Carpet Scheduling

Thomas Radke and Jonathan Thornburg

2 November 2006

Abstract

This document describes how scheduling works in (**PUGH** and) **Carpet**. This information is particularly useful if you're using **Carpet** and you need to do a mixture of local computations at each grid point (e.g. computing some grid function), and global operations which need to operate across multiple grids (e.g. computing norms of this grid function).

1 Introduction

Cactus offers a predefined set of about 20 schedule bins corresponding to major phases of a simulation, like `BASEGRID`, `INITIAL`, `POSTINITIAL`, `EVOL`, etc. See the Cactus Users' Guide for an up-to-date list of all the schedule bins (currently these are listed in section E3).[1]

There are two major pieces of software involved in scheduling:

- The Cactus flesh.[2] The flesh knows about everything in `schedule.ccl` files, and handles sorting scheduled routines into an order which is consistent with the `BEFORE` and `AFTER` clauses in all the schedule groups. The flesh also handles repeatedly calling scheduled routines which are scheduled with a `WHILE` clause. In addition, the flesh determines when storage is turned on/off for grid scalars, functions, and arrays and when grid arrays and functions are synchronised, based on the `STORAGE:` and `SYNC:` statements in schedule blocks.

  The flesh does *not* know anything about the spatial grid or grids, or how grid functions are stored in memory or distributed across processors.

- The driver, typically **PUGH** or **Carpet**. The driver defines the set of allowed schedule bins. The driver knows about the grid or grids, and how grid functions are stored in memory, distributed across processors, and synchronised.

  The driver does *not* know what's in `schedule.ccl` files.

The basic flow of control in Cactus is that the flesh starts up, does some initialization, then calls the driver. The driver does some more initialization, then sequences through the schedule bins; for each one it calls back into the flesh to have the flesh call all the scheduled routines in that bin in the correct order.

2 PUGH Scheduling

When using the **PUGH** unigrid driver, the scheduling process is pretty simple, and is shown in figure 1. Nowdays most evolutions use **MoL**; all the **MoL** evolution happens inside the `EVOL` schedule bin.

3 Carpet Scheduling

With **Carpet** the scheduling process is much more complicated, because there are (in general) many different grids, and the various actions on the different grids have to be carefully coordinated to obtain accurate results. Erik Schnetter, Scott Hawley, and Ian Hawke give a nice general description of Carpet in their paper [**?**].

[1] Note that a few schedule bins (notably `STARTUP`, `RECOVER_PARAMETERS`, and `SHUTDOWN`) have special semantics; much of the description here doesn't apply to them. Unless you really know what you're doing, you should probably avoid scheduling routines in these bins; instead use the `WRAGH`, `BASEGRID`, and `TERMINATE` schedule bins, respectively.

[2] Strictly speaking, what we're calling the "flesh" is the combination of what's done by the CST when configuring Cactus, and by the actual Cactus flesh when Cactus runs. But this distinction isn't important here, so for simplicity we just refer to the "flesh".

3.1 The Berger-Oliger Algorithm

The basic idea underlying **Carpet** is *mesh refinement*: use small high-resolution grids in those parts of our problem domain where they're needed, and use lower resolution elsewhere. The goal is to approximate the accuracy of using the finest grid spacing everywhere, while being vastly more efficient. The basic algorithm Carpet uses to do this was first published by Marhsa Berger and Joseph Oliger [**?**, **?**, **?**, **?**, **?**].

The Berger-Oliger algorithm uses locally-uniform grids. When the grid resolution needs to be changed, it's changed by a fixed (integer) refinement factor, typically a factor of 2. Fine grids overlap coarser ones.[3] The key to the Berger-Oliger algorithm is that, when doing the time evolution, in each time step coarse grids are stepped first, and their values are then interpolated (in space and/or time) as necessary to provide boundary conditions for fine grids. This process is illustrated in figure 2.

Notice that after each fine grid has taken enough time steps to be at the same time level as next coarser grid, the fine-grid values at points where there is a coarse-grid point are copied ("injected") back to the coarse grid. This keeps the coarse-grid evolution from gradually drifting away from the (more accurate) fine-grid evolution.

