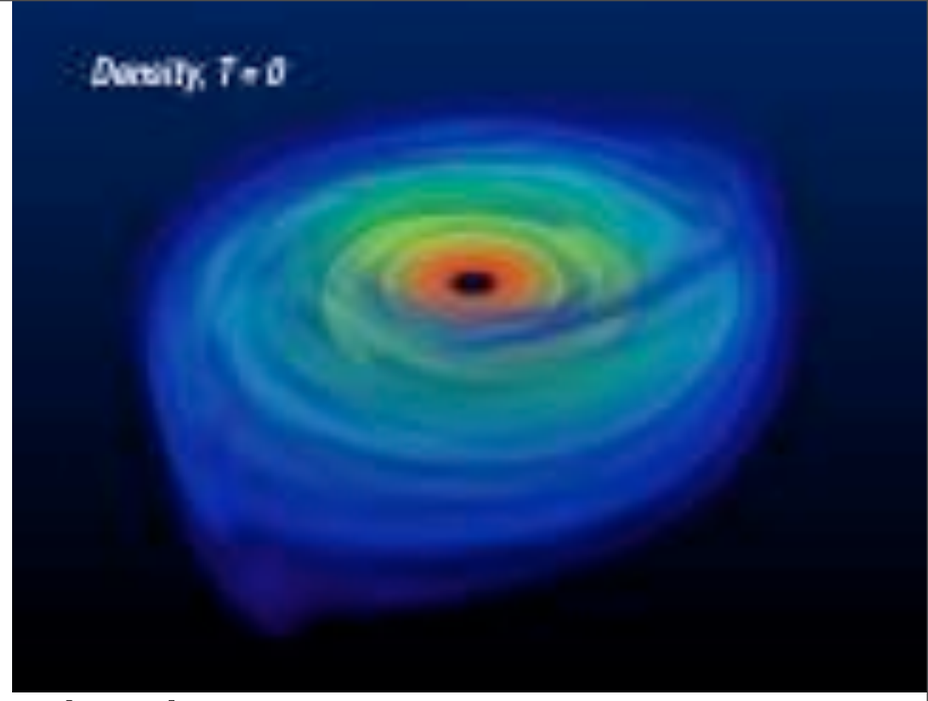
CSC 7700: Scientific Computing
Module C: Simulations and Application Frameworks
Lecture 2: Simulating Complex Systems

Dr. Erik Schnetter

Gnuplot Introduction
(leftover bits from module A)

