IOUtil

Thomas Radke, Gabrielle Allen

Date: 2008/08/28 21:03:45

1 Introduction

This document details the thorns provided in the standard Cactus distribution for the output of grid variables, and describes how to set parameters for I/O and checkpointing/recovery using these thorns.

Input and output of data (I/O) in Cactus is provided by infrastructure thorns, which interact with the flesh via a fixed interface, which is described in the Users' Guide. The standard release of Cactus contains a number of thorns which provide so-called I/O methods implementing the actual I/O in a variety of data formats and styles. All these provided I/O methods use thorn **IOUtil** which provides general utilities for I/O (such as parsing parameter strings to decide which variables to output), and a general set of parameters which are inherited by the different I/O methods (such as the output directory). Thorn **IOUtil** by itself provides no I/O methods.

More information about I/O and visualisation of Cactus data can be found in the individual I/O thorns, and in the `Visualization-HOWTO` available on the Cactus web pages.

2 I/O Methods in Cactus

Cactus has several I/O methods for the output of grid variables in different data formats and styles. Each I/O method comes with its own parameters by which it can be customised, and all methods are registered with the flesh, satisfying the Cactus API, allowing them to be called directly from application thorns. An I/O method registers itself with the flesh along with it's name, and these registered names are the labels we now use to describe the various methods.

The standard provided Cactus I/O methods are shown in Table 1. As described above, each of these I/O thorns inherit parameters from thorn **IOUtil**, which must be included in your ThornList and activated in your parameter files before any of these I/O methods can be used. **IOUtil** allows you to set the default behaviour for all the I/O methods described above, for example, setting the parameter

I/O method	**Description**	**Providing Thorn**
`Scalar`	output of scalars or grid array reductions in xgraph or gnuplot format	`CactusBase/IOBasic`
`Info`	screen output of scalars or grid array reductions	`CactusBase/IOBasic`
`IOASCII_1D`	1D line output of grid arrays in xgraph or gnuplot format	`CactusBase/IOASCII`
`IOASCII_2D`	2D slice output of grid arrays in gnuplot format	`CactusBase/IOASCII`
`IOASCII_3D`	full output of 3D grid arrays in gnuplot format	`CactusBase/IOASCII`
`IOJpeg`	2D slice output of grid arrays in jpeg image format	`CactusIO/IOJpeg`
`IOHDF5`	full output of arbitrary grid variables in HDF5 format	`CactusPUGHIO/IOHDF5`
`IOFlexIO_2D`	2D slice output of grid arrays in FlexIO format	`CactusPUGHIO/IOFlexIO`
`IOFlexIO`	full output of arbitrary grid variables in FlexIO format	`CactusPUGHIO/IOFlexIO`

Table 1: Standard I/O methods provided with the Cactus distribution

`IO::out_every = 1` will result in any chosen I/O method providing output on each iteration. The default behaviour can be overridden by specific parameters for each method. For example, you may want scalar and 1D output at every iteration but computationally expensive 3D output only every 10th iteration, with these files going into another directory on a scratch partition.

3 Providing Your Own I/O Method

If you as a Cactus developer have your own input/output routines and want to share this functionality with other people you should do this by putting them into an new I/O thorn and register them as an I/O method. A description on how to register a new I/O method with the flesh's I/O subsystem can be found in the *Infrastructure Thorn Writer's Guide* (as part of the *Cactus User's Guide*).

New I/O thorns should always inherit from thorn **IOUtil** in order to reuse as much of the existing I/O infrastructure as possible, and to maintain a uniform interface on how to use the I/O methods.

4 Standard Parameters

Here we describe a few of the standard parameters used by **IOUtil** to control output.