3.2 Grid Attributes

In **Carpet**, a local grid (a "cuboid" that has a uniform spacing in each axis, and lives on a single processor) has a number of attributes:

`mglevel`
: This is an integer specifying a "convergence level" in the sense of "convergence testing". Convergence levels are numbered from 0 (the lowest-resolution simulation) up up to some maximum value (≥ 0) for the highest-resolution simulation. It's important to realise that in this context "resolution" refers to the entire ensemble of grids at at different refinement levels in the Berger-Oliger algorithm. That is, incrementing `mglevel` means coarsening *every* grid in the Berger-Oliger algorithm by (typically) a factor of 2.

[3] One could imagine omitting those coarse grid points where there is also a finer grid, but the cost saving would be small, and the bookkeeping quite messy, so it's not worth doing this. (If you want to try, Carpet does support this; it is only necessary to choose a grid structure that has a "hole" in the coarse grid.)

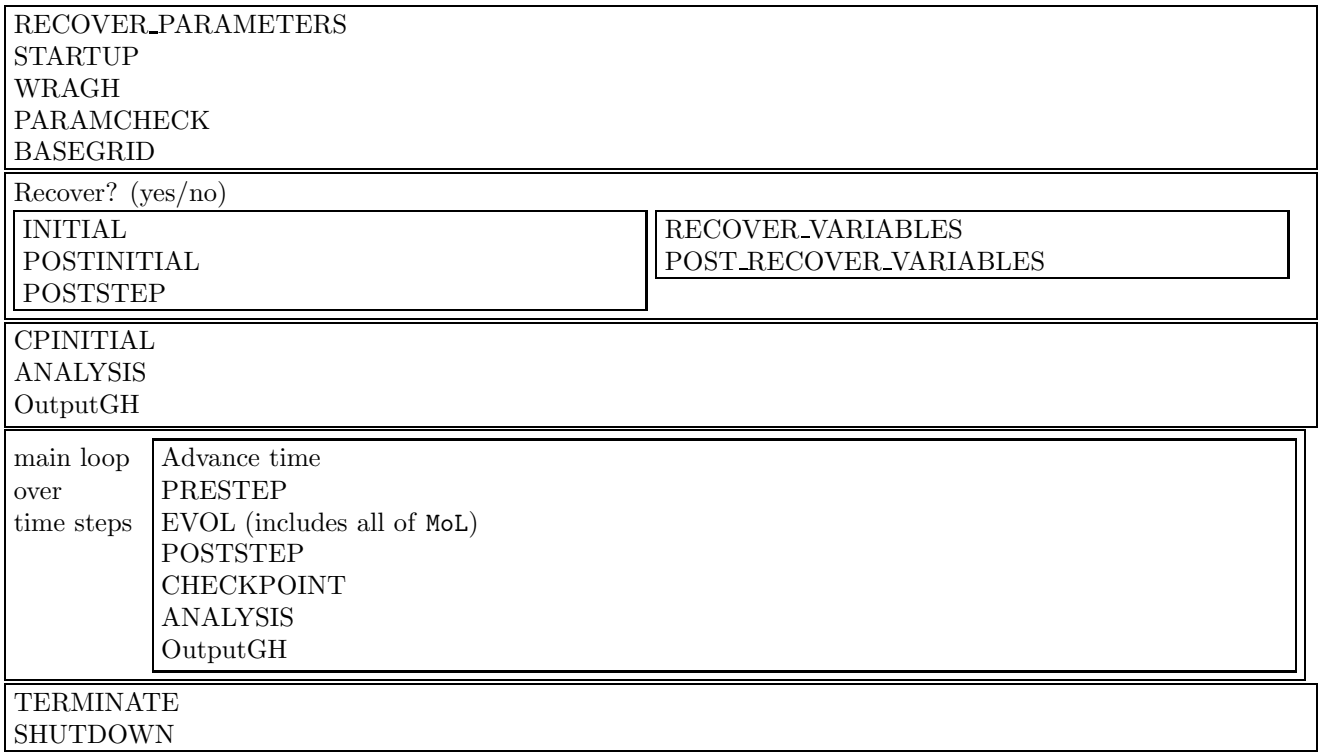

Figure 1: This figure shows the order in which **PUGH** sequences through the shedule bins.

