CSC 7700: Scientific Computing
Module C: Simulations and Application Frameworks
Lecture 3: Framework Architecture

Dr. Erik Schnetter

Simulation Computer Science

- Computer Science is relevant for simulations in many ways
- Lecture 2 introduced two such ways:
  - Parallelisation: data structures, load balancing, domain decomposition
  - Software Engineering: multi-physics simulations, large projects, distributed code development

Fundamental Parallel Strategies

HPCC

- High Performance Computing Challenge, http://icl.cs.utk.edu/hpcc/
- Define 7 strategies that are commonly used to develop parallel algorithms
- Provide benchmarks with sample implementations that measure these
- (Also collecting a large table of supercomputer performance data)

1. HPL (Linpack): floating point performance (see also Top 500, http://www.top500.org/)
2. DGEMM: floating point performance
3. STREAM: memory bandwidth
4. PTRANS: communication bandwidth
5. RandomAccess: communication latency
6. FFT: floating point performance
7. Communication bandwidth and latency

Today's Goal

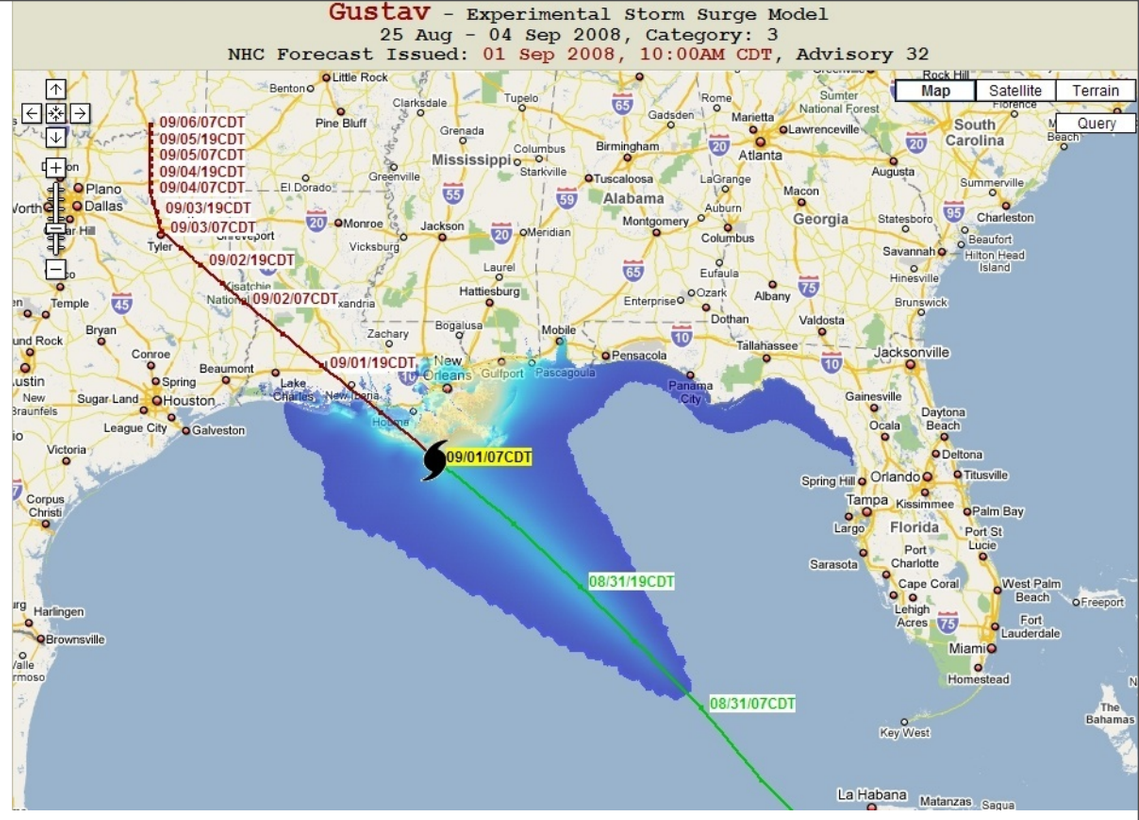
- Discuss the component model as software architecture for real-world simulation codes
- Introduce the Cactus Software Framework and Einstein Toolkit as examples

Part 1: Component Model

Simulation Code Requirements

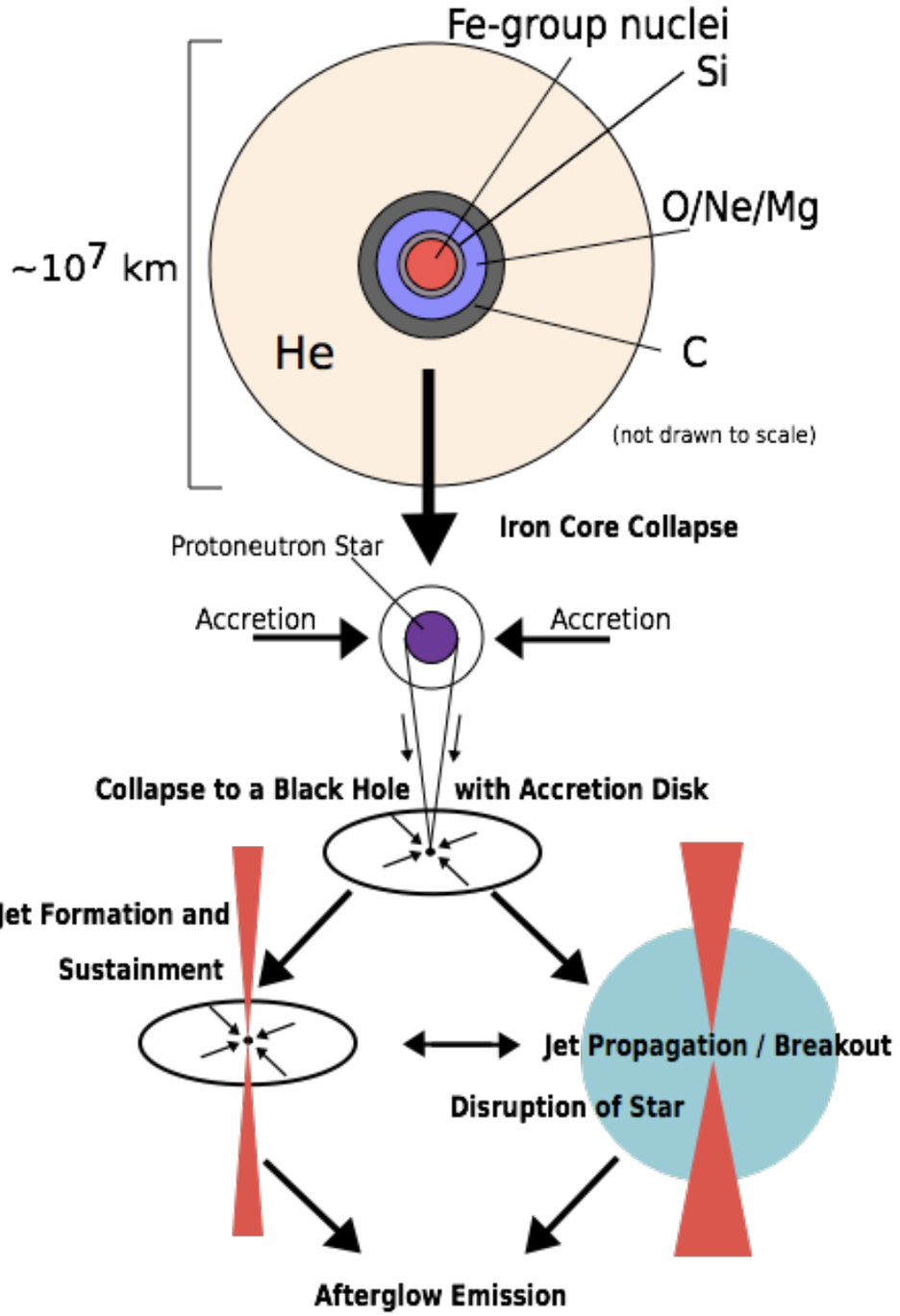
- Reliability, so that one trusts its results
- Extensibility, so that researchers can add and experiment with new ideas
- Usability, so that graduate students don't waste too much time
- Performance, because supercomputing time is expensive