- `IO::out_dir`
  The name of the directory to be used for output. All the I/O methods described here will write by default to this directory (which itself defaults to the current working directory). Individual methods have parameters which can direct their output to a different directory.
- `IO::out_criterion`
  The criterion that decides when to output. The default is to output every so many iterations (see `IO::out_every`).
- `IO::out_every`
  How often, in terms of iterations, each of the Cactus I/O methods will write output. Again, individual methods can set their own parameters to override this. The default is to never write output.
- `IO::out_dt`
  How often, in terms of simulation time, each of the Cactus I/O methods will write output. Again, individual methods can set their own parameters to override this. The default is to never write output.

5 Saving/Generating Parameter Files

Thorn **IOUtil** can save a copy of the parameter file of a run, or can also automatically generate a parameter file from all current parameter settings. This is controlled by the `IO::parfile_write` parameter:

- `IO::parfile_write="copy"`
  This is the default option, and makes an exact replica of the input parameter file in the standard output directory (this is particularly useful when the output directory is going to be archived).
- `IO::parfile_write="generate"`
  Generate a new parameter file from runtime information, containing the Cactus version, the name of the original parameter file, the run time/date, the host to run on, and the number of processors - all on comment lines. Following this the parameter file contains the ActiveThorns list plus a sorted list of all active thorns' parameters which have been set in the original parameter file.
- `IO::parfile_write="no"`
  Switch off writing of a new parameter file.

The name of the new parameter file defaults to the original filename, unless the parameter `IO::parfile_name` is set. Note that an already existing file with the chosen name will be overwritten unless it is identical with the original parameter file, or if Cactus was recovered from a checkpoint (in which case you don't want to overwrite an existing parameter file with your recovery parameter file).

6 I/O Modes

For a run on multiple processors, scalar, 1D, and 2D output will always be written from only processor zero (that is, required data from all other processors will be sent to processor zero, which then outputs all the gathered data). For full-dimensional output of grid arrays this may become a quite expensive operation since output by only a single processor will probably result in an I/O bottleneck and delay further computation. For this reason Cactus offers different I/O modes for such output which can be controlled by the `IO::out_mode` parameter, in combination with `IO::out_unchunked` and `IO::out_proc_every`. These parameters allow I/O to be optimised for your particular machine architecture and needs:

- `IO::out_mode = "onefile"`
  As for the 1D and 2D I/O methods, writing to file is performed only by processor zero. This processor gathers all the output data from the other processors and then writes to a single file. The gathered grid array data from each processor can be either written in chunks (`IO::out_unchunked = "no"`) with each chunk containing the data from a single processor, or collected into a single global array before writing (`IO::out_unchunked = "yes"`). The default is to write the data in chunks. This can be changed by adding an option string to the group/variable name(s) in the `out_vars` parameter with the key `out_unchunked` and an associated string value `"yes|no|true|false"`.
- `IO::out_mode = "np"`
  Output is written in parallel for groups of processors. Each group consists of `IO::out_proc_every` processors which have assigned one I/O processor which gathers data from the group and writes it to file. The chunked output will go into `IO::out_proc_every` files. The default number of processors in a group is eight.
- `IO::out_mode = "proc"`
  This is the default output mode. Every processor writes its own chunk of data into a separate output file.

Probably the single-processor `"proc"` mode is the most efficient output mode on machines with a fast I/O subsystem and many I/O nodes (e.g. a Linux cluster with local disks attached to each node) because it provides the highest parallelity for outputting data. Note that on very large numbers of processors you may have to fall back to `"np"`, doing output by every so many processors, mode if the system limit of maximum open file descriptors is exceeded (this is true for large jobs on a T3E).

While the `"np"` and `"proc"` I/O modes are fast for outputting large amounts of data from all or a group of processors in parallel, they have the disadvantage of writing chunked files. These files then have to be recombined during a postprocessing phase so that the final unchunked data can be visualized by standard tools. For that purpose a recombiner utility program is provided by the thorns offering parallel I/O methods.