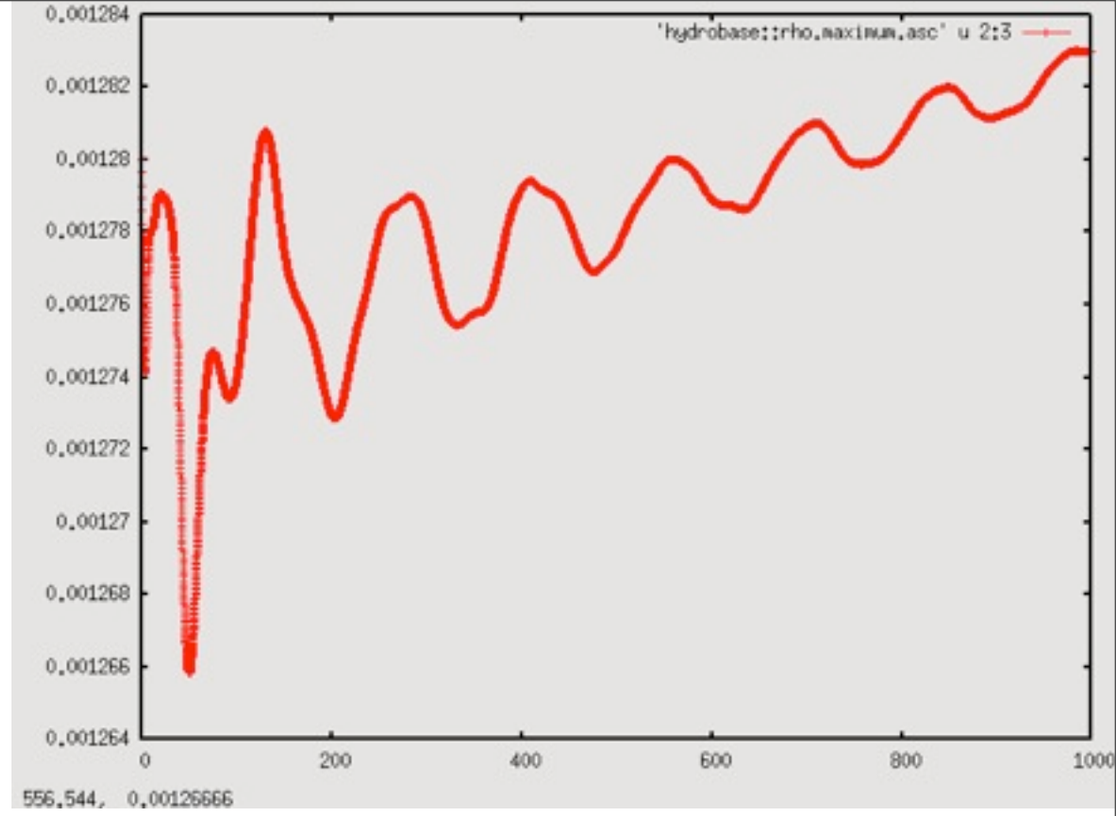
Visualization



- Simulation results need to be analysed
- Visualization is very intuitive analysis method
- Many and often quite different possibilities
  - different dimentionality, e.g. 1D plots vs. 3D videos
  - different data formats
  - different visualization tools
- Ranges from 1D plots of kByte of data to 3D rendering of TBytes

Visualization, gnuplot

http://www.gnuplot.info/



- Simple 1D/2D plotting tool
- Command line interface
- Text input (or anything which can be on-the-fly be converted to text)
- Many output formats (screen, pdf, eps, latex, png, ...)
- Very customizable, but no shallow learning curve
- Mostly used for debugging or final 1D graphs for papers

Alternatives: Mathematica, Maple, ...

Visualization, gnuplot

Command	Effect
`plot [0:10] sin(x)`	Plots $\sin(x)$ in the x-range $[0, 10]$
`plot 'data.dat' using 1:3`	Plots columns 1 and 3 of file `data.dat` as x and y
`plot 'data.dat' with lines`	Plots columns 1&2 using lines
`plot '<bzcat data.dat.bz2'`	Decompresses file `data.dat.bz2` on the fly and plot content
`set terminal postscript eps`	Specify eps output format
`set output "plot.eps"`	Set output filename to `plot.eps`

http://cactuscode.org/documentation/visualization/gnuPlot/

Visualization, gnuplot

Consider the following example data file ("data") where the first, second, third, and fourth columns hold the x, y, min x, and max x values, respectively:

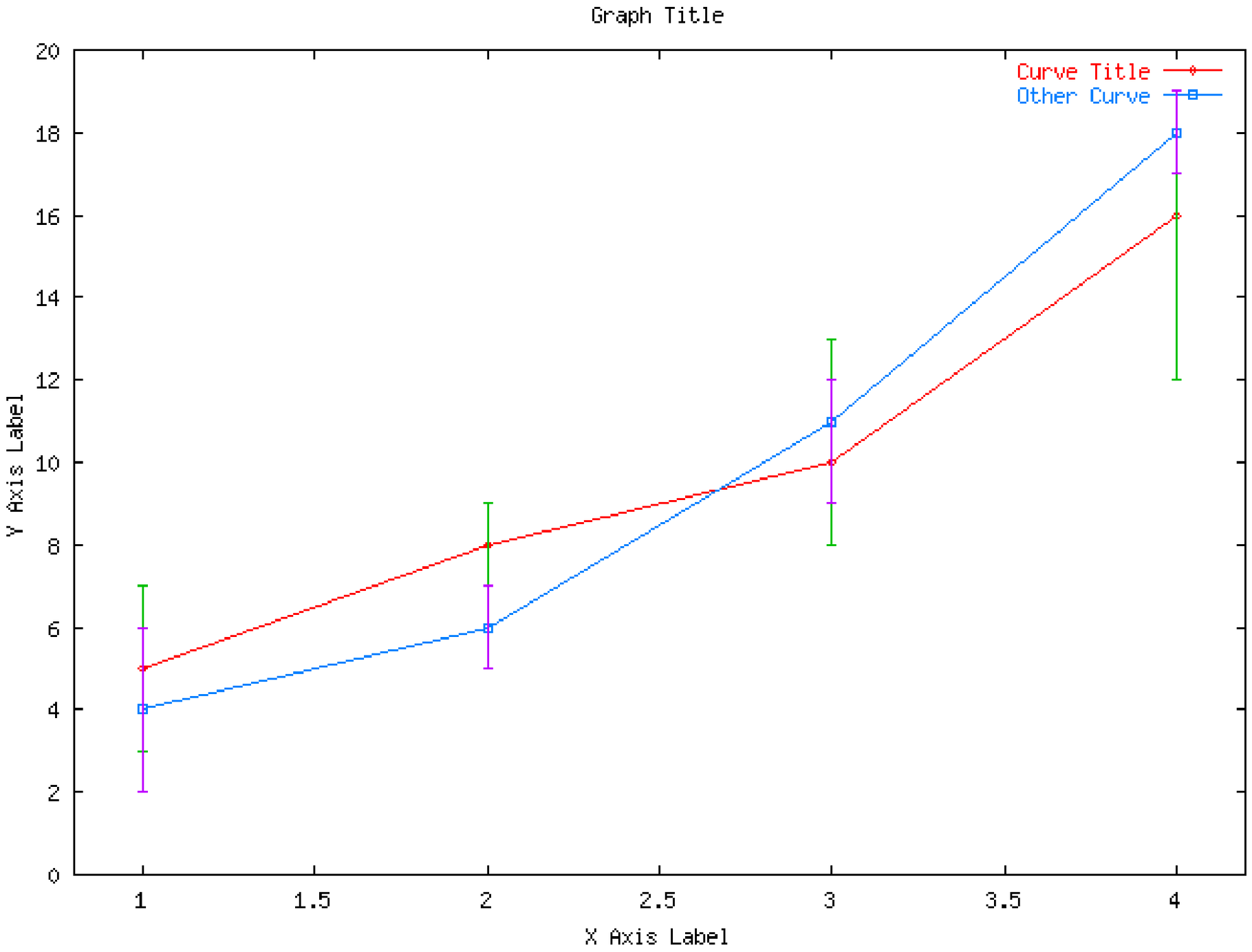
```
1 5 3 7 4
2 8 5 9 6
3 10 8 13 11
4 16 12 19 18
```

You can plot this with following gnuplot command script:

```
set title "Graph Title"
set xlabel "X Axis Label"
set ylabel "Y Axis Label"
set term gif
set output "2D.gif"
# To make postscript do
# set term postscript eps
# set output "2D.eps"
set data style lp
plot [.8:4.2] "2D.data" using 1:2 t "Curve Title", \
              "2D.data" using 1:2:3:4 notitle with errorbars lt 1 ps 0, \
              "2D.data" using 1:5 t "Other Curve", \
              "2D.data" using 1:5:6:7 notitle with errorbars lt 1 ps 0
```

This generates a gif file "2D.gif" ...

Visualization, gnuplot



Simulating Complex Systems

- Lecture 1 described the physicist's (application scientist's) point of view
- This lecture discusses *computer science issues* in simulations and simulation codes
- In most research groups, a computer scientist is an expensive luxury;
  only large projects can afford computer scientists

Simulation Science Basics

- To go from physics to a simulation, one usually
  1. finds *PDEs* expressing the physics;
  2. *discretises* these PDEs;
  3. *implements* these on a supercomputer
- Many simulation codes have a similar structure
- Many supercomputers have a similar architecture

Today's Goal

- Discuss two computer science issues in simulations:
  - Parallel computing (algorithm design)
  - Component model (software engineering)
- Both are crucial for large-scale real-world simulation software