Complex Simulations



- Real-world problems are complex, not just a single physics system
- Consequently, modern simulations may contain several models at once
- Each has its own set of PDEs and may have its own discretisation
  - How to handle this complexity?

Example: (Long) Gamma-Ray Bursts



- General Relativity (black hole!)
- relativistic hydrodynamics (star)
- microphysics, equation of state (shock wave)
- neutrino radiation (cooling, heating)
- magnetic fields (jet formation – mechanism not yet understood)
- photon radiation (afterglow)

Typical Research Scenario

- Different models are contributed by different people (each expert in his/her area), and then combined into a single code
- Physicists contribute models, mathematicians contribute discretisation methods
- Computer scientists need to contribute:
  - A software architecture that makes this possible in a safe yet efficient manner

Added Problems

- Example: Einstein Toolkit (not untypical)
- Code 12+ years old, grad students leave after 3 productive years, most original authors not available any more
- Developers distributed over many places in several continents
- Most physicists are not good programmers

einstein toolkit

- Goal: have state-of-the-art set of tools for NR available as open source
- Organised by Einstein Consortium, open to everyone
- See http://einsteintoolkit.org

Component Architecture

- Split program into independent *components*
- *Framework* provides lean glue between these
  - Each component is developed independently by a small group of developers
  - Only end user assembles all the code: no central control, no authoritative version

Core Einstein Toolkit
svn.einsteintoolkit.org

Cactus Flesh
and CCTK
svn.cactuscode.org

Einstein Toolkit
svn.partnersite.org

Carpet AMR
git.carpetcode.org

Tools, Parameter Files,
& Data
svn.einsteintoolkit.org

GetComponents

Assemble
Simulation

GetComponents

Research Group

Group Modules
cvs.groupthorns.org

Individual Modules
cvs.mythorns.org

Component Framework

- Basic principle: *control inversion*, where main program is provided by framework, and components look like libraries
  - no component is “more important”
- Framework itself does no real work, just glues components together
  - Components don’t interact with each other, only via framework

People

- 49 contributors over the past decade, both from physics and CS; many left the field by now
- currently 50 members from 14 sites in 7 countries
- 9 maintainers from 5 sites
- >200 publications, >30 theses building on these components
- about 1/3 of the talks in GR19 B2 session

Physics/Computational Components

- In the Einstein Toolkit, e.g. the following are components:
- Evolution systems (PDEs)
- Boundary conditions
- Initial conditions
- Time stepping method
- (Finite differencing)
- But also the following are components:
- Simulation grid (distributed array)
- I/O, output to file
- Simulation domain specification
- Termination condition
- Twitter client

Science Capabilities

- BSSN
  (phi, W; 1+log, Gamma driver; up to 8th order)
- GR Hydro
  (based on Whisky; Valencia formulation)
- BH / NS initial data
  (TwoPunctures, Lorene)
- Excision / Turduckening
- Runge-Kutta a.o.
- AMR
- Horizon finder
- Wave extraction
- MPI, OpenMP
- HDF5 output, visualisation