7 Output of Hyperslab Data

While some I/O methods (`IOHDF5, IOFlexIO`) can dump the full contents of a multidimensional CCTK variable, others such as `IOASCII_1D` and `IOASCII_2D` will output only a subset of the data (e.g. 1D lines or 2D planes of 3D grid functions). Such a subset (called a *hyperslab*) is generally defined as an orthogonal region into the multidimensional dataset, with a start point and a length in any direction, and an optional downsampling factor.

Thorn **IOUtil** defines a set of hyperslab parameters for all I/O methods which determine the default positions of 1D line or 2D slice output along the axes. I/O thorns can also define their own hyperslab parameters which then will overwrite the defaults provided by **IOUtil**.

- IO::out_[xyz]line_[xyz]
  specifies the slice center for 1D x,y,z-line output by coordinate values of the underlying physical grid
- IO::out_[xyz]line_[xyz]i
  specifies the slice center of 1D x,y,z-line output by index points of the underlying computational grid
- IO::out_[{xy}{xz}{yz}]plane_[xyz]
  specifies the slice center of 2D xy,xz,yz-plane output by coordinate values of the underlying physical grid
- IO::out_[{xy}{xz}{yz}]plane_[xyz]i
  specifies the slice center of 2D xy,xz,yz-plane output by index points of the underlying computational grid
- IO::out_downsample_[xyz]
  specifies the downsampling factor for output in every direction

Setting the index points for the slice centers in a parameter file has precedence over setting their location by coordinate values. If nothing was specified the default values of IO::out_[xyz]line_[xyz] and IO::out_[{xy}{xz}{yz}]plane_[xyz] will be used. These are set to be all zeros which causes the output to go through the coordinate system's origin or the closest grid point to this (see figure 1 for an example).

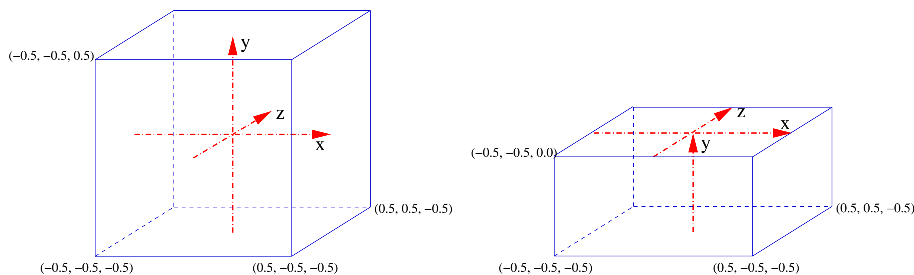

Figure 1: Default 1D *x,y,z*-line output for a 3D grid in box mode (left) and *xy*-bitant mode (right)

If the coordinate values specified in IO::out_[xyz]line_[xyz] or IO::out_[{xy}{xz}{yz}]plane_[xyz] lie outside the physical grid, the slice center simply reverts to the center of the box in that direction. This fallback method method can be changed to using 0-slices instead by setting the corresponding IO::out_[xyz]line_[xyz]i and IO::out_[{xy}{xz}{yz}]plane_[xyz]i parameter(s) to the value -2.

Although they are not hyperslabs by the above definition, output of 1D diagonals for 3D grid arrays is also supported by I/O method IOASCII_1D but has the restriction that the line will always start in the bottom-left corner of the computational grid and steadily rise by one grid point in every direction (see figure 2 for an example).

8 Data Filenames

The standard I/O thorns in Cactus make use of a consistent set of filenames and extensions, which identify the variables and data format used in the file. The filenames are listed in the following table.