Using convergence levels, you can run several simulations with different resolutions concurrently. This might be useful in a multigrid solver and/or for a "shadow hierarchy" of grids for estimating the local accuracy of an evolution (FIXME: REFERENCE FOR SHADOW HIERARCHY).

At present convergence levels aren't used, so `mglevel` is always set to 0.

`reflevel`
: This is an integer specifing the grid's "refinement level" in the Berger-Oliger algorithm (at this convergence level). Refinement levels are numbered from 0 (the coarsest or "base" grid) up to some maximum value (≥ 0) for the finest grid.

`map`
: This is an integer (≥ 0) specifying the "map" (grid patch) at this convergence level and refinement level.[4] This will always be 0 unless you're doing a multipatch simulation.

`component`
: This is an integer (≥ 0) specifying one of the local grids in this map/patch. By definition, a given local grid lives on a single processor; there may be multiple local grids on a single processor.

3.3 Modes

When looping over grids, Carpet has a standard order to loop over the attributes described in the previous section. This is shown in figure 3. Corresponding to this, Carpet defines a set of "modes", which correspond to the depth in this set of nested loops. In more detail, the modes are as follows:

meta mode
: In meta mode Carpet is logically outside its loops over convergence levels, refinement levels, maps/patches, and components. Therefore, *none* of `mglevel`, `reflevel`, `map`, or `component` are defined.

global mode
: In global mode Carpet is logically inside its loop over convergence levels, but outside its loops over re-

[4] The terms "map" and "patch" mean exactly the same thing. Carpet originally used the term "map", but recently "patch" has become more common. As a rule of thumb, the thorns in the **Carpet** arrangement generally use "map", but other infrastructure thorns (like **MultiPatch** and **GZPatchSystem**) generally use "patch".

This figure doesn't exist yet. :(

Figure 2: This figure shows an example of the sequence of operations for taking a single coarse-grid time step using the Berger-Oliger mesh-refinement algorithm. There are 3 grids, a coarse grid (shown on the left), a medium grid (shown in the middle), and a fine grid (shown on the right).

finement levels, maps/patches, and components. Therefore, `mglevel` is defined, but `reflevel`, `map`, and `component` aren't defined.

level mode
: In level mode Carpet is logically inside its loops over convergence levels and refinement levels, but outside its loops over maps/patches and components. Therefore, `mglevel` and `reflevel` are defined, but `map` and `component` are not defined.

singlemap mode
: In singlemap mode Carpet is logically inside its loops over convergence levels, refinement levels, and maps/patches, but outside its loop over components. Therefore, `mglevel`, `reflevel`, and `map` are defined, but `component` is not defined.

local mode
: In local mode Carpet is logically inside its loops over convergence levels, refinement levels, maps/patches, and components. Therefore, all of `mglevel`, `reflevel`, `map`, and `component` are defined.

```
#
# meta mode
#
begin loop over mglevel (convergence level)
        #
        # global mode
        #
        begin loop over reflevel (refinement level)
                #
                # level mode
                #
                begin loop over map
                        #
                        # singlemap mode
                        #
                        begin loop over component
                                #
                                # local mode
                                #
                        end loop over component
                        #
                        # singlemap mode
                        #
                end loop over map
                #
                # level mode
                #
        end loop over reflevel
        #
        # global mode
        #
end loop over mglevel
#
# meta mode
#
```

Figure 3: This figure shows how Carpet loops over the various attributes of local grids, and how the Carpet modes correspond to depth in this set of nested loops.

When you schedule a routine (by writing a schedule block in a `schedule.ccl` file), you specify what mode it should run in. If you don't specify a mode, the default is local. You can also specify various schedule options; some of these are used in the Carpet scheduling algorithms discussed in section 3.4.

Since convergence levels aren't used at present, for most practical purposes meta mode is identical to global mode. Similarly, unless you're doing multipatch simulations, singlemap mode is identical to level mode.

