CSC 7700: Scientific Computing

Module D: Simulations and Application Frameworks

Lecture 3: Cactus Framework Architecture

Dr Peter Diener

Center for Computation and Technology
Louisiana State University, Baton Rouge, LA

November 22, 2013

Goals

Goals

- Lecture 1 introduced:
  - The concept of a simulation and it's ingredients.
  - Super computers from the application scientist's point of view.
- Lecture 2 introduced:
  - Parallelisation: data structures, load balancing, domain decomposition.
  - Software Engineering: multi-physics simulations, large projects, distributed code development.
- In this lecture we will discuss:
  - The component model as software architecture for real-world simulation codes.
  - The Cactus Software Framework as a specific example.

Summary

Summary

- To go from physics to a simulation, one usually
  1. Finds a mathematical model (e.g. PDEs) expressing the physics.
  2. Discretise the model (e.g. PDEs).
  3. Implement the discretised equations on a supercomputer
- Many simulation codes have a similar structure.
- Many supercomputers have a similar architecture.

Summary

- Parallel algorithm are necessary due to size of the problems (memory) and computational cost (CPU time).
- MPI is the tool of choice. Requires domain decomposition, advanced data structures and load balancing algorithms.
- A component model is necessary to develop complicated multi-physics codes using geographically distributed code developers.
- A *framework* provides the glue between components.
- We introduced the Einstein Toolkit as a real world example.

Cactus Software Framework

Cactus

- Open source HPC software framework developed at LSU/AEI/Caltech/PI.
- First version designed at AEI in 1997 in the field of Numerical Relativity to address simulation code difficulties described earlier.
- Redesigned in 1999 to allow it to be more easily used by other fields of science.
- Highly portable (from notebook to supercomputers).
- Highly parallel (tested on $> 130,000$ cores).

Cactus Design

- The core of the framework (*the flesh*) is lean. It only provides *glue* between components. It performs no "real work".
- The components (*the thorns*) perform all the work, both computational and physics.
- It contains many standard thorns such as providing coordinate systems, standard boundary conditions, MPI drivers, efficient parallel I/O, etc.