9 Checkpointing and Recovery in Cactus

The I/O methods for arbitrary output of CCTK variables also provide functionality for *checkpointing* and *recovery*. A checkpoint is a snapshot of the current state of the simulation (*i.e.* the contents of all the grid variables and the parameter settings) at a chosen timestep. Each checkpoint is saved into a

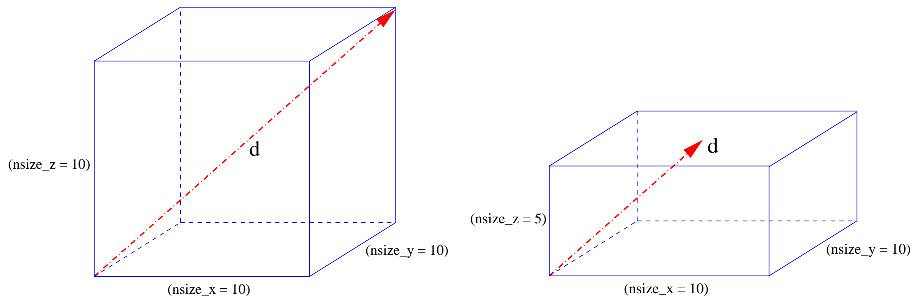

Figure 2: 1D diagonal output for a 3D cubical (left) and non-cubical (right) computational grid

I/O method	**Filename for output of variable** `var`
`Info`	only outputs to screen
`Scalar`	`<var>.{asc|xg}` for scalar variables
	`<var>_<reduction>.{asc|xg}` for reduction values from grid arrays
`IOASCII_1D`	`<var>_<slice>_[<center_i>][center_j>].{asc|xg}`
`IOASCII_2D`	`<var>_<plane>_[<center>].asc`
`IOASCII_3D`	`<var>_3D.asc`
`IOJpeg`	`<var>_<plane>_[<center>].jpeg`
`IOHDF5`	`<var>_3D.h5`
`IOFlexIO_2D`	`<var>_2D.ieee`
`IOFlexIO`	`<var>_3D.ieee`

Table 2: Filenames used by standard I/O thorns

checkpoint file which can be used to restart a new simulation at a later time, recreating the exact state at which it was checkpointed.

Checkpointing is especially useful when running Cactus in batch queue systems where jobs get only limited CPU time. A more advanced use of checkpointing would be to restart your simulation after a crash or a problem had developed, using a different parameter set recovering from the latest stable timestep. Additionally, for performing parameter studies, compute-intensive initial data can be calculated just once and saved in a checkpoint file from which each job can be started.

Again, thorn **IOUtil** provides general checkpoint & recovery parameters. The most important ones are:

- `IO::checkpoint_every` (steerable)
  specifies how often to write a evolution checkpoint in terms of iteration number.
- `IO::checkpoint_every_walltime_hours` (steerable)
  specifies how often to write a evolution checkpoint in terms of wall time. Checkpointing will be triggered if either of these conditions is met.
- `IO::checkpoint_next` (steerable)
  triggers a checkpoint at the end of the current iteration. This flag will be reset afterwards.
- `IO::checkpoint_ID`
  triggers a checkpoint of initial data
- `IO::checkpoint_on_terminate` (steerable)
  triggers a checkpoint at the end of the last iteration of a simulation run
- `IO::checkpoint_file` (steerable)
  holds the basename for evolution checkpoint file(s) to create
  Iteration number and file extension are appended by the individual I/O method used to write the checkpoint.

- `IO::checkpoint_ID_file` (steerable)
  holds the basename for initial data checkpoint file(s) to create
  Iteration number and file extension are appended by the individual I/O method used to write the checkpoint.
- `IO::checkpoint_dir`
  names the directory where checkpoint files are stored
- `IO::checkpoint_keep` (steerable)
  specifies how many evolution checkpoints should be kept
  The default value of 1 means that only the latest evolution checkpoint is kept and older checkpoints are removed in order to save disk space. Setting `IO::checkpoint_keep` to a positive value will keep so many evolution checkpoints around. A value of -1 will keep all (future) checkpoints.
- `IO::recover_and_remove`
  determines whether the checkpoint file that the current simulation has been successfully recovered from, should also be subject of removal, according to the setting of `IO::checkpoint_keep`
- `IO::recover`
  keyword parameter telling if/how to recover.
  Choices are `"no"`, `"manual"`, `"auto"`, and `"autoprobe"`.
- `IO::recover_file`
  basename of the recovery file
  Iteration number and file extension are appended by the individual I/O method used to recover from the recovery file.
- `IO::recover_dir`
  directory where the recovery file is located
- `IO::truncate_files_after_recovering`
  whether or not to truncate already existing output files after recovering