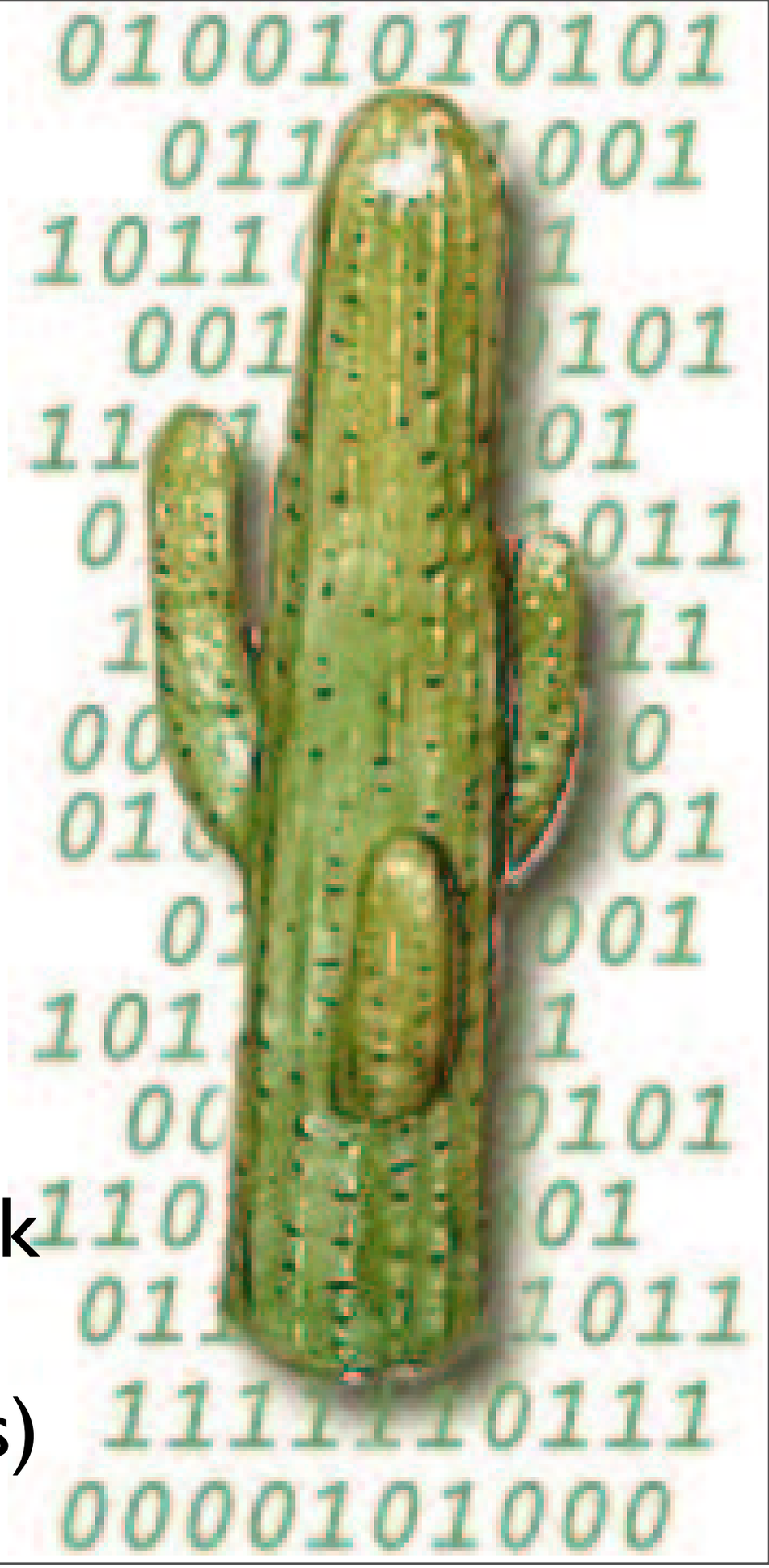
Component Model Summary

- Modern simulation codes are complex, contain more than just one physics model
- Component model can provide necessary abstraction and encapsulation
- Software *Framework* provides glue between components
- Important for research: enables loosely coupled long-distance collaborations

Part 2: Cactus Software Framework

Cactus

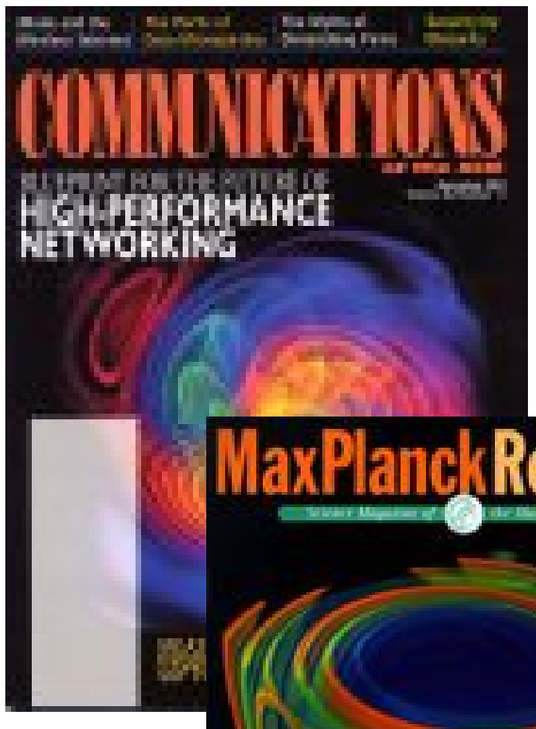
- Open source HPC software framework developed at LSU/ AEI/Caltech
- First version designed in 1997 to address simulation code difficulties described earlier
- Highly portable (from notebook to supercomputer), highly parallel (tested on >130k cores)

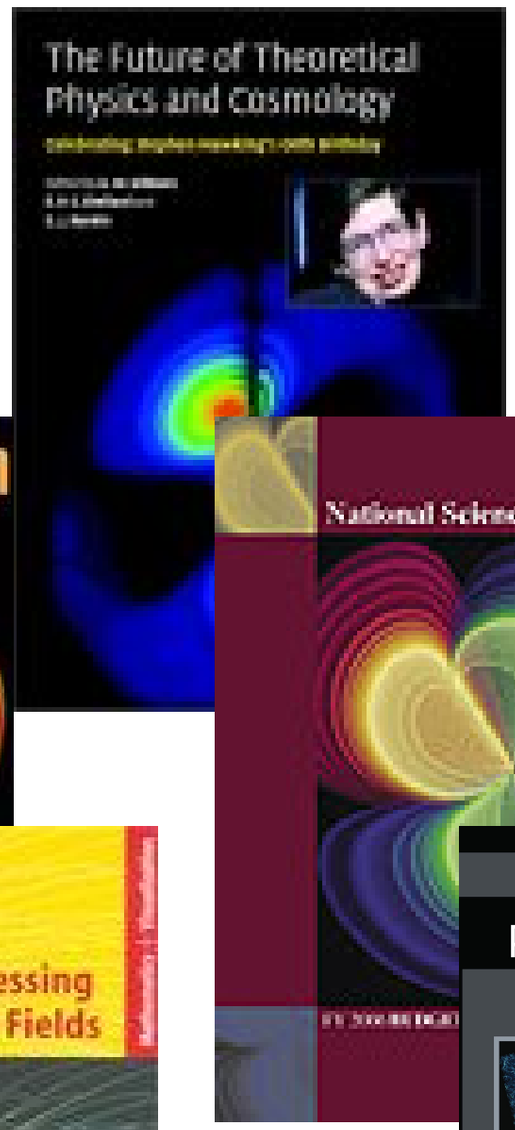


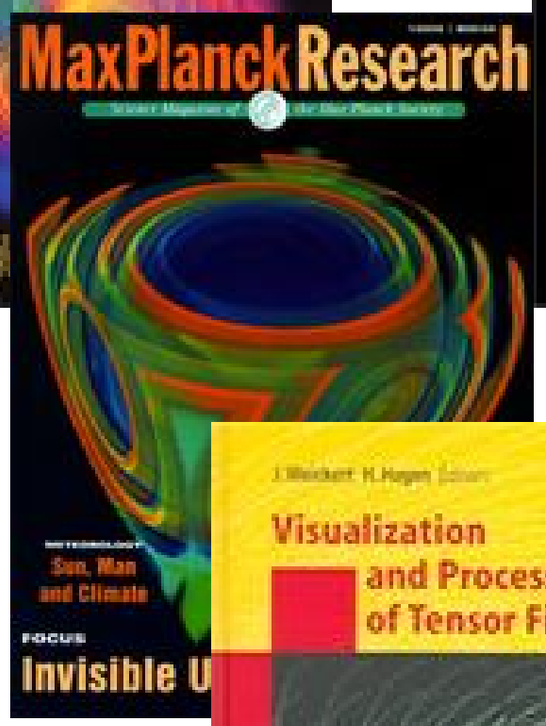
Cactus Design

- The core of the framework (*flesh*) is lean, provides only glue between components, performs no "real work"
- The components (*thorns*) perform all the work, both computational and physics
- Many standard thorns, e.g. for coordinate systems, standard boundary conditions, efficient parallel I/O, etc.

