) :: real_prec
  CHARACTER(len=*), OPTIONAL, INTENT(in) :: mode
  INTEGER, INTENT(in), OPTIONAL :: mpicomm
  LOGICAL, INTENT(in), OPTIONAL :: mpiposix
```

Purpose:

Open an existing file with filename file and returns a file identifier. Notes: mpiposix is not supported in hdf5-1.8.13 and later!

Arguments:

```
file          IN: file name
fid          OUT: file identifier
mode          IN: Read only if mode='r' or 'R'. Default is read-write.
real_prec     IN: kind of reals: 'd' or 'D' for DP datasets.
mpicomm       IN: MPI communicator
mpiposix      IN: use MPI Posix if true. Default is MPI-IO.
```

6.21 putarr

```
SUBROUTINE putarr(fid, name, array, desc, compress, pardim, ionode)
  IMPLICIT NONE
  INTEGER, INTENT(in) :: fid
  CHARACTER(len=*), INTENT(in) :: name
  <TYPE>, INTENT(in) :: array
  CHARACTER(len=*), INTENT(in), OPTIONAL :: desc
```

```
    LOGICAL, INTENT(in), OPTIONAL :: compress
    INTEGER, INTENT(in), OPTIONAL :: pardim
    INTEGER, INTENT(in), OPTIONAL :: ionode
```

Purpose:

Write array to a new dataset. In a **collective** call, only the MPI rank `ionode` will perform the output if `ionode` is specified.

Arguments:

```
fid          IN: file identifier
name         IN: name of group or dataset
array        IN: array of rank=1,..,6 and of types I, SP, DP, C and Z
desc         IN: description of dataset
compress     IN: compress the array  elements. Default: not compressed
pardim       IN: dimension which is partitioned
ionode       IN: the node (MPI rank) which does the writing
```

6.22 putarrnd

```
  SUBROUTINE putarrnd(fid, name, array, pardim, garea, desc, compress)
    IMPLICIT NONE
    INTEGER, INTENT(in) :: fid
    CHARACTER(len=*), INTENT(in) :: name
    <TYPE>, INTENT(in) :: array
    INTEGER, INTENT(in), OPTIONAL :: garea(:)
    INTEGER, INTENT(in) :: pardim(:)
    CHARACTER(len=*), INTENT(in), OPTIONAL :: desc
    LOGICAL, INTENT(in), OPTIONAL :: compress
```

Purpose:

Write the local array into dataset. This is a collective call from ALL processors in the communicator (with a defined cartesian topology) passed to the file open/creation routine

Arguments:

```
fid          IN: file identifier
name         IN: name of group or dataset
array        IN: array of rank=1,..,6 and of types I, SP, DP, C and Z
pardim       IN: dimensions which are partitioned. Its size should be the
                 same as the number of dimensions of the toplogy associated
                 with the communicator.
garea        IN: ghost area of local array. For example, for 2d partition
                 a ghost area of 2 elements on each side in each dimension
                 is defined by garea=(/2,2/). Default is no ghost area.
desc         IN: description of dataset
compress     IN: compress the array  elements. Default: not compressed
```

6.23 putfile

```
SUBROUTINE putfile(fid, name, path, desc, compress, ionode)
  IMPLICIT NONE
  INTEGER, INTENT(in) :: fid
  CHARACTER(len=*), INTENT(in) :: name
  CHARACTER(len=*), INTENT(in) :: path
  CHARACTER(len=*), INTENT(in), OPTIONAL :: desc
  LOGICAL, INTENT(in), OPTIONAL :: compress
  INTEGER, INTENT(in), OPTIONAL :: ionode
```

Purpose:

Write the file specified in path to a new dataset. In a **collective** call, only the MPI rank `ionode` will write the dataset if `ionode` is specified.

Arguments:

```
fid          IN: file identifier
name         IN: name of group or dataset
path         IN: pathname of file
desc         IN: description of dataset
compress     IN: compress the dataset. Default: not compressed
ionode       IN: the node (MPI rank) which does the writing
```

6.24 split

```
SUBROUTINE split(fullname, group, name)
  IMPLICIT NONE
  CHARACTER(len=*), INTENT(in) :: fullname
  CHARACTER(len=*), INTENT(out) :: group, name
```

Purpose:

Split a dataset full name into group name and dataset name.

Arguments:

```
fullname     IN: full name of dataset
group       OUT: group name
name        OUT: dataset name
```

7 Reference Manual for HASHTABLE

7.1 htable_init

```
SUBROUTINE htable_init
  TYPE(BUFFER_TYPE), INTENT(INOUT)          :: buf
  INTEGER, INTENT(IN), OPTIONAL             :: buffer_length_in
  INTEGER, INTENT(IN), OPTIONAL             :: ionode_in
```

Purpose:

Initialise a hashtable.

Arguments:

```
buf            INOUT: the hashtable.
buffer_length_in  IN: buffer length/number of timesteps per write.
ionode_in         IN: which node does the communication.
```

7.2 add_record

```
 SUBROUTINE add_record(buf,name,description,value,parallel_comm,mpi_operation)
   TYPE(BUFFER_TYPE), INTENT(INOUT) :: buf
   CHARACTER(len=*), INTENT(IN) :: name
   CHARACTER(len=*), INTENT(IN) :: description
   DOUBLE PRECISION, INTENT(IN) :: value
   INTEGER, OPTIONAL,INTENT(IN) :: parallel_comm
   INTEGER, OPTIONAL,INTENT(IN) :: mpi_operation
```

Purpose:

Add a record to a hashtable.

Arguments:

```
buf            INOUT: the hashtable.
name              IN: name of the dataitem.
description       IN: description of the data.
value             IN: value of the data at the current timestep.
parallel_comm     IN: parallel communicator for collective operation.
mpi_operation     IN: which MPI operation to perform (defaul is none).
```

7.3 htable_endstep

```
  SUBROUTINE htable_endstep(buf)
   TYPE(BUFFER_TYPE), INTENT(INOUT) :: buf
```

Purpose:

Signal that the timestep is complete, writing to HDF5 file if necessary.

Arguments:

```
buf            INOUT: the hashtable.
```

7.4 htable_hdf5_flush

```
SUBROUTINE htable_endstep(buf)
 TYPE(BUFFER_TYPE), INTENT(INOUT) :: buf
```

Purpose:

Flush records to HDF5 file.

Arguments:

```
buf            INOUT: the hashtable.
```

7.5 set_htable_fileid

```
SUBROUTINE set_htable_fileid(buf,fresid_in,groupname_in)
  TYPE(BUFFER_TYPE), INTENT(INOUT) :: buf
  INTEGER, INTENT(IN) :: fresid_in
  CHARACTER(len=*), INTENT(in), OPTIONAL :: groupname_in
```

Purpose:

Set HDF5 file parameters.

Arguments:

```
buf            INOUT: the hashtable.
fresid_in         IN: the HDF5 file identifier
groupname_in      IN: the groupname for the 0D data: default is '/data/var0d/'.
```