To checkpoint your simulation, you need to enable checkpointing by setting the boolean parameter `checkpoint`, for one of the appropriate I/O methods to `yes`. Checkpoint filenames consist of a basename (as specified in `IO::checkpoint_file`) followed by `".chkpt.it_<iteration_number>"` plus the file extension indicating the file format (`"*.ieee"` for IEEEIO data from `CactusPUGHIO/IOFlexIO`, or `"*.h5"` for HDF5 data from `CactusPUGHIO/IOHDF5`).

Use the `"manual"` mode to recover from a specific checkpoint file by adding the iteration number to the basename parameter.

The `"auto"` recovery mode will automatically recover from the latest checkpoint file found in the recovery directory. In this case `IO::recover_file` should contain the basename only (without any iteration number).

The `"autoprobe"` recovery mode is similar to the `"auto"` mode except that it would not stop the code if no checkpoint file was found but only print a warning message and then continue with the simulation. This mode allows you to enable checkpointing and recovery in the same parameter file and use that without any changes to restart your simulation. On the other hand, you are responsible now for making the checkpoint/recovery directory/file parameters match — a mismatch will not be detected by Cactus in order to terminate it. Instead the simulation would always start from initial data without any recovery.

Because the same I/O methods implement both output of arbitrary data and checkpoint files, the same I/O modes are used (see Section 6). Note that the recovery routines in Cactus can process both chunked and unchunked checkpoint files if you restart on the same number of processors — no recombination is needed here. That's why you should always use one of the parallel I/O modes for checkpointing. If you want to restart on a different number of processors, you first need to recombine the data in the checkpoint file(s) to create a single file with unchunked data. Note that Cactus checkpoint files are platform independent so you can restart from your checkpoint file on a different machine/architecture.

By default, existing output files will be appended to rather than truncated after successful recovery. If you don't want this, you can force I/O methods to always truncate existing output files. Thorn **IOUtil** provides an aliased function for other I/O thorns to call:

```
CCTK_INT FUNCTION IO_TruncateOutputFiles (CCTK_POINTER_TO_CONST IN cctkGH)
```

This function simply returns 1 or 0 if output files should or should not be truncated.

WARNING:
Checkpointing and recovery should **always** be tested for a new thorn set. This is because only Cactus grid variables and parameters are saved in a checkpoint file. If a thorn has made use of saved local variables, the state of a recovered simulation may differ from the original run. To test checkpointing and recovery, simply perform one run of say 10 timesteps, and compare output data with a checkpointed and recovered run at say the 5th timestep. The output data should match exactly if recovery was successful.

10 Reading Data from Files into Cactus

The very same routines which implement checkpointing/recovery functionality in the `IOHDF5` and `IOFlexIO` thorns are also used to provide file reader capabilities within Cactus. They enable users to read variables, whose contents were written to files in HDF5 or IEEEIO data format, back into Cactus at a later time. This is especially useful if compute-intensive initial data is calculated only once and stored in a file. Such data can then be read back in at startup and immediately used by following evolution runs.