Table 1 shows which of the grid attributes and predefined Cactus macros and variables are defined in each mode. [Notice that for backwards compatability and simpler program in the most common (single-patch) case, some variables which are logically only defined in singlemap and local modes, are "extended" to also be defined in level mode in single-patch simulations.]

3.3.1 What to Do in Each Mode

As can be seen from table 1, grid functions are defined *only* in local mode. Since most physics code needs to manipulate grid functions, it therefore must run in local mode. (That's why local is the default mode in a `schedule.ccl` schedule block.)

However, in general code scheduled in local mode will run multiple times (because it's nested inside loops over `mglevel`, `reflevel`, `map`, and `component`). Sometimes you don't want this. For example, you may want to open or close an output file, or initialize some global property of your simulation. Meta or global modes (recall that they're identical for most purposes) are good for this kind of thing.

Level mode is in some sense the natural mode for Berger-Oliger mesh refinement. That is, the Berger-Oliger algorithm is naturally written in terms of computations done in level mode. Level mode is convenient for things like convergence tests that depend on the grid spacing.

Singelmap mode is mostly useful (and differs from level mode) only if you're doing multipatch simulations.

Synchronization and turning storage on/off happen in level mode.[5] (At any time only the "current" refinement level has storage turned on.) Boundary conditions must be selected (in the sense of thorn **Boundary**) in level mode. Cactus output must be done in level mode.

Reduction/interpolation of grid arrays and/or grid functions may be done in either level mode (applying only to that refinement level), or in global mode (applying to all refinement levels).

[5] For backwards compatibility, these operations are also allowed (with a warning message printed) in singlemap and local modes if there is only one component per processor.

	meta	global	level	singlemap	local
`mglevel`	×	√	√	√	√
`reflevel`	×	×	√	√	√
`map`	×	×	×	√	√
`component`	×	×	×	×	√
`CCTK_ORIGIN_SPACE`	×	×	√ if single-patch	√	√
`CCTK_DELTA_SPACE`	×	×	√ if single-patch	√	√
`CCTK_DELTA_TIME`	×	×	√	√	√
`cctk_origin_space`	×	√(coarse)	√(coarse) if single-patch	√(coarse)	√(coarse)
`cctk_delta_space`	×	√(coarse)	√(coarse) if single-patch	√(coarse)	√(coarse)
`cctk_delta_time`	×	√(coarse)	√(coarse)	√(coarse)	√(coarse)
grid scalars	×	√	√	√	√
grid arrays	×	√	√	√	√
grid functions	×	×	×	×	√

Table 1: This table shows what grid attributes and Cactus variables are defined in each Carpet mode.
"√ if single-patch" means that the corresponding macros are defined if this is a single-patch simulation, but not if this is a multipatch simulation.
"√(coarse)" means that the corresponding variables are defined, and that they describe the *coarsest* refinement level.

3.3.2 Querying and Changing Modes

Normally, Carpet changes between modes automatically. But for advanced programming, sometimes you need to do this explicitly. Carpet has various functions to query what mode you're in, and functions and macros to change modes. These are all defined in `Carpet/Carpet/src/modes.hh`, and are only usable from C++ code.

To use any of these facilities, put the line "`uses include: carpet.hh`" in your `interface.ccl`, then include `"carpet.hh"` in your C++ source code (this must come *after* including `"cctk.h"`).

To query the current mode, just use any of the Boolean predicates `is_meta_mode()`, `is_global_mode()`, ..., `is_local_mode()`. A common usage of these is in assertions, to verify that the mode is what it should be, e.g.

```
#include <cassert>

#include "cctk.h"
#include "carpet.hh"

void my_function(...)
{
// make sure we're in level mode
assert(Carpet::is_level_mode());
...
}
```

For changing modes, the macros defined in `carpet.hh` provide a higher-level interface, and automagically enforce proper nesting of the modes (in the sense of figure 3). The mode-changing functions provide a lower-level interface, and do not enforce proper nesting, but they can be used with explicit `Carpet::` namespace qualifications (i.e., without having to import the entire `Carpet::` namespace).

The macros set the `cctkGH` entries as appropriate. However, they don't define or undefine grid functions, grid variables, or any `cctk_*` variables; to do this, you need to place a `DECLARE_CCTK_ARGUMENTS` inside the macro loop.