Covers


COMMUNICATIONS
HIGH-PERFORMANCE NETWORKING

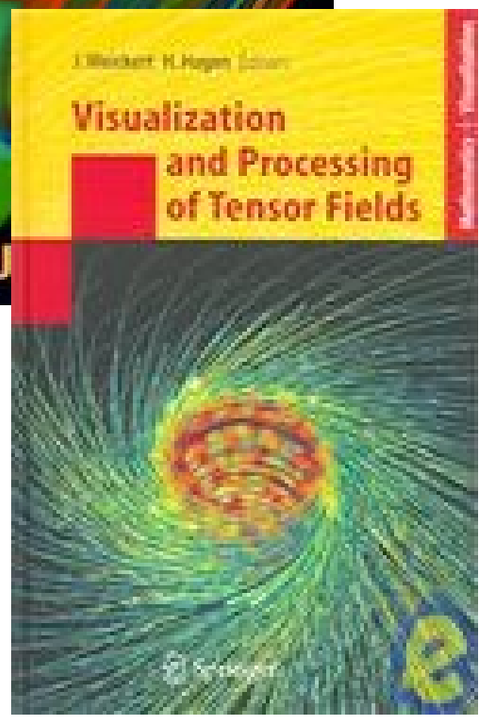

The Future of Theoretical Physics and Cosmology

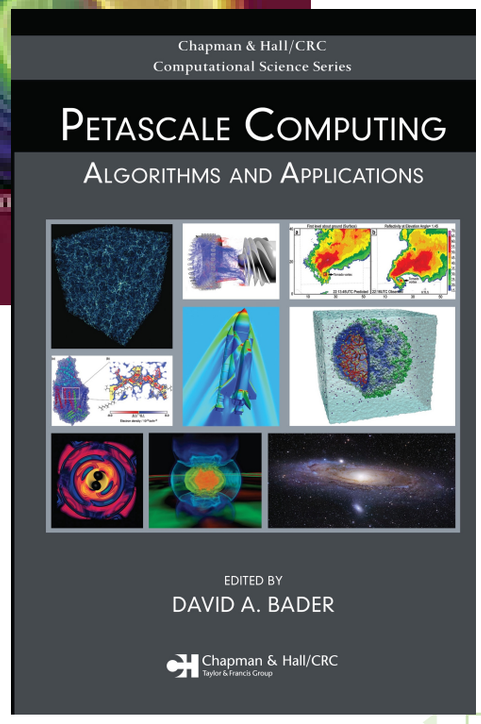

MaxPlanckResearch
Sun, Man and Climate
FOCUS
Invisible U

National Science Foundation

from the grou

National Science Board
Science and Engineering Indicators 2006
Volume 1

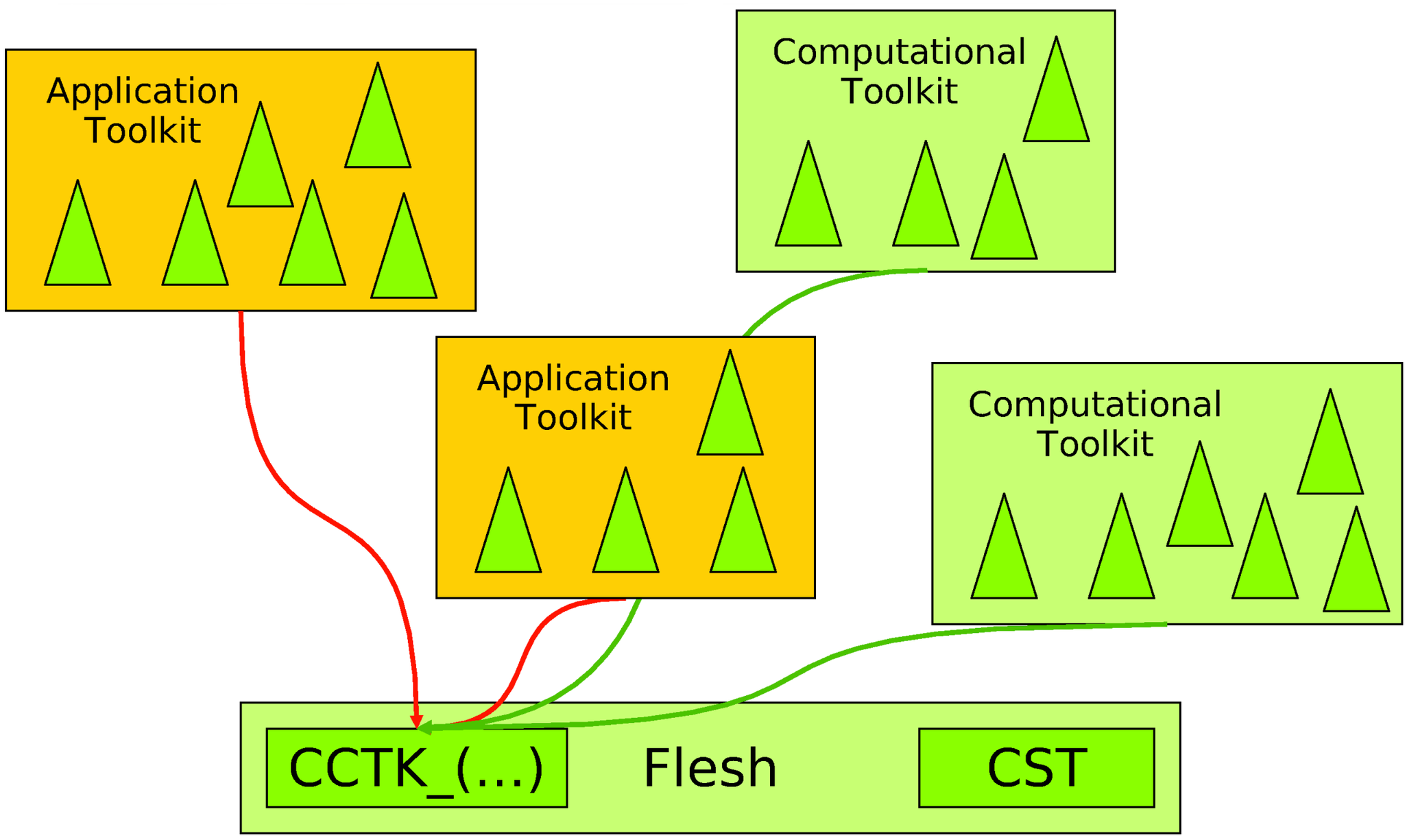

Visualization and Processing of Tensor Fields


Chapman & Hall/CRC
Computational Science Series
Petascale Computing
Algorithms and Applications
EDITED BY
DAVID A. BADER
Chapman & Hall/CRC


Naturwissenschaftliche Rundschau
NR 689

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
  - (most are libraries, not frameworks)
- Frameworks are also common in other software areas
  - e.g. KDE (desktop), Eclipse (software development)

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 compiled thorns
- At run time, thorns can be activated if needed for the particular simulation