Part 1: Parallel Computing

MPI:
Message Passing Interface

MPI

- MPI is an API; it is **THE** industry standard for parallel HPC programming
- Supported on all important HPC platforms
- Very successful (standard since 1994) since it makes it *possible* to implement efficient parallel algorithms
- I didn't say MPI makes it *easy* to do so
- www.mpi-forum.org

HPC History

- Before MPI: *Vector architectures*, e.g. Cray Y-MP (until ~1992)
- Each instruction acts on 100s or 1000s of data elements "simultaneously" (SIMD)

- Much more efficient than scalar processor (compare conveyor belt vs. hand assembly)
- Disadvantages: too inflexible for dynamic data structures, too expensive due to custom-designed (low-volume) hardware

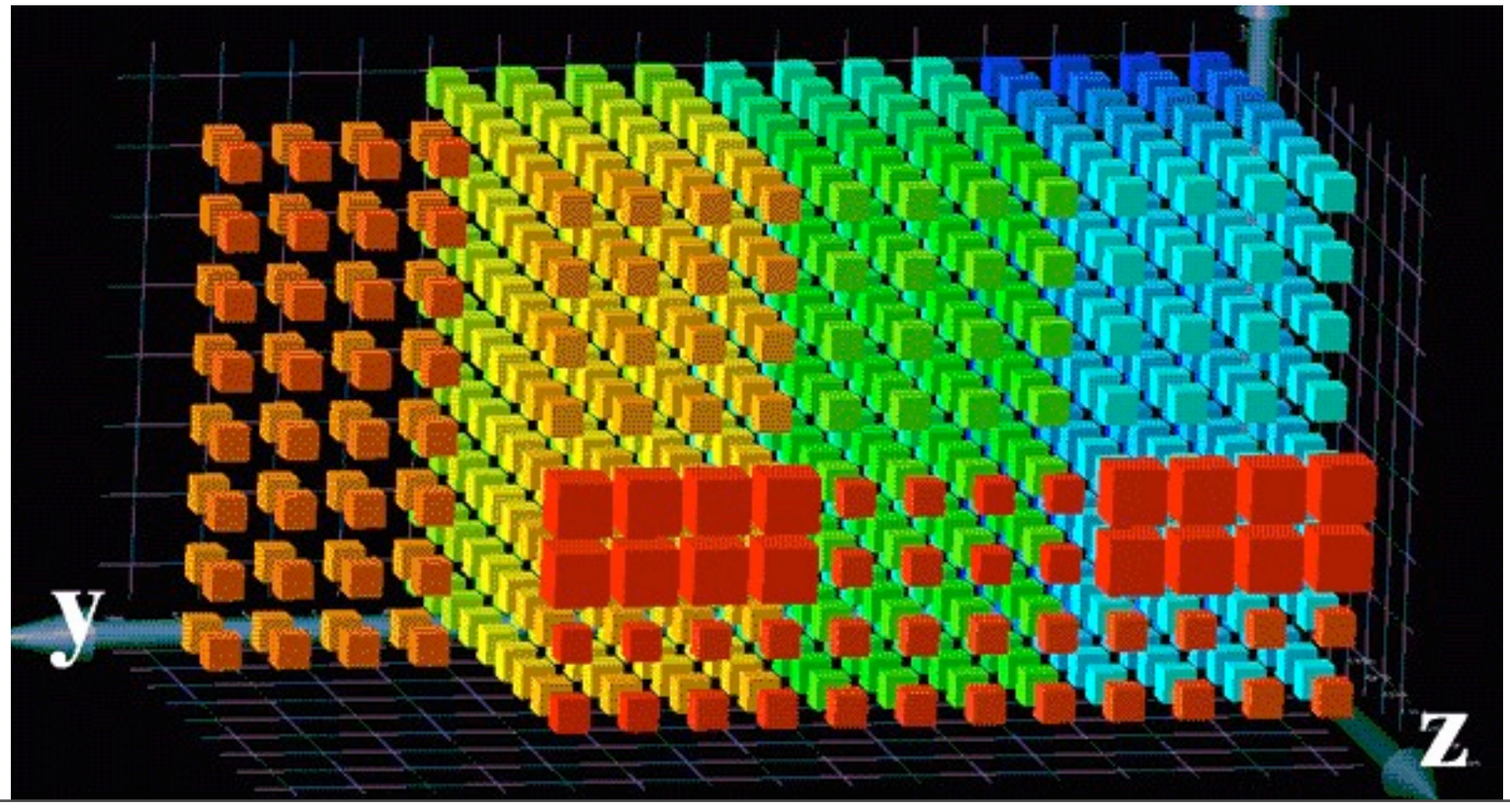
HPC History, Cont'd

- After vector machines, *cluster architecture* became prevalent (Cray T3D, 1993)

- Basic idea: have many simple nodes, connected by high-speed network

- Nodes need to communicate (*exchange messages*) during computation

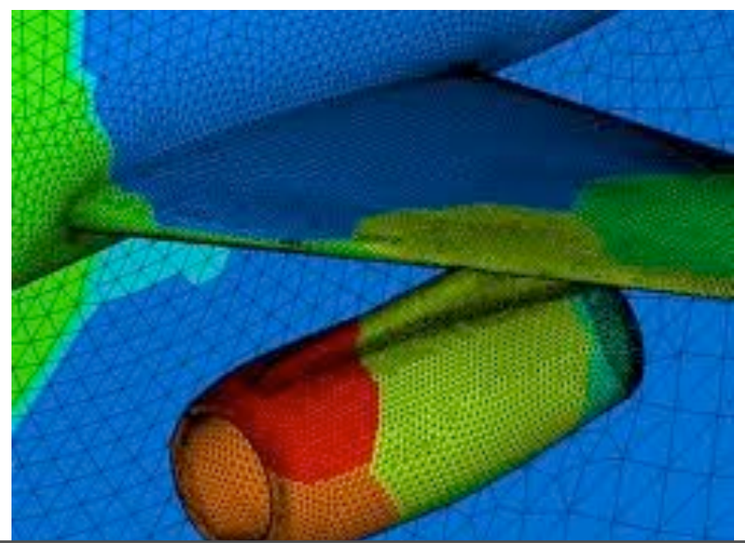
- Also called *Beowulf architecture*, esp. if only a cheap network is used



Node connectivity in a Cray T3E

MPI Programming

- Each process runs an independent copy of the program
- Each copy has a unique number assigned to it (0...*N*-1)
- The program needs to divide the total workload into N pieces, and assign one to each process



- The processes can talk to each other *only* by exchanging messages
- MPI hides low-level, system-dependent communication details from the programmer
- MPI messages are (by default) *ordered* and *reliable*

Examples of Real Life Message Passing

- Message Passing is very common, even in the real world; for example:
  - Letters via the post office
  - Email
  - Phone text messages
  - Newspapers
  - Chinese whispers game
  - Monopoly
- However, these are NOT message passing – they are *streams* or *interactive* instead:
  - Phone conversation
  - Watching TV
  - Google Docs
  - Charades game
  - WoW

Distributing Data Structures

- Example: Distributing a (large) array over multiple MPI processes
- Assuming: 50 elements, 5 processes, thus each process *owns* 10 elements
- Arrays support two operations: set-element and get-element; we need to implement these with MPI, so that each process can access non-local elements

Distributed Array Implementation

- set-element(n,x):

1. determine which process owns element n
2. send message to that process containing index n and value x
3. check whether message has been received
4. if so, set element n to x

- x = get-element(n):

1. determine which process owns element n
2. send message to that process with index n
3. check whether message has been received
4. if so, get element n ...
5. ... and send message back with value x
6. wait for result