The most commonly-used macros are probably the looping ones, which come in pairs `BEGIN_*_LOOP` and `END_*_LOOP`. These let you move one level deeper in the loop hierarchy. There are also `ENTER_*_MODE` and `LEAVE_*_MODE` macros, which let you escape out of a level (or even levels!) in the loop hierarchy.

Some of these macros also require a `grouptype` argument; this is either `CCTK_GF` or `CCTK_ARRAY` to specify whether you are looping over grid function components or grid array components.

3.4 The Carpet Scheduling Pipeline

It's useful to think of the overall Carpet scheduling process as a pipeline (in the Unix sense) of 3 subprocesses:

1. First, Carpet runs the Berger-Oliger mesh refinement algorithm, as described in section 3.1. Figure 4 gives a summary of how Carpet uses the Berger-Oliger algorithm to step through scheule groups and refinement levels. Figure 5 describes this in (much!) more detail.

   Notice that while some loops in the algorithm traverse the different refinement levels from coarse grids to fine grids as described above, other loops traverse the levels in the opposite direction. This is done to make boundary conditions and symmetries work better.

   Logically, the output of Carpet's Berger-Oliger algorithm is a sequence of tuples:

   (schedule bin, `mglevel`, `reflevel`)

   Since these tuples have definite `mglevel` and `reflevel` values, but no `map` or `component` values, they're semantically in level mode.

2. Next, for each schedule bin output by the previous "pipeline stage", the Cactus flesh sorts all the scheduled routines in that bin into some order which is consistent with the combination of all active thorns' `schedule.ccl` files.[6] This is where `BEFORE`, `AFTER`, and `WHILE` clauses in `schedule.ccl` files are interpreted.

[6] Strictly speaking, "scheduled routine" here also includes entry to and exit from a schedule group, since storage may need to be turned on or off when these events happen.

Logically, the output of this pipeline stage is a sequence of tuples:

(scheduled routine, schedule mode, schedule options, `mglevel`, `reflevel`)

For this "pipeline stage", the schedule mode and options are just uninterpreted tokens to be passed along to the next stage — there's no knowledge (yet) of what they mean.

3. Finally, Carpet applies the schedule options for each scheduled routine, in the manner shown in figure 6. This generates a sequence of calls on scheduled routines. Some calls may be performed in a loop (e.g. for local mode), while others may be skipped (e.g. for global mode).

3.5 Examples, Tips, and Tricks

This section needs more writing. :(

3.5.1 Example 1

Problem

How to perform a global operation (e.g. maximum reduction) once at `CCTK_POSTINITIAL` (call the routine which performs this global operation routine A), before another routine, B, which is scheduled in local mode. As a further constraint, suppose that the scheduling of thorn B cannot be modified.

A solution

Use the Carpet namespace functions to manually change mode, before executing the global computation, and then go back to the original mode. That is, schedule A in local mode `BEFORE` B:

```
schedule A  AT CCTK_PostInitial BEFORE B
{
  LANG: C
} "global operation"
```

where A is the C++ (this is necessary in order to access the carpet namespace variables and functions which are used in the routine) routine described in Fig. 7

3.6 Other Miscellaneous Stuff

Erik says: I am surprised that the `POSTSTEP` loop for the initial data is fine–to–coarse. I think people may apply boundary conditions in this bin... Maybe this loop needs to be reversed?