Thornlists

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

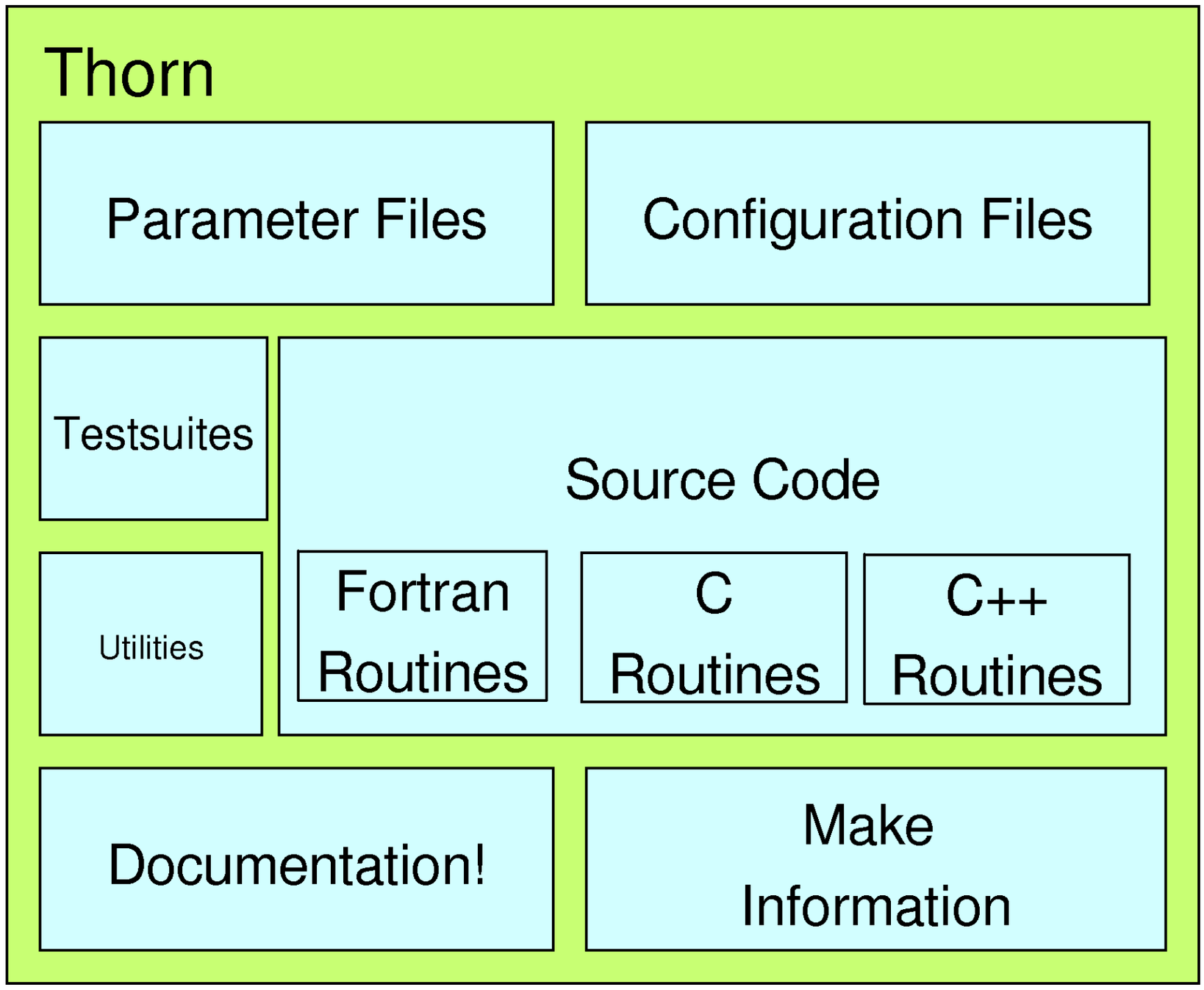
```
!CRL_VERSION = 1.0

# 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

Three configuration files per thorn:

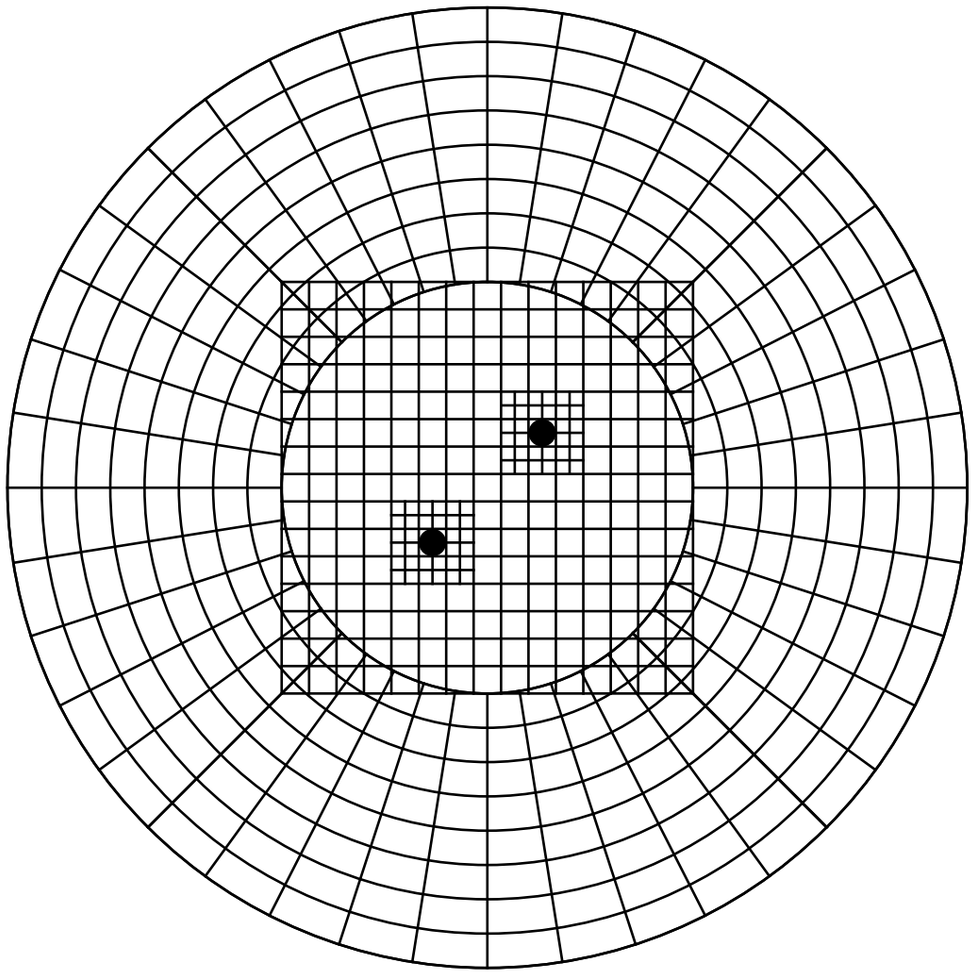
- interface.ccl declares:
  - an 'implementation' name
  - inheritance relationships between thorns
  - Thorn variables
  - Global functions, both provided and used
- schedule.ccl declares:
  - When the flesh should schedule which functions
  - When which variables should be allocated/freed
  - Which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

Grid Functions

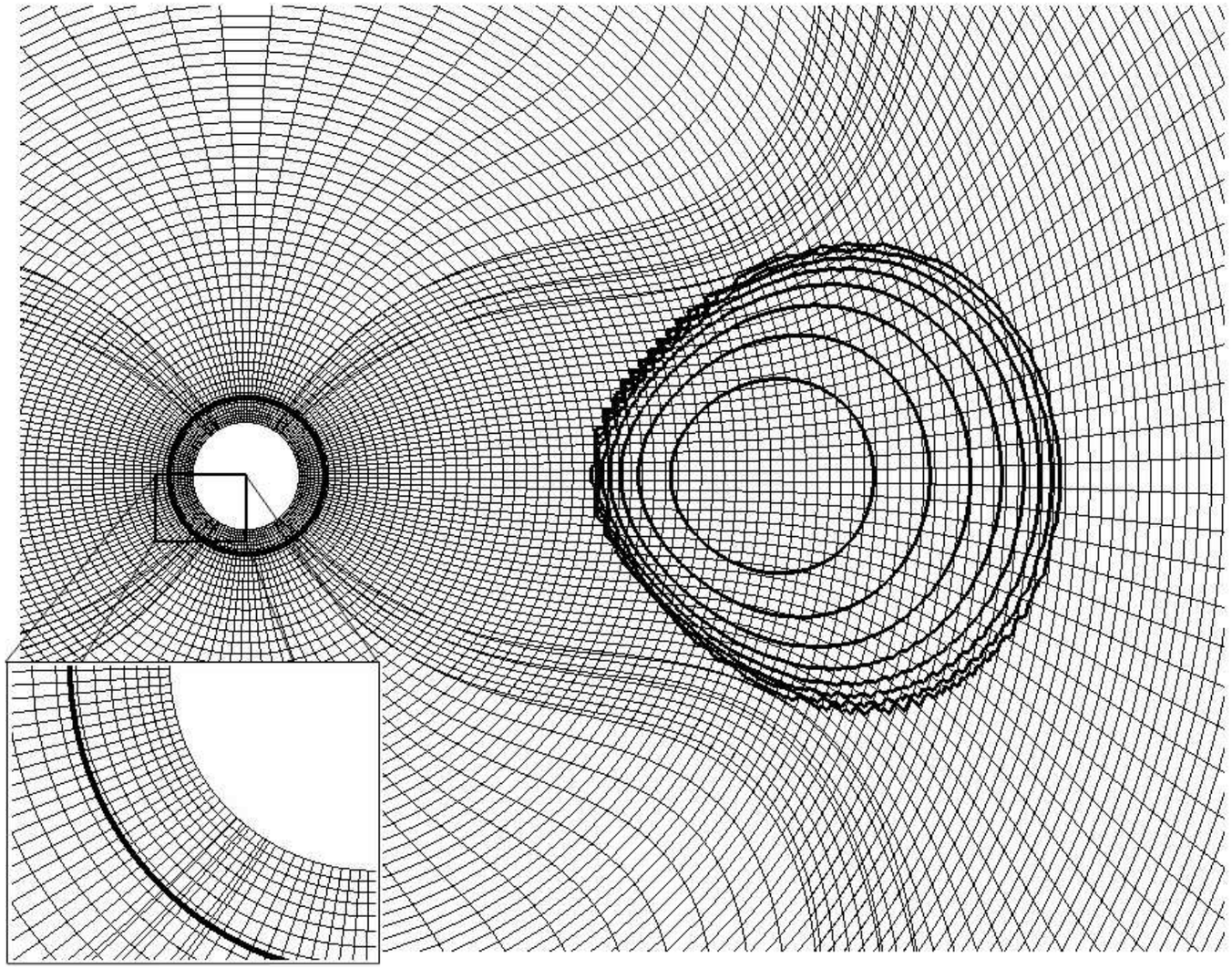
- A discretised physical quantity (density, pressure, velocity) lives on a grid function
- If a regular discretisation is used, these are essentially distributed arrays (hence “grid”)
  - (Could also be graph, tree, etc. instead)