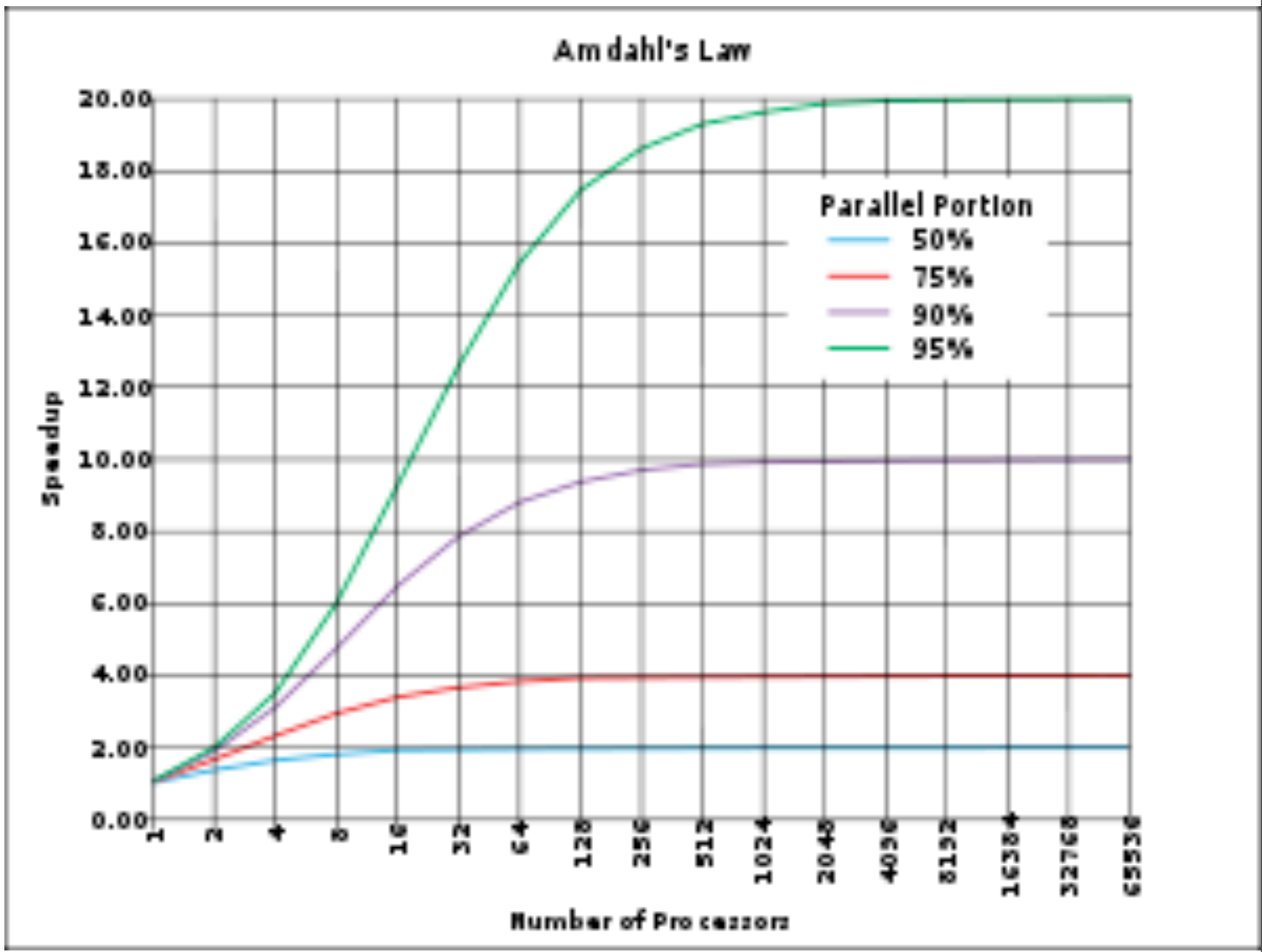
In reality, would not send individual elements,
but would send *batches* of elements to reduce latency
(which would change the API; see e.g. PETSc's SetValue function)

Distributing Data Structures, Cont'd

- Gee, this is complicated!
- (... and I didn't even go into the details ...)
- Yes, but if you invest the effort, it can be very efficient
- Most alternatives to MPI are less efficient, especially on 1000+ (10,000+) processes
- Alternatives to MPI are e.g. Co-Array Fortran (CAF), Unified Parallel C (UPC)

Amdahl's Law

- When running on *N* processes, not necessarily *N* times as fast – overhead
- Overhead determines maximum possible parallel speedup



100,000-fold speedup requires >99.999% parallelisation