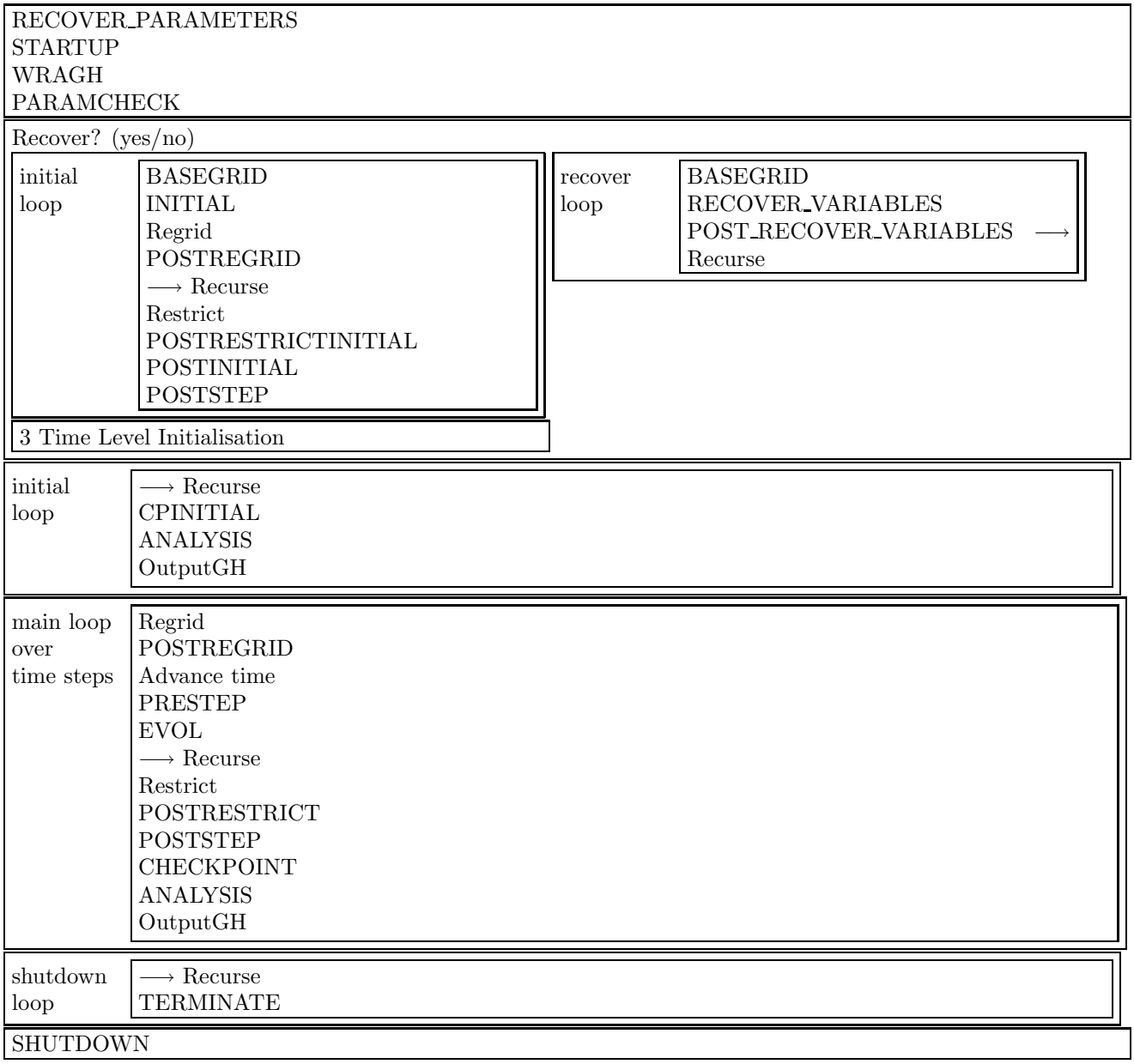

Figure 4: This figure gives an outline of how Carpet uses the Berger-Oliger algorithm to sequence through the schedule bins and grids. See figure 5 for a (much!) more detailed description of the algorithm.
In general, all the loops are traversed in the order from coarse grids to fine grids. FIXME: ARE THERE EXCPETIONS TO THIS RULE (AMONG THE LOOPS SHOWN IN THIS FIGURE)?