- Grid functions are fundamental data types in a simulation

Example Cactus Grid Function Shapes

Adaptive Mesh Refinement and Multi-Block



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! ("manual assembly"); end user probably doesn't understand thorn details

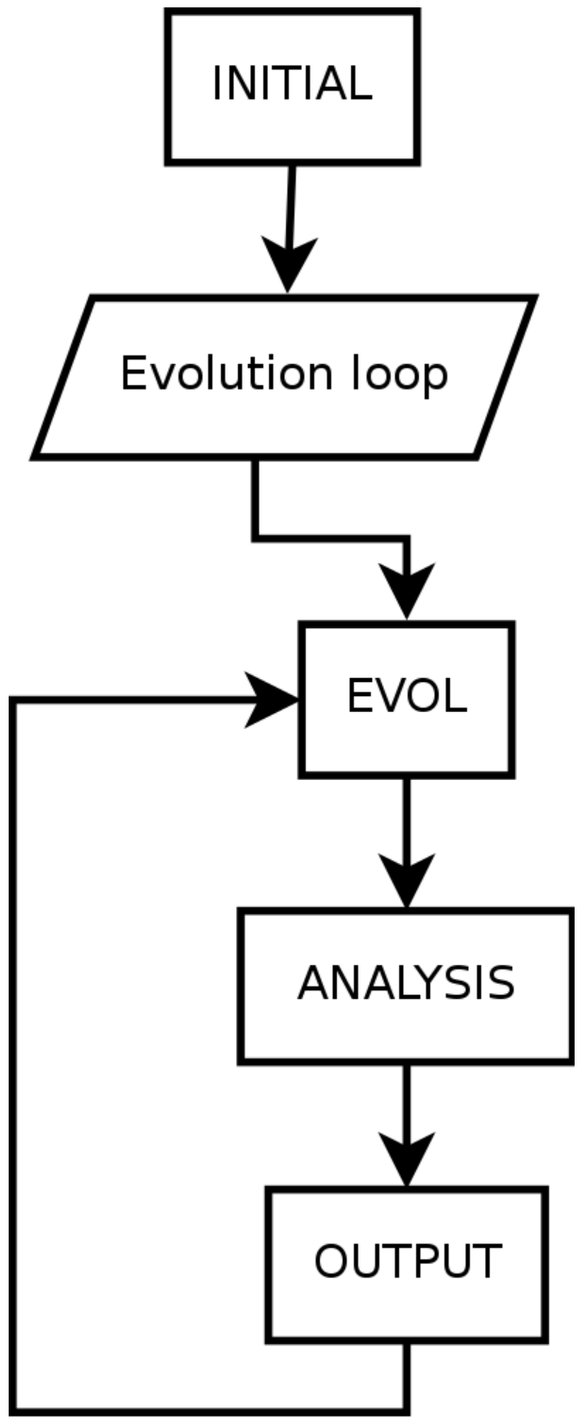
schedule.ccl

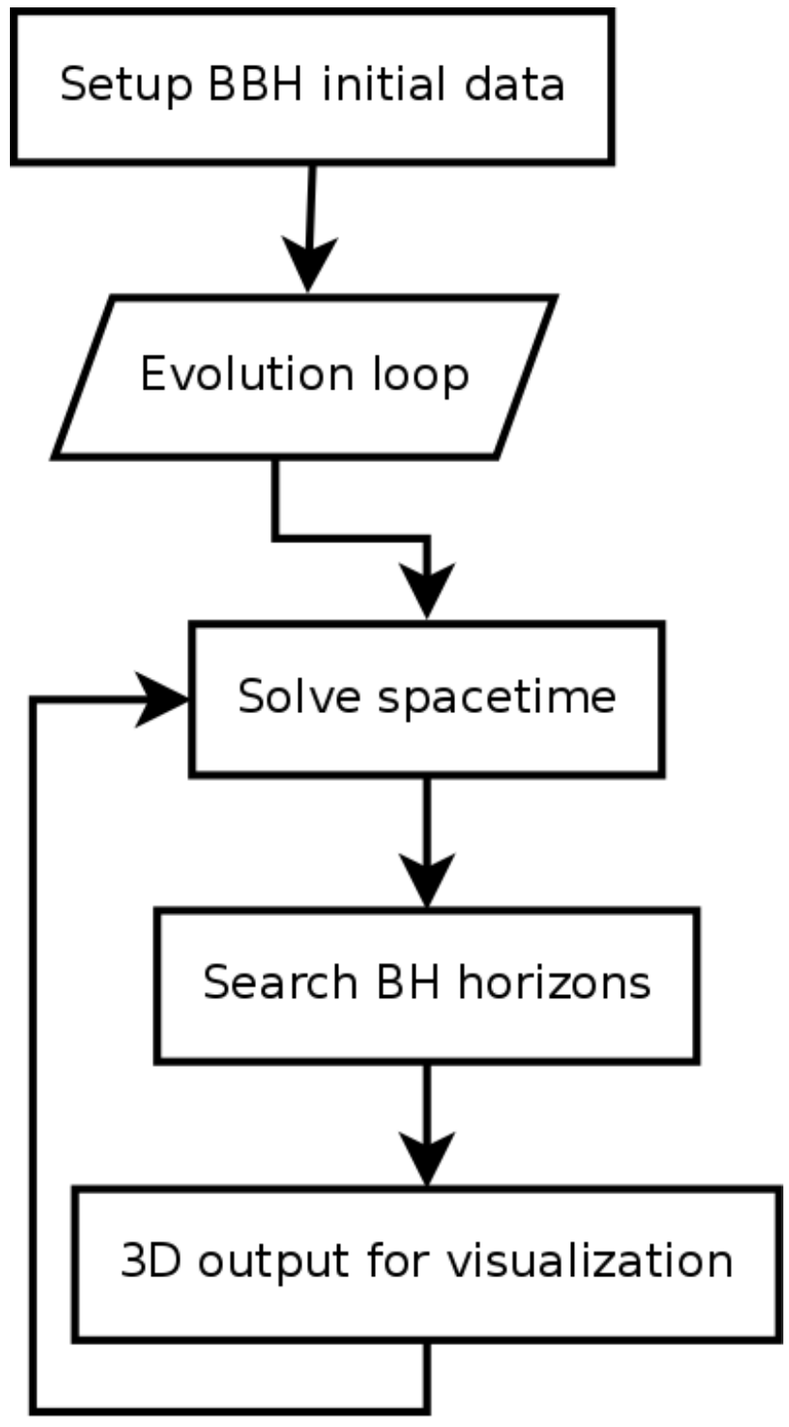
- Flesh contains a flexible rule based scheduler
- Order is prescribed in `schedule.ccl`
- Scheduler also handles when variables are allocated, freed or syncronized between parallel processes
- Functions or groups of functions can be
  - grouped and whole group scheduled
  - scheduled before or after 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.

`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, debug 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 extened 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
```

