IOHDF5

Thomas Radke

Date: 2008/05/12 21:03:21

Abstract

Thorn *IOHDF5* provides an I/O method to output variables in HDF5 file format. It also implements checkpointing/recovery functionality using HDF5.

1 Purpose

Thorn *IOHDF5* uses the standard I/O library HDF5[1] to output any type of CCTK grid variables (grid scalars, grid functions, and grid arrays of arbitrary dimension) in the HDF5 file format.

Output is done by invoking the *IOHDF5* I/O method which thorn *IOHDF5*registers with the flesh's I/O interface at startup.

Data is written into files named `"<varname>.h5"`. Such datafiles can be used for further postprocessing (eg. visualization) or fed back into Cactus via the filereader capabilities of thorn **IOUtil**.

2 *IOHDF5* Parameters

Parameters to control the *IOHDF5* I/O method are:

- `IOHDF5::out_every` (steerable)
  How often to do periodic *IOHDF5* output. If this parameter is set in the parameter file, it will override the setting of the shared `IO::out_every` parameter. The output frequency can also be set for individual variables using the `out_every` option in an option string appended to the `IOHDF5::out_vars` parameter.
- `IOHDF5::out_vars` (steerable)
  The list of variables to output using the *IOHDF5* I/O method. The variables must be given by their fully qualified variable or group name. The special keyword *all* requests *IOHDF5* output for all variables. Multiple names must be separated by whitespaces.
  An option string can be appended in curly braces to a group/variable name. Supported options are `out_every` (to set the output frequency for individual variables) and hyperslab options (see section 4 for details).
- `IOHDF5::out_dir`
  The directory in which to place the *IOHDF5* output files. If the directory doesn't exist at startup it will be created.
  If this parameter is set to an empty string *IOHDF5* output will go to the standard output directory as specified in `IO::out_dir`.

3 Serial versus Parallel Output

According to the ouptput mode parameter settings (`IO::out_mode`, `IO::out_unchunked`, `IO::out_proc_every`) of thorn **IOUtil**, thorn *IOHDF5* will output distributed data either

[1] Hierarchical Data Format version 5, see `http://hdf.ncsa.uiuc.edu/whatishdf5.html` for details

- in serial into a single unchunked file

```
IO::out_mode      = "onefile"
IO::out_unchunked = "yes"
```

- in parallel, that is, into separate files containing chunks of the individual processors' patches of the distributed array

```
IO::out_mode      = "proc|np"
```

The default is to output data in parallel, in order to get maximum I/O performance. If needed, you can recombine the resulting chunked datafiles into a single unchunked file using the recombiner utility program. See section 9 for information how to build the recombiner program.

4 Output of Hyperslab Data

By default, thorn *IOHDF5* outputs multidimensional Cactus variables with their full contents resulting in maximum data output. This can be changed for individual variables by specifying a hyperslab as a subset of the data within the N-dimensional volume. Such a subset (called a *hyperslab*) is generally defined as an orthogonal region into the multidimensional dataset, with an origin (lower left corner of the hyperslab), direction vectors (defining the number of hyperslab dimensions and spanning the hyperslab within the N-dimensional grid), an extent (the length of the hyperslab in each of its dimensions), and an optional downsampling factor.

Hyperslab parameters can be set for individual variables using an option string appended to the variables' full names in the `IOHDF5::out_vars` parameter.

Here is an example which outputs two 3D grid functions `Grid::r` and `Wavetoy::phi`. While the first is output with their full contents at every 5th iteration (overriding the `IOHDF5::out_every` parameter for this variable), a two-dimensional hyperslab is defined for the second grid function. This hyperslab defines a subvolume to output, starting with a 5 grid points offset into the grid, spanning in the yz-plane, with an extent of 10 and 20 grid points in y- and z-direction respectively. For this hyperslab, only every other grid point will be output.

```
IOHDF5::out_every = 1
IOHDF5::out_vars  = "Grid::x{ out_every = 5 }
                     Wavetoy::phi{ origin     = {4 4 4}
                                   direction  = {0 1 0
                                                 0 0 1}
                                   extent     = {10 20}
                                   downsample = {2 2}   }"
```

The hyperslab parameters which can be set in an option string are:

- `origin[N]`
  This specifies the origin of the hyperslab. It must be given as an array of integer values with N elements. Each value specifies the offset in grid points in this dimension into the N-dimensional volume of the grid variable.
  If the origin for a hyperslab is not given, if will default to 0.

- `direction[N][M]`
  The direction vectors specify both the directions in which the hyperslab should be spanned (each vector defines one direction of the hyperslab) and its dimensionality (= the total number of dimension vectors). The direction vectors must be given as a concatenated array of integer values. The direction vectors must not be a linear combination of each other or null vectors.
  If the direction vectors for a hyperslab are not given, the hyperslab dimensions will default to N, and its directions are parallel to the underlying grid.

- `extent[M]`
  This specifies the extent of the hyperslab in each of its dimensions as a number of grid points. It must be given as an array of integer values with M elements (M being the number of hyperslab dimensions).
  If the extent for a hyperslab is not given, it will default to the grid variable's extent. Note that if the origin is set to a non-zero value, you should also set the hyperslab extent otherwise the default extent would possibly exceed the variable's grid extent.

- `downsample[M]`
  To select only every so many grid points from the hyperslab you can set the downsample option. It must be given as an array of integer values with M elements (M being the number of hyperslab dimensions).
  If the downsample option is not given, it will default to the settings of the general downsampling parameters `IO::out_downsample_[xyz]` as defined by thorn **IOUtil**.