The following **IOUtil** parameters exist to specify what variables should be read from file(s) as initial data:

- `IO::filereader_ID_dir`
  root directory for files to be read
- `IO::filereader_ID_files`
  list of files to read in as initial data (multiple filenames must be separated by spaces)
  The same file naming conventions (what I/O mode used, which iteration number) apply as for checkpoint files.
- `IO::filereader_ID_vars`
  list of CCTK variables to read in from the given initial data files (variables are identified by their full name, multiple variable names must be separated by spaces)
  This is useful if a datafile contains multiple variables but only some of them should be read. Thus it is possible to recover distinguished variables from a full checkpoint file.
  Note that if the file contains several timesteps of the same variable only the last one is taken by default. This can be changed by adding an option string with the key `cctk_iteration` and an associated integer scalar value to the variable name denoting the iteration number to choose, like in `IO::filereader_ID_vars = "wavetoy::phi{ cctk_iteration = 10 }"`.

Thorn **IOUtil** also provides a filereader API which can be called by any application thorn at any time. It gets passed the equivalent information to the filereader parameters, plus a pointer to the underlying CCTK grid hierarchy. The return code denotes the total number of variables recovered by the filereader.

C API:

```
#include "CactusBase/IOUtil/src/ioutil_CheckpointRecovery.h"


int IOUtil_RecoverVarsFromDatafiles (cGH *GH,
                                     const char *in_files,
                                     const char *in_vars);
```

Fortran API:

```
call IOUtil_RecoverVarsFromDatafiles (result, GH, in_files, in_vars)

  integer       result
  CCTK_POINTER  GH
```

```
character*(*) in_files
character*(*) in_vars
```

If data is to be imported from files which were not created by `IOHDF5` or `IOFlexIO` it needs to be converted first into the appropriate HDF5 or IEEEIO file format and the file layout which either one of these thorns uses. This is described in detail in the thorns' documentation, along with a simple C source file which can be used as a template to build your own data converter program.

11 Example Parameter Files

Here we give examples of the parameters for the different I/O methods.

- **Output information to screen using** `IOBasic's "Info"` I/O method

```
ActiveThorns = "IOBasic IOUtil PUGHReduce ..."

# Output using all methods on iteration 0, 10, 20, ...
IO::out_every = 10

# Group of variables to output to screen
IOBasic::outInfo_vars = "evolve::vars"
```

- **Scalar Output from** `IOBasic's "Scalar"` **I/O method**

```
ActiveThorns = "IOBasic IOUtil PUGHReduce ..."

# Output vars using scalar method on iteration 0, 10, 20, ...
IOBasic::outScalar_every = 10

# Group of variables to output to file
IOBasic::outScalar_vars = "evolve::vars"
```

- **ASCII 1D and 2D Output with** `IOASCII's "IOASCII_1D"` **and** `"IOASCII_2D"` **I/O methods**

```
ActiveThorns = "IOASCII IOUtil PUGHSlab ..."

# Output vars in 1D on iteration 0, 10, 20, ...
IOASCII::out1D_every = 10

# Output vars in 2D on iteration 0, 50, 100, ...
IOASCII::out2D_every = 50
```

- **HDF5 Output with** `IOHDF5's "IOHDF5"` **I/O method**

```
ActiveThorns = "IOHDF5 IOUtil PUGHSlab ..."

# Output vars in HDF5 format on iteration 0, 5, 10, ...
IOHDF5::out_every = 5

# Group of variables to output
IOHDF5::out_vars = "evolve::vars"

# Special I/O directory for HDF5 output
IOHDF5::out_dir = "/scratch/tmp"

# Full output unchunked to one file
```

```
# (Only using a small number of processors)
IO::out_mode      = "onefile"
IO::out_unchunked = "yes"

# Downsample full data by a factor of 3 in each direction
IO::out_downsample_x = 3
IO::out_downsample_y = 3
IO::out_downsample_z = 3
```