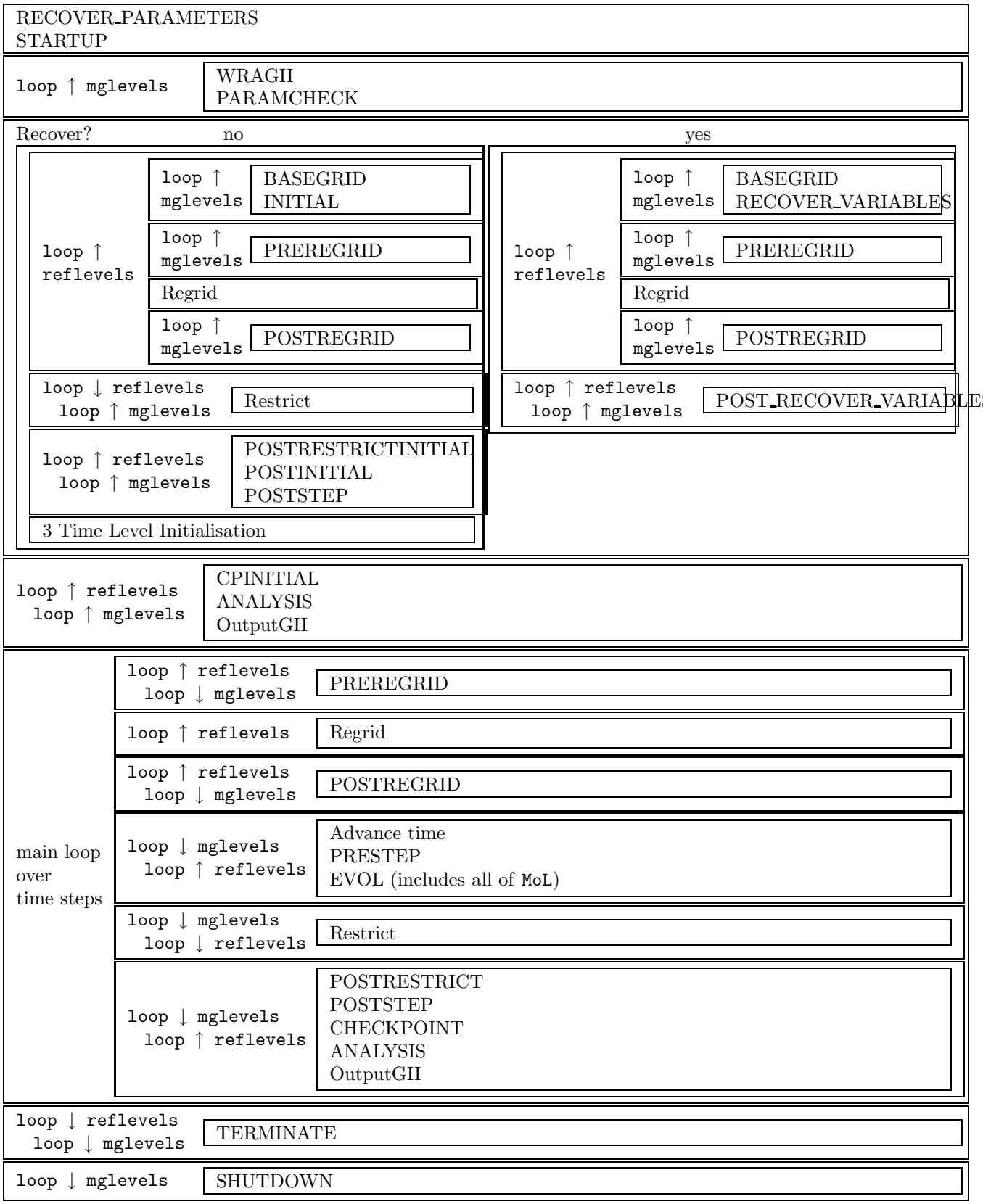

Figure 5: This figure gives a detailed description of how Carpet uses the Berger-Oliger algorithm to sequence through the schedule bins and grids (see figure 4 for a summary of the algorithm).
↑ loops iterate upwards on `mglevel` or `reflevel`, i.e. (for `reflevel`) from coarse grids to fine grids.
↓ loops iterate downwards on `mglevel` or `reflevel`, i.e. (for `reflevel`) from fine grids to coarse grids.
At present `mglevels` aren't used, i.e. the `mglevels` loops have only a single iteration. **Erik: ES: The postregrid bin loops now over all time levels. The initial bin can also do this if init_all_timelevels is selected. Erik: ES: Some of these loops may have change recently. Check.**