5 *IOHDF5* Output Restrictions

Due to the naming scheme used to build unique names for HDF5 datasets (see 7, the *IOHDF5* I/O method currently has the restriction that it can output a given variable (with a specific timelevel) – or a hyperslab of it – only once per iteration.

As a workaround, you should request output in such a case by using the flesh's I/O API `CCTK_OutputVarAsByMethod()` routine with a different alias name for each output. Note that this will create multiple output files for the same variable then.

6 Checkpointing & Recovery

Thorn *IOHDF5* can also be used for creating HDF5 checkpoint files and recovering from such files later on.

Checkpoint routines are scheduled at several timebins so that you can save the current state of your simulation after the initial data phase, during evolution, or at termination. Checkpointing for thorn *IOHDF5* is enabled by setting the parameter `IOHDF5::checkpoint = "yes"`.

A recovery routine is registered with thorn **IOUtil** in order to restart a new simulation from a given HDF5 checkpoint. The very same recovery mechanism is used to implement a filereader functionality to feed back data into Cactus.

Checkpointing and recovery are controlled by corresponding checkpoint/recovery parameters of thorn **IOUtil** (for a description of these parameters please refer to this thorn's documentation). The parameter `IO::checkpoint_every_walltime_hours` is not (yet) supported.

7 Importing External Data Into Cactus With *IOHDF5*

In order to import external data into Cactus (eg. to initialize some variable) you first need to convert this data into an HDF5 datafile which then can be processed by the registered recovery routine of thorn *IOHDF5*.

The following description explains the HDF5 file layout of an unchunked datafile which thorn *IOHDF5* expects in order to restore Cactus variables from it properly. There is also a well-documented example C program provided (`IOHDF5/doc/CreateIOHDF5datafile.c`) which illustrates how to create a datafile with *IOHDF5* file layout. This working example can be used as a template for building your own data converter program.

1. Actual data is stored as multidimensional datasets in an HDF5 file. There is no nested grouping structure, every dataset is located in the root group.
   A dataset's name must match the following naming pattern which guarantees to generate unique names:

```
"<full variable name> timelevel <timelevel> at iteration <iteration>"
```

IOHDF5's recovery routine parses a dataset's name according to this pattern to determine the Cactus variable to restore, along with its timelevel. The iteration number is just informative and not needed here.

2. The type of your data as well as its dimensions are already inherited by a dataset itself as metainformation. But this is not enough for *IOHDF5* to safely match it against a specific Cactus variable. For that reason, the variable's groupname, its grouptype, and the total number of timelevels must be attached to every dataset as attribute information.

3. Finally, the recovery routine needs to know how the datafile to recover from was created:

   - Does the file contain chunked or unchunked data ?
   - How many processors were used to produce the data ?
   - How many I/O processors were used to write the data ?
   - What Cactus version is this datafile compatible with ?

   Such information is put into as attributes into a group named `"Global Attributes"`. Since we assume unchunked data here the processor information isn't relevant — unchunked data can be fed back into a Cactus simulation running on an arbitrary number of processors.
   The Cactus version ID must be present to indicate that grid variables with multiple timelevels should be recovered following the new timelevel scheme (as introduced in Cactus beta 10).

The example C program goes through all of these steps and creates a datafile `x.h5` in *IOHDF5* file layout which contains a single dataset named `"grid::x timelevel 0 at iteration 0"`, with groupname `"grid::coordinates"`, grouptype `CCTK_GF` (thus identifying the variable as a grid function), and the total number of timelevels set to 1.

The global attributes are set to `"unchunked" = "yes"`, `nprocs = 1`, and `ioproc_every = 1`.

Once you've built and ran the program you can easily verify if it worked properly with

```
h5dump x.h5
```

which lists all objects in the datafile along with their values. It will also dump the contents of the 3D dataset. Since it only contains zeros it would probably not make much sense to feed this datafile into Cactus for initializing your x coordinate grid function :-)

8 Building A Cactus Configuration with *IOHDF5*

The Cactus distribution does not contain the HDF5 header files and library which is used by thorn *IOHDF5*. So you need to configure it as an external software package via:

```
make <configuration>-config HDF5=yes
                            [HDF5_DIR=<path to HDF5 package>]
```

The configuration script will look in some default places for an installed HDF5 package. If nothing is found this way you can explicitly specify it with the `HDF5_DIR` configure variable.

Thorn *IOHDF5* inherits from *IOUtil* and *IOHDF5Util* so you need to include these thorns in your thorn list to build a configuration with *IOHDF5*.

9 Utility Programs provided by *IOHDF5*

Thorn *IOHDF5* provides the following utility programs:

- `hdf5_recombiner`
  Recombines chunked HDF5 datafile(s) into a single unchunked HDF5 datafile. By applying the `-single_precision` command line option, double precision floating-point datasets can be converted into single precision during the recombination.

- `hdf5_convert_from_ieeeio`
  Converts a datafile created by thorn **IOFlexIO** into an HDF5 datafile. Your thornlist must include this thorn in its thornlist in order to build the FlexIO-to-HDF5 utility program.

- `hdf5_convert_from_sdf`
  Converts a datafile created by thorn **CactusIO/IOSDF** or other Cactus-external programs into an HDF5 datafile. Your thornlist must include this thorn in its thornlist in order to build the SDF-to-HDF5 utility program.

All utility programs are located in the `src/util/` subdirectory of thorn *IOHDF5*. To build the utilities just do a

```
make <configuration>-utils
```

in the Cactus toplevel directory. The executables will then be placed in the `exe/<configuration>/` subdirectory.

All utility programs are self-explaining – just call them without arguments to get a short usage info. If any of these utility programs is called without arguments it will print a usage message.