- **IEEEIO 2D hyperslab and full output using** `IOFlexIO`**'s** `"IOFlexIO_2D"` **and** `"IOFlexIO"` **I/O methods**

```
ActiveThorns = "IOFlexIO FlexIO IOUtil PUGHSlab ..."

# Output vars in 2D IEEEIO format on iteration 0, 100, 200, ...
IOFlexIO::out2D_every = 100

# Output vars in IEEEIO format on iteration 0, 5, 10, ...
IOFlexIO::out_every = 5

# Group of variables to output to file for each method
IOFlexIO::out2D_vars = "evolve::vars"
IOFlexIO::out_vars   = "evolve::vars"

# 2D output goes into standard I/O directory
IO::out_dir = "test"

# Special I/O directory for full output
IOFlexIO::out_dir = "/scratch/tmp"

# Full output chunked to one file for every eight processors
# (Run on large number of processors)
IO::out_mode       = "np"
IO::out_proc_every = 8
IO::out_unchunked  = "no"

# Downsample full data by a factor of 3 in each direction
IO::out_downsample_x = 3
IO::out_downsample_y = 3
IO::out_downsample_z = 3
```

- **Checkpointing using thorn** `IOFlexIO`

```
ActiveThorns = "IOFlexIO FlexIO IOUtil PUGHSlab ..."

# Use IEEEIO data format for checkpoint files
IOFlexIO::checkpoint = "yes"

# Make a new checkpoint file every 300 iterations
IO::checkpoint_every = 300

# Name and directory of checkpoint file
IO::checkpoint_file = "run5"
IO::checkpoint_dir  = "/scratch/tmp"
```

- **Recovering from a checkpoint file**

```
ActiveThorns = "IOFlexIO FlexIO IOUtil PUGHSlab ..."

# automatically choose the latest checkpoint file
IO::recover = "auto"

# Name and directory of checkpoint file to recover from
IO::recover_file = "run5"
IO::recover_dir  = "/scratch/tmp"
```

12 Providing Your own Checkpointing/Recovery Method

This section is for Cactus developers to describe how they can add a new method for checkpointing/recovery using the existing I/O parameters and the function API of thorn **IOUtil**. Inheriting this functionality from thorn **IOUtil** helps you to reuse existing code and maintain a uniform user interface on how to invoke different checkpointing/recovery methods.

12.1 Adding a Checkpointing Method

Checkpointing is similar to performing full output of an individual grid variable, except that

1. the output is done for all grid variables existing in a grid hierarchy
2. in addition to the contents of all variables, also the current setting of all parameters is saved as well as some other information necessary for recovering from the checkpoint at a later time

A thorn routine providing this checkpointing capability should register itself with the flesh's scheduler at the `CPINITIAL` (for initial data checkpoints), `CHECKPOINT` (for periodic checkpoints of evolution data), and `TERMINATE` time bins (for checkpointing the last timestep of a simulation).

It should also decide whether checkpointing is needed by evaluating the corresponding checkpoint parameters of **IOUtil** (see section 9).

Before dumping the contents of a distributed grid array into a checkpoint file the variable should be synchronized in case synchronization was not done before implicitly by the scheduler.

To gather the current parameter values you can use the C routine

```
char *IOUtil_GetAllParameters (const cGH *GH, int all);
```

from thorn **IOUtil**. This routine returns the parameter settings in an allocated single large string. Its second argument `all` flags whether all parameter settings should be gathered ($! = 0$) or just the ones which have been set before ($== 0$). Note that you should always save all parameters in a checkpoint in order to reproduce the same state after recovery.

As additional data necessary for proper recovery, the following information must be saved in a checkpoint file:

- the current main loop index (used by the driver as the main evolution loop index)
- the current CCTK iteration number (`GH->cctk_iteration`)
- the physical simulation time (`GH->cctk_time`)

Moreover, information about the I/O mode used to create the checkpoint (chunked/unchunked, serial versus parallel I/O), the active thorns list, or this run's Cactus version ID (for compatibility checking at recovery time) could be relevant to save in a checkpoint.

12.2 Adding a Recovery Method

Recovering from a checkpoint is a two step operation:

1. Right after reading the parameter file and finding out if recovery was requested, all the parameters are restored from the checkpoint file (overwriting any previous settings for non-steerable parameters).
2. After the flesh has created the grid hierarchy with all containing grid variables and the driver has set up storage for these, their contents is restored from the checkpoint (overwriting any previously initialized contents).