procedure `Carpet_use_modes`(**function** `scheduled_routine`,
string `schedule_mode`,
string `schedule_options`,
int `mglevel`, **int** `reflevel`)
select case `schedule_mode`
case META**:**
if (`do_global_mode_routines_now()` **and** this is the coarsest convergence level)
then call `scheduled_routine()` in META mode
case GLOBAL**:**
if (`do_global_mode_routines_now()`)
then call `scheduled_routine(mglevel)` in GLOBAL mode
case LEVEL**:**
call `scheduled_routine(mglevel, reflevel)` in LEVEL mode
case SINGLEMAP**:**
begin loop over all maps `m` in (`mglevel, reflevel`)
call `scheduled_routine(mglevel, reflevel, m)` in SINGLEMAP mode
end loop over map `m`
case LOCAL**:**
begin loop over all maps `m` in (`mglevel, reflevel`)
begin loop over all components `c` in (`mglevel, reflevel, m`)
call `scheduled_routine(mglevel, reflevel, m, c)` in LOCAL mode
end loop over components `c`
end loop over map `m`
end select case
end procedure

Boolean function `do_global_mode_routines_now()`
FIXME: THIS IS COMPLICATED AND VARIES FROM ONE SCHEDULE BIN TO ANOTHER,
BUT IT IS USUALLY A TEST OF THE FORM
`reflevel ==` R
where R is either the coarsest or the finest `reflevel`
end function

In `Carpet_use_modes()`, suppose `schedule_mode` is X. Then if `schedule_options` is "loop-Y", replace

```
call scheduled_routine(X_arguments) in X mode
```

by

```
loop over all ... in ...
        loop over all ... in ...
                call scheduled_routine(X_arguments, ...) in Y mode
        end loop over ...
end loop over ...
```

with the appropriate number and kind of loops to get from X mode to Y mode, and with the corresponding set of extra arguments to the scheduled routine. [If X is LEVEL, SINGLEMAP, or LOCAL, then this gives exactly the same result as just scheduling `scheduled_routine` in Y mode, so the "loop-Y" schedule option is only interesting (gives new semantics) if X is META or GLOBAL.]

Figure 6: This figure shows how Carpet uses modes when calling scheduled routines. The input to this algorithm is the sequence of (scheduled routine, schedule mode, schedule options, `mglevel`, `reflevel`) tuples produced as Carpet steps through the Berger-Oliger algorithm (figures 4 and 5).

```
#include "Carpet/Carpet/src/carpet.hh"

void A(CCTK_ARGUMENTS)
{
 DECLARE_CCTK_ARGUMENTS;
 DECLARE_CCTK_PARAMETERS;
 static int flag = true; // flag used to have this routine effectively executed only once

 // Store the present values of the active refinement level, single map and component.
 // Variables from the Carpet namespace.
 int rl = Carpet::reflevel;
 int singlemap = Carpet::map;
 int comp = Carpet::component;

 if (flag)
    {
    // Go to global mode
    // That is:
    // Leave local mode
    assert(Carpet::is_local_mode());
    Carpet::leave_local_mode (cctkGH);

    // Leave singlemap mode
    assert(Carpet::is_singlemap_mode());
    Carpet::leave_singlemap_mode (cctkGH);

    // Leave level mode
    assert(Carpet::is_level_mode());
    Carpet::leave_level_mode (cctkGH);

    assert(Carpet::is_global_mode());

    // Do the global operation
    ...
    // Go back to local mode

    // Enter level mode
    Carpet::enter_level_mode (cctkGH, rl);
    assert(Carpet::is_level_mode());

    // Enter singlemap mode
    Carpet::enter_singlemap_mode (cctkGH, singlemap);
    assert(Carpet::is_singlemap_mode());

    // Enter local mode
    Carpet::enter_local_mode (cctkGH, comp);
    assert(Carpet::is_local_mode());
  }
// After this has run once, set the flag so that this does not run again
flag = false;
     } // end if (flag)
   else
     {
      return;
     }
}
```

Figure 7: Example1: perform a global operation A, before given local routine B, in the case that the scheduling of thorn B cannot be modified.