Covers

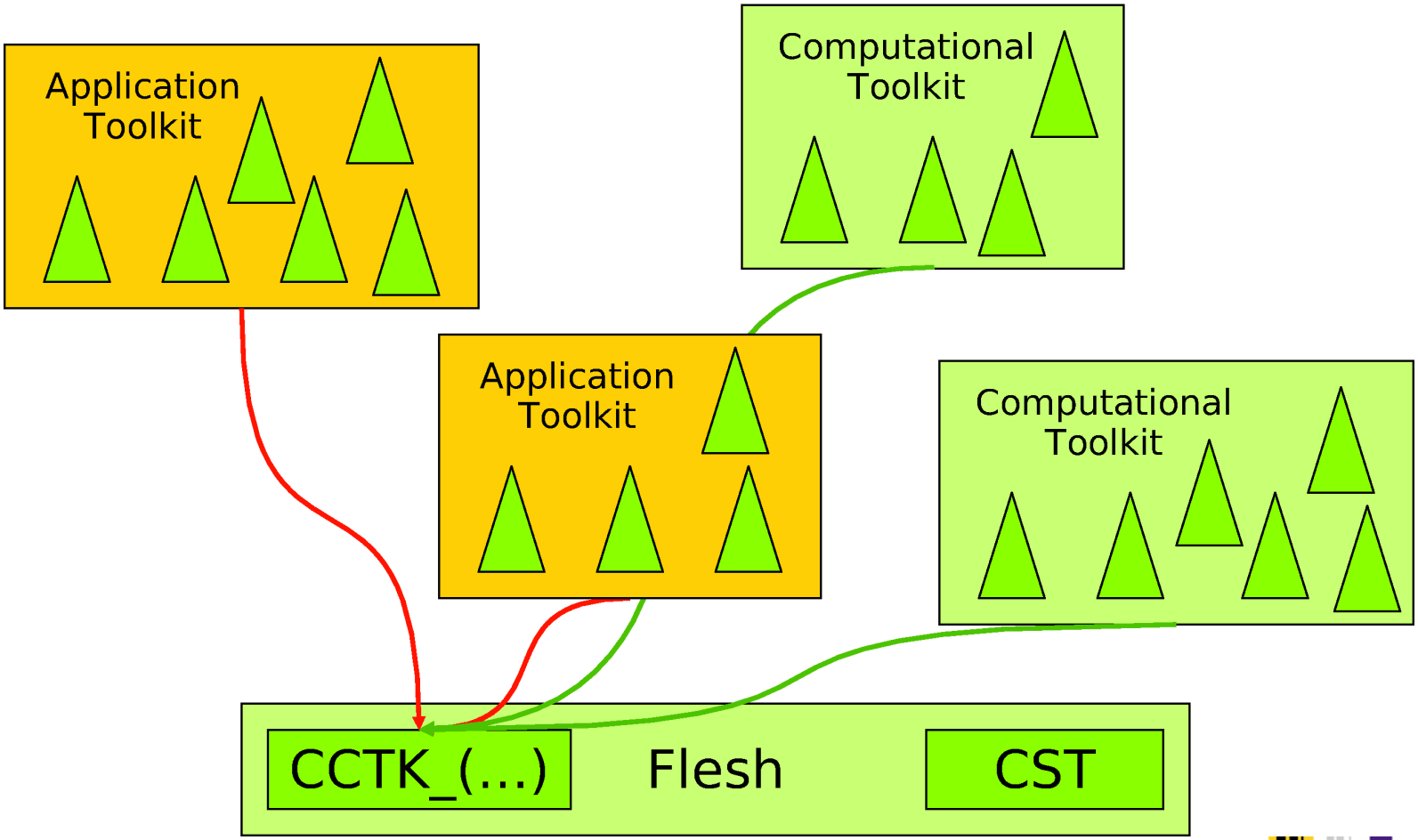

COMMUNICATIONS
HIGH-PERFORMANCE NETWORKING
The Future of Theoretical Physics and Cosmology
MaxPlanckResearch
Sun, Man and Climate
FOCUS
Invisible U
National Science Foundation
Visualization and Processing of Tensor Fields
Chapman & Hall/CRC
Computational Science Series
PETASCALE COMPUTING
ALGORITHMS AND APPLICATIONS
EDITED BY
DAVID A. BADER
Chapman & Hall/CRC
NATIONAL SCIENCE BOARD
SCIENCE AND ENGINEERING INDICATORS 2006
VOLUME 1
Naturwissenschaftliche Rundschau
NR 689

Many arrangements with many modules...

CactusBase	Basic utility and interface thorns
CactusBench	Benchmark utility thorns
CactusConnect	Network utility thorns
CactusElliptic	Elliptic solvers / interface thorns
CactusExamples	Example thorns
CactusExternal	External library interface thorns
CactusIO	General I/O thorns
CactusNumerical	General numerical methods
CactusPUGH	Cactus Unigrid Driver thorn
CactusPUGHIO	I/O thorns specific for PUGH driver
CactusTest	Thorns for Cactus testing
CactusUtils	Misc. utility thorns
CactusWave	Wave example thorns

Application View

The structure of an application that is built upon the Cactus computational framework



Other Frameworks

- Several other HPC simulation frameworks or framework-like architectures exist.
  - However, most are libraries rather than frameworks.
- Frameworks are also common in other software areas.
  - e.g. KDE (linux desktop) and Eclipse (software development environment).

Code assembly

- Different from a "traditional" program. In a framework only the end user controls what components (plugins) are present (active).
- In Cactus, this is handled via *thorn lists* defining the names and download locations of all thorns to be compiled into the executable.
- At run time, thorns can be activated if needed for a given simulation.

Thornlists

- List of thorn names
- Corresponding download methods and locations (optional)
- Supported download methods:
  - CVS / Subversion / Git / Mercurial
  - http / https / ftp
- Example:

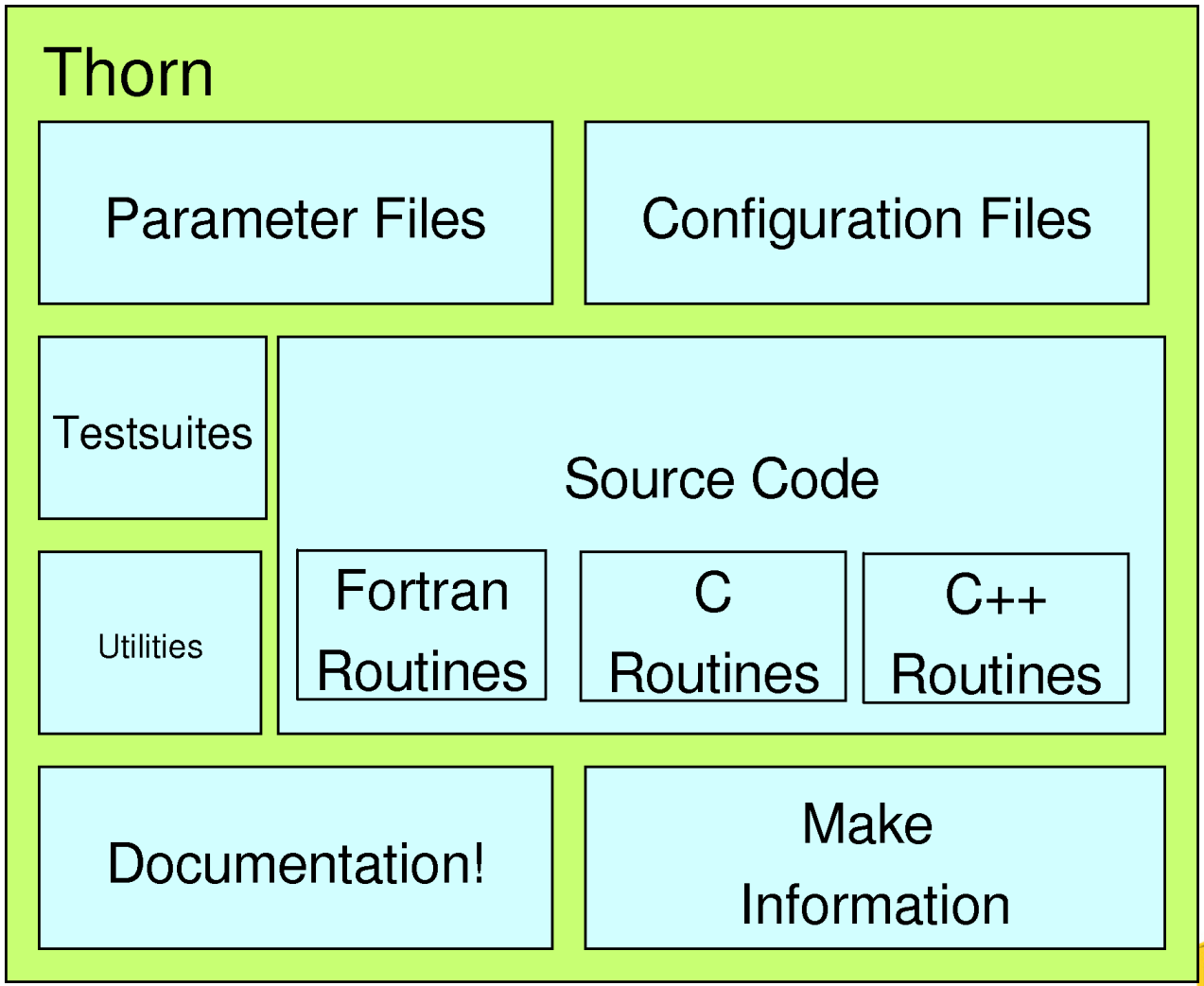
```
!CRL_VERSION = 1.0

# Cactus Flesh
!TARGET   = $ROOT
!TYPE     = svn
!URL      = http://svn.cactuscode.org/flesh/trunk
!CHECKOUT = Cactus
!NAME     = .

# Cactus thorns
!TARGET   = Cactus/arrangements
!TYPE     = svn
!URL      = http://svn.cactuscode.org/arrangements/$1/$2/trunk
!CHECKOUT =
CactusBase/Boundary
CactusBase/CartGrid3D
CactusBase/CoordBase
CactusBase/IOASCII
CactusBase/IOBasic
```

Component structure

- A component (thorn) has:
  - An *interface* that contains its external specification, describing how it looks to the framework and to other thorns.
  - An *implementation* that defines how it actually works, including documentation and test cases.

Thorn Structure

Inside view of a plug-in module, or thorn for Cactus



Thorn Structure

Directory structure:

```
Cactus
`-- arrangements
    `-- Introduction
        `-- HelloWorld
            |-- interface.ccl
            |-- param.ccl
            |-- schedule.ccl
            |-- README
            |-- doc
            |   `-- documentation.tex
            |-- src
            |   |-- HelloWorld.c
            |   `-- make.code.defn
            |-- test
            `-- utils
```

Thorn Specification

Four configuration files per thorn:

- interface.ccl declares:
  - an 'implementation' name
  - inheritance relationships between thorns
  - Thorn variables
  - Global functions, both provided and used

Thorn Specification

Four configuration files per thorn:

- interface.ccl declares:
  - an 'implementation' name
  - inheritance relationships between thorns
  - Thorn variables
  - Global functions, both provided and used
- schedule.ccl declares:
  - When the flesh should schedule which functions
  - When which variables should be allocated/freed
  - Which variables should be syncronized when

Thorn Specification

Four configuration files per thorn:

- interface.ccl declares:
  - an 'implementation' name
  - inheritance relationships between thorns
  - Thorn variables
  - Global functions, both provided and used
- schedule.ccl declares:
  - When the flesh should schedule which functions
  - When which variables should be allocated/freed
  - Which variables should be syncronized when
- param.ccl declares:
  - Runtime parameters for the thorn
  - Use/extension of parameters of other thorns

Thorn Specification

Four configuration files per thorn:

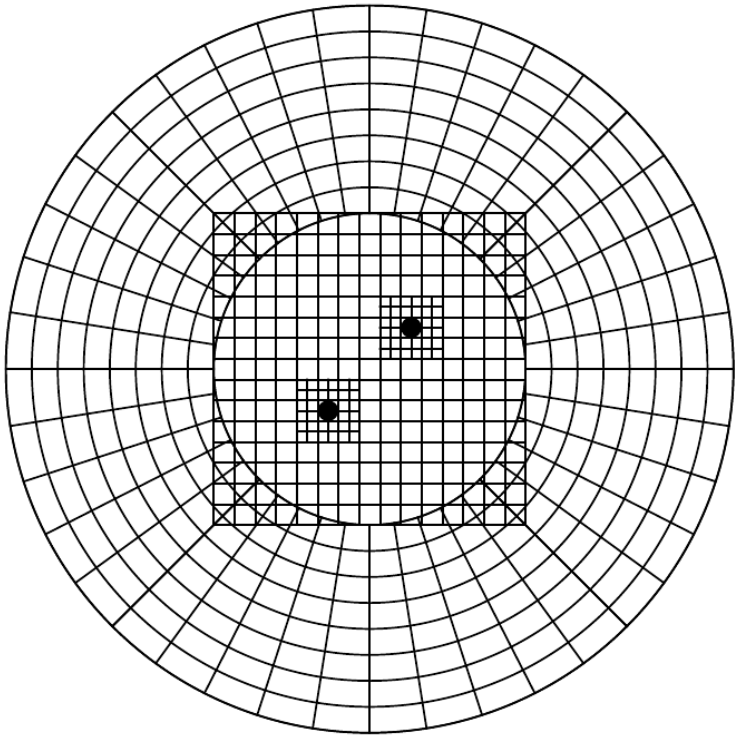
- interface.ccl declares:
  - an 'implementation' name
  - inheritance relationships between thorns
  - Thorn variables
  - Global functions, both provided and used
- schedule.ccl declares:
  - When the flesh should schedule which functions
  - When which variables should be allocated/freed
  - Which variables should be syncronized when
- param.ccl declares:
  - Runtime parameters for the thorn
  - Use/extension of parameters of other thorns
- configuration.ccl (optional) declares:
  - Capabilities provided by or used by the thorn
  - Capabilities are usually related to external libraries

Grid Functions

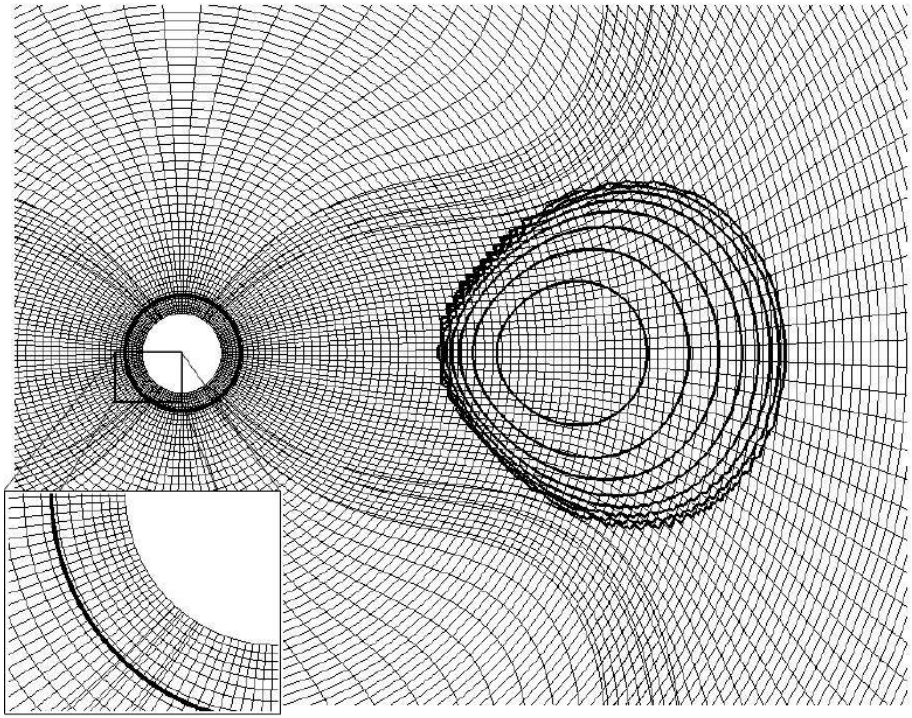
- A discretised physical quantity (e.g. density, pressure, velocity. . .) lives on a grid function.
- If a regular discretisation is used these are essentially distrubuted arrays (hence "grid")
  - Could also be a graph, tree, etc. instead.
- Grid functions are fundamental data types in a simulation.

Example of Cactus Grid Function Shapes

Adaptive Mesh Refinement and Multi-Block



Multi-Block with distorted coordinate systems



interface.ccl

Variables:

- Flesh needs to know about thorn variables for which it has to care about allocation, parallelism, inter-thorn use
- Scopes: public, private
- Many different basic types (double, integer, string, ...
- Different 'group types' (grid functions, grid arrays, scalars, ...)
- Different 'tags' (not to be checkpointed, vector types, ...)

Syntax of interface.ccl

```
IMPLEMENTS: <interface name>
INHERITS: <interface name> ...

[PUBLIC:|PRIVATE:]

[REAL|COMPLEX|INT] <group name> TYPE=[gf|array]
        TIMELEVELS=<number> [DIM=... SIZE=...]
{
  <variable name>
  ...
} <description>
```

Syntax of `interface.ccl` cont.

```
[REAL|COMPLEX|INT|POINTER] FUNCTION <function name> (
        [REAL|COMPLEX|INT|STRING|POINTER] \
                [ARRAY] [IN|OUT] <argument name>,
        ...
        )

[USES|REQUIRES] FUNCTION <function name>

PROVIDES FUNCTION <function name>
         WITH <implementation name> LANG [C|FORTRAN]
```

Example interface.ccl

```
IMPLEMENTS: wavetoy

INHERITS: grid

PUBLIC:

REAL scalarevolve TYPE=gf TIMELEVELS=3
{
  phi
} "The evolved scalar field"
```

Example interface.ccl cont.

```
CCTK_INT FUNCTION Boundary_SelectVarForBC (
        CCTK_POINTER_TO_CONST IN cctkGH,
        CCTK_INT IN faces,
        CCTK_INT IN boundary_width,
        CCTK_INT IN options_handle,
        CCTK_STRING IN var_name,
        CCTK_STRING IN bc_name
        )

REQUIRES FUNCTION Boundary_SelectVarForBC
```

Execution Control Scheduling

- Components are developed independently yet need to execute in a certain order.
  - e.g. calculate pressure first then forces from pressure gradient.
- Don't want end user to have to specify this. This would require "manual assembly" of the components. The end user probably doesn't understand the thorn details.

schedule.ccl

- Flesh contains a flexible rule based scheduler
- Order is prescribed in `schedule.ccl`
- Scheduler also handles when variables are allocated, freed or synchronized between parallel processes
- Functions or groups of functions can be
  - grouped and whole group scheduled
  - scheduled before or after each other
  - scheduled depending on parameters
  - scheduled while some condition is true
- Flesh scheduler sorts all rules and flags an error for inconsistent schedule requests
- Note: The scheduler is in the process of being revamped

Schedule Groups

CCTK_STARTUP For routines, run before the grid hierarchy is set up, for example function registration.

CCTK_PARAMCHECK For routines that check parameter combinations, routines registered here only have access to the grid size and the parameters.

CCTK_BASEGRID Responsible for setting up coordinates, etc.

CCTK_INITIAL For generating initial data.

CCTK_POSTINITIAL Tasks which must be applied after initial data is created.

CCTK_PRESTEP Stuff done before the evolution step.

CCTK_EVOL The evolution step.

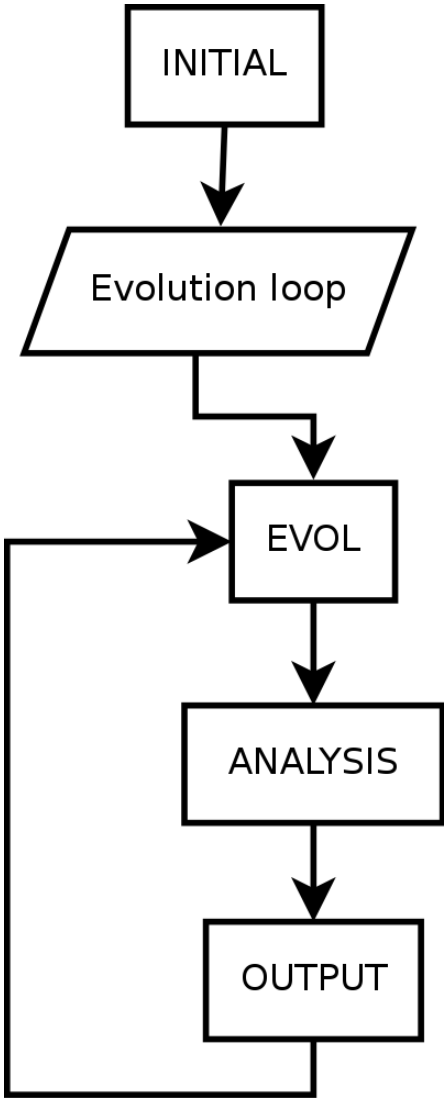
CCTK_POSTSTEP Stuff done after the evolution step.

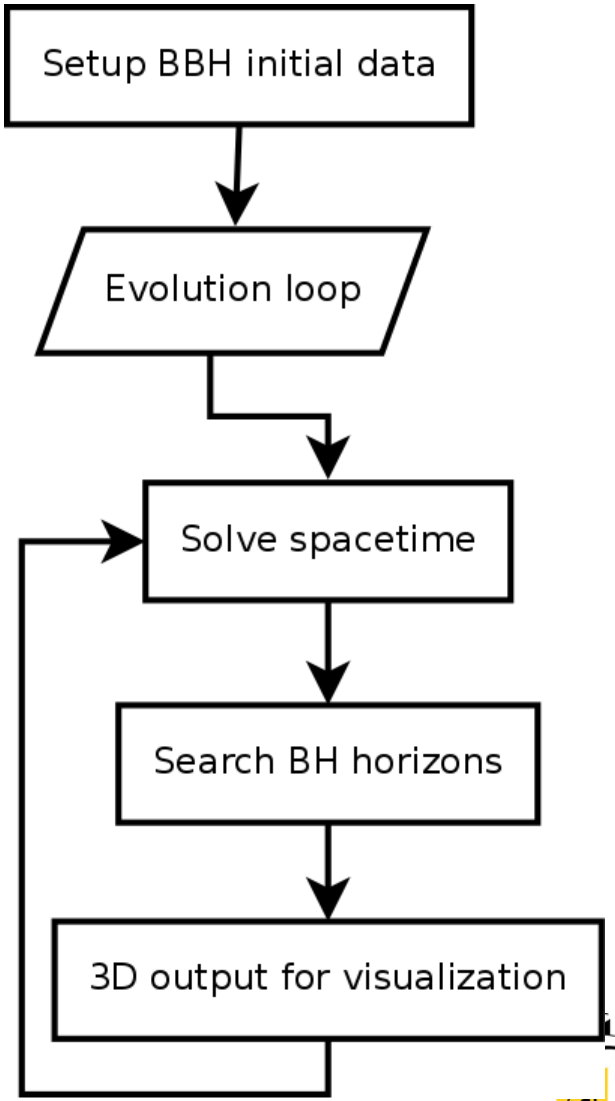
CCTK_ANALYSIS For analysing data.

`schedule.ccl` cont.

- Hierarchy: using schedule groups leads to a *schedule tree*
- Execution order: can schedule `BEFORE` or `AFTER` other items
  e.g.: take time step after calculating RHS
- Loops: can schedule `WHILE` a condition is true
  e.g.: loop while error is too large
- Conditions: can schedule `if` a parameter is set
  e.g.: choose between boundary conditions
- Perform analysis at run time: `TRIGGERS` statements: call routine only if result is needed for I/O

Example scheduling tree





Syntax of schedule.ccl

```
SCHEDULE <function name> [AT <schedule bin>|
                          IN <schedule group>]
{
  LANG: [C|Fortran]
  SYNC: <group name> ...
} <description>

SCHEDULE GROUP <name> [AT <schedule bin>|
                       IN <schedule group>]
{
} <description>}

STORAGE: <group name>[timelevels] ...
```

Example schedule.ccl

```
SCHEDULE WaveToyC_Evolution AT evol
{
  LANG: C
} "Evolution of 3D wave equation"

SCHEDULE GROUP WaveToy_Boundaries AT evol \
                AFTER WaveToyC_Evolution
{
} "Boundaries of 3D wave equation"

STORAGE: scalarevolve[3]
```

Simulation Specification

- Most thorns have some parameters that are only set at run time.
  - e.g. number of grid points, initial data model, boundary conditions, number of time steps, output frequency,...
- End user has to specify these parameters to describe the complete simulation setup when choosing which thorns to activate.

param.ccl

- Definition of parameters
- Scopes: Global, Restricted, Private
- Thorns can use and extend each others parameters
- Different types (double, integer, string, keyword, . . .)
- Range checking and validation
- Steerablility at runtime

Syntax of param.ccl

```
[SHARES: <implementation>]

[PUBLIC:|RESTRICTED:|PRIVATE:]

[BOOLEAN|KEYWORD|INT|REAL|STRING]
    <parameter name> <description> [STEERABLE=...]
{
    <allowed value>                :: <description>
    <lower bound>:<upper bound> :: <description>
    <pattern>                      :: <description>
    ...
} <default value>
```

Example param.ccl

```
SHARES: grid
USES KEYWORD type

PRIVATE:
KEYWORD initial_data "Type of initial data"
{
    "plane"    :: "Plane wave"
    "gaussian" :: "Gaussian wave"
} "gaussian"

REAL radius "The radius of the gaussian wave"
{
    0:* :: "Positive"
} 0.0
```

Hello World Thorn

- Begin with a simple C program.
- Design the thorn CCL files.
- Convert the source code to Cactus source.
- Write a parameter file (it needs to activate the thorn).
- Run it and look at the output.

Hello World, Standalone

Standalone in C:

```
#include <stdio.h>
int main(void)
{
  printf("Hello World!\n");
  return 0;
}
```

Hello World Thorn

- interface.ccl:

```
implements: HelloWorld
```

- schedule.ccl:

```
schedule HelloWorld at CCTK_EVOL
{
  LANG: C
} "Print Hello World message"
```

- param.ccl: empty
- README:

```
Cactus Code Thorn HelloWorld
Author(s)    : Frank Löffler <knarf@cct.lsu.edu>
Maintainer(s): Frank Löffler <knarf@cct.lsu.edu>
Licence      : GPL
--------------------------------------------------------------------------

1. Purpose

Example thorn for tutorial Introduction to Cactus
```

Hello World Thorn cont.

- src/HelloWorld.c:

```
#include "cctk.h"
#include "cctk_Arguments.h"

void HelloWorld(CCTK_ARGUMENTS)
{
  DECLARE_CCTK_ARGUMENTS
  CCTK_INFO("Hello World!");
  return;
}
```

- src/make.code.defn:

```
SRCS = HelloWorld.c
```

Hello World Thorn

- parameter file:

```
ActiveThorns = "HelloWorld"
Cactus::cctk_itlast = 10
```

- run: `[mpirun] <cactus executable> <parameter file>`

Hello World Thorn

- Screen output:

```
       10
  1   0101       ************************
  01  1010 10      The Cactus Code V4.0
 1010 1101 011      www.cactuscode.org
  1001 100101    ************************
    00010101
     100011     (c) Copyright The Authors
      0100      GNU Licensed. No Warranty
      0101

Cactus version:    4.0.b17
Compile date:      Sep 08 2011 (13:15:01)
Run date:          Sep 08 2011 (13:15:54-0500)
[...]

Activating thorn Cactus...Success -> active implementation Cactus
Activation requested for
--->HelloWorld<---
Activating thorn HelloWorld...Success -> active implementation HelloWorld
--------------------------------------------------------------------------------
INFO (HelloWorld): Hello World!
INFO (HelloWorld): Hello World!
INFO (HelloWorld): Hello World!
INFO (HelloWorld): Hello World!
[...] 6x
--------------------------------------------------------------------------------
Done.
```

Cactus Summary

- We introduced the Cactus framework.
- Thorns have implementation (regular code) and interface (ccl) files.
- Introduced ccl file syntax. See users' guide, reference manual and examples for details.
  - `make UsersGuide`
  - `make ReferenceManual`
  - Resulting pdf files will be in `Cactus/doc`
- A parameter file is needed to run a simulation. The parameter file also activates the participating thorns.