The flesh provides the special time bins `RECOVER_PARAMETERS` and `RECOVER_VARIABLES` for these two steps (see also the chapter on *Adding a Checkpointing/Recovery Method* in the *Infrastructure Thorn Writer's Guide* as part of the *Cactus User's Guide*).

Thorn **IOUtil** evaluates the recovery parameters (determines the recovery mode to use, construct the name(s) of the recovery file(s) etc.). It also controls the recovery process by invoking the recovery methods of other I/O thorns, one after another until one method succeeded.

A recovery method must provide a routine with the following prototype:

```
int Recover (cGH *GH, const char *basefilename, int called_from);
```

This routine will be invoked by **IOUtil** with the following arguments:

- a GH pointer refer to the grid hierarchy and its grid variables
  Note that this can also be a `NULL` pointer if the routine was called at `CP_RECOVER_PARAMETERS` when no grid hierarchy exists yet.
- the basename of the checkpoint file to recover from
  This name is constructed by **IOUtil** from the settings of `IO::recovery_dir` and `IO::recover_file`. It will also include an iteration number if one of the `auto` recovery modes is used. The filename extension by which checkpoint files of different recovery methods are distinguished must be appended explicitly.
- a flag identifying when the routine was called
  This can be one of the keywords `CP_INITIAL_DATA, CP_EVOLUTION_DATA, CP_RECOVER_PARAMETERS, CP_RECOVER_DATA`, or `FILEREADER_DATA`). This flag tells the routine what it should do when it is being called by **IOUtil**. Note that **IOUtil** assumes that the recovery method can also be used as a filereader routine here which is essentially the same as recovering (individual) grid variables from (data) files.

To perform the first step of recovery process, a recovery method must register a routine with the flesh's scheduler at the `RECOVER_PARAMETERS` time bin. This routine itself should call the **IOUtil** API

```
int IOUtil_RecoverParameters (int (*recover_fn) (cGH *GH,
                                                 const char *basefilename,
                                                 int called_from),
                              const char *file_extension,
                              const char *file_type)
```

which will determine the recovery filename(s) and in turn invoke the actual recovery method's routine as a callback function. The arguments to pass to this routine are

- a function pointer `recover_fn` as a reference to the recovery method's actual recovery routine (as described above)
- `file_extension` – the filename extension for recovery files which are accepted by this recovery method
  When **IOUtil** constructs the recovery filename and searches for potential recovery files (in the `auto` recovery modes) it will only match filenames with the basename as given in the `IO::recovery_file` parameter, appended by `file_extension`.

- `file_type` – the type of checkpoint files which are accepted by this recovery method
  This is just a descriptive string to print some info output during recovery.

The routine registered at `RECOVER_PARAMETERS` should return to the scheduler a negative value if parameter recovery failed for this recovery method for some reason (e.g. if no appropriate recovery file was found). The scheduler will then continue with the next recovery method until one finally succeeds (a positive value is returned). If none of the available recovery methods were successful the flesh would stop the code.

A value of zero should be returned to the scheduler to indicate that no recovery was requested.

The second step during recovery — restoring the contents of grid variables from the recovery file — is invoked by thorn **IOUtil** which registers a routine at `RECOVER_VARIABLES`. This routine calls all recovery methods which were registered before with **IOUtil** via the API

```
int IOUtil_RegisterRecover (const char *name,
                            int (*recover_fn) (cGH *GH,
                                               const char *basefilename,
                                               int called_from));
```

With this registration, all recovery method's actual recovery routines are made known to **IOUtil**, along with a descriptive name under which they are registered.

At `RECOVER_VARIABLES` thorn **IOUtil** will loop over all available recovery routines (passing the same arguments as for parameter recovery) until one succeeds (returns a positive value).