Running Hello World as Example Program

- Begin with a simple C program
- Convert it to a Cactus thorn
- Design the thorn CCL files
- Write a parameter file (that also activates this 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
- REAME:

```
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;
}
```

- make.code.defn:

```
SRCS = HelloWorld.c
```

Hello World Thorn

- parameter file:

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

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

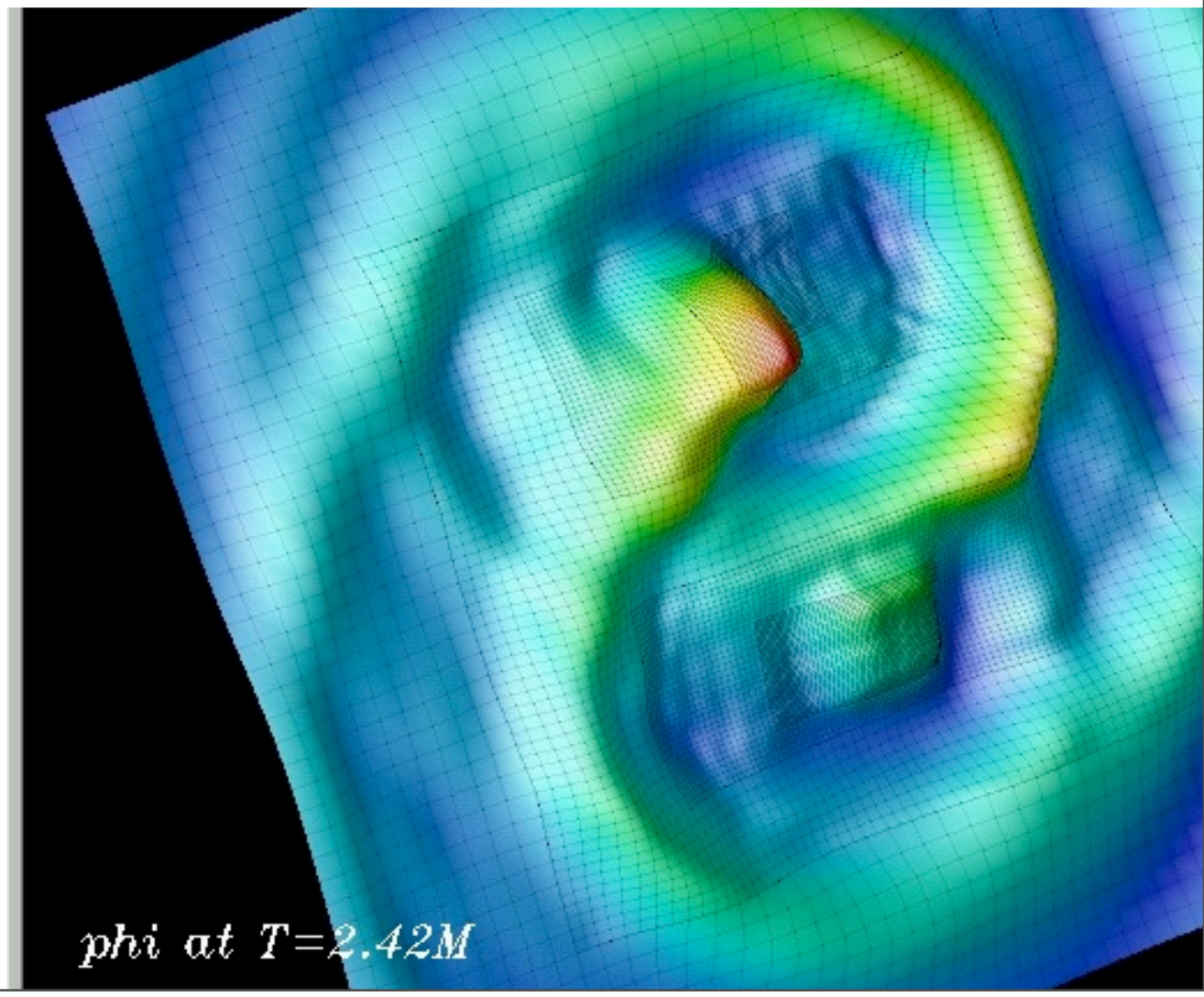
Hello World Thorn

- 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:      May 06 2009 (13:15:01)
Run date:          May 06 2009 (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!
[...] 8x
--------------------------------------------------------------------------------
Done.
```

WaveToy

- WaveToy is our example application for Cactus
- WaveToy evolves the scalar wave equation (e.g. sound propagation)
- This is also a (very) simple model for the Einstein Equations
- Already discussed WaveToy discretisation and parallelisation



WaveToy Thorn

Directory structure:

```
WaveToy/
|-- COPYRIGHT
|-- README
|-- configuration.ccl
|-- doc
|   `-- documentation.tex
|-- interface.ccl
|-- schedule.ccl
|-- param.ccl
`-- src
    |-- WaveToy.c
    `-- make.code.defn
```

WaveToy Thorn

- `interface.ccl`:

```
IMPLEMENTS: wavetoy_simple
INHERITS: grid

PUBLIC:

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

CCTK_INT FUNCTION Boundary_SelectVarForBC( \
  CCTK_POINTER_TO_CONST IN GH, CCTK_INT IN faces, \
  CCTK_INT IN boundary_width, CCTK_INT IN table_handle, \
  CCTK_STRING IN var_name, CCTK_STRING IN bc_name)

REQUIRES FUNCTION Boundary_SelectVarForBC
```

WaveToy Thorn cont.

- `schedule.ccl`:

```
STORAGE: scalarevolve[3]

SCHEDULE WaveToy_InitialData AT CCTK_INITIAL
{
  LANG: C
} "Initial data for 3D wave equation"

SCHEDULE WaveToy_Evolution AT CCTK_EVOL
{
  LANG: C
  SYNC: scalarevolve
} "Evolution of 3D wave equation"

SCHEDULE WaveToy_Boundaries AT CCTK_EVOL AFTER WaveToy_Evolution
{
  LANG: C
} "Select boundary conditions for the evolved scalar"

SCHEDULE GROUP ApplyBCs as WaveToy_ApplyBCs AT CCTK_EVOL AFTER WaveToy_Boundaries
{
} "Apply boundary conditions"
```

WaveToy Thorn cont.

- `param.ccl`:

```
CCTK_REAL amplitude "The amplitude of the waves"
{
 *:* :: "Anything"
} 1.0

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

CCTK_REAL sigma "The sigma for the gaussian wave"
{
 0:* :: "Positive"
} 0.1
```

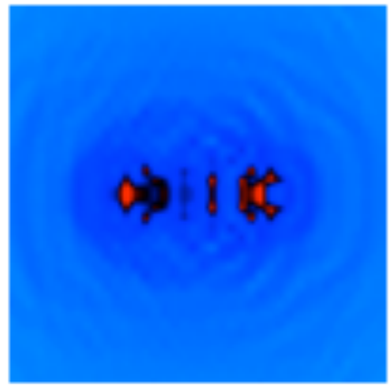
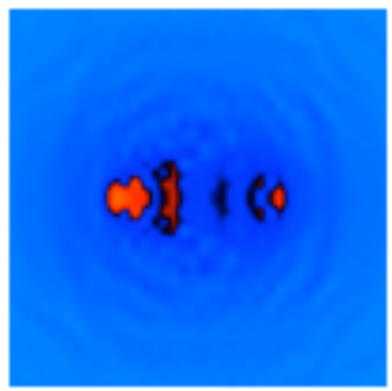
WaveToy Thorn cont.

- Example parameter file:

```
Cactus::cctk_run_title = "Simple WaveToy"

ActiveThorns = "time boundary Carpet CarpetLib CartGrid3D"
ActiveThorns = "CoordBase ioutil CarpetIOBasic CarpetIOASCII"
ActiveThorns = "CarpetIOHDF5 SymBase wavetoy"

cactus::cctk_itlast = 10000
time::dtfac = 0.5

IO::out_dir             = $parfile
IOBasic::outInfo_every  = 1
IOASCII::out1D_every    = 1
IOASCII::out1D_vars     = "wavetoy_simple::phi"

iohdf5::out_every = 10
iohdf5::out_vars        = "grid::coordinates{out_every=10000000} wavetoy_simple::phi"
```

Live WaveToy Demo

- There is a Cactus thorn that implements a web server
- This allows you to watch simulations as they run
- Live demo at http://www.cactuscode.org/demo/

Variable Slice	Description	Image
WAVETOY::phi `xz_[20]`	*Jpegs of slices*	
WAVETOY::phi `yz_[20]`	*Jpegs of slices*	

Summary

- Introduced Cactus Framework
- Thorns have implementation (regular code) and interface (ccl files)
- Introduced ccl file syntax; see users' guide and examples for details
- To run a simulation, need a parameter file that also activates participating thorns

Homework/Project: Design and Implement a Cactus Thorn

Physics: Calculate Kinetic Energy

- Physics problem: Find out "where" a star is oscillating
- Method: Calculate kinetic energy density in a neutron star simulation, see where it is largest
- Kinetic energy: is large (a) when velocity is large, or (b) when large mass is moving

Equations

- Kinetic energy density: e_kin = 1/2 ρ v^2
- Velocity *v* is vector [vx, vy, vz]
- Remember: $v^2 = vx^2 + vy^2 + vz^2$
- The quantities ρ, vx, vy, vz are defined in thorn HydroBase

Algorithm

- Kinetic energy density is defined at every grid point; needs to be stored in a grid function
- Calculate by iterating over all grid points...
  - ... and then evaluating the formulae from the previous slide for each grid point

Accessing Grid Functions

- See Cactus users' guide, section C.1.6.2
- For a Fortran example, see also e.g. thorn EinsteinInitialData/GRHydro_InitData, file src/GRHydro_ShockTube.F90
- Look also at C/C++/Fortran implementations of Cactus WaveToy for further examples

Thorn CCL Files

- interface.ccl:
  declare grid functions
  declare dependencies on other thorns's interfaces

- schedule.ccl:
  schedule routines
  request storage

- param.ccl:
  nothing to do here today

interface.ccl

- Inherit from implementation HydroBase which defines ρ and *v*
- Declare new grid function for kinetic energy
  - Also update thorn list to actually build the thorn

schedule.ccl

- Schedule routine in `analysis` bin that performs the actual calculation
  - Also update makefile fragment to actually compile the code
- Request storage for new grid function for this calculation

Running the Simulation

- Plan: Take existing simulation, augment it by adding the "kinetic energy density" analysis step
- Begin with Neutron Star example simulation from Einstein Toolkit tutorial (see last homework)

Parameter File

- Update existing parameter file that defines the simulation (.par file, different from param.ccl) to activate the new thorn
  - Also output the new variable in the parameter files
- Run, then visualise result

Debugging

- Build errors: Build without simfactory, but with `SILENT=no` make option
- Run-time errors: Configure with `--debug` option
- Output more variables in the parameter file
- Speed up simulations by reducing the number of iterations in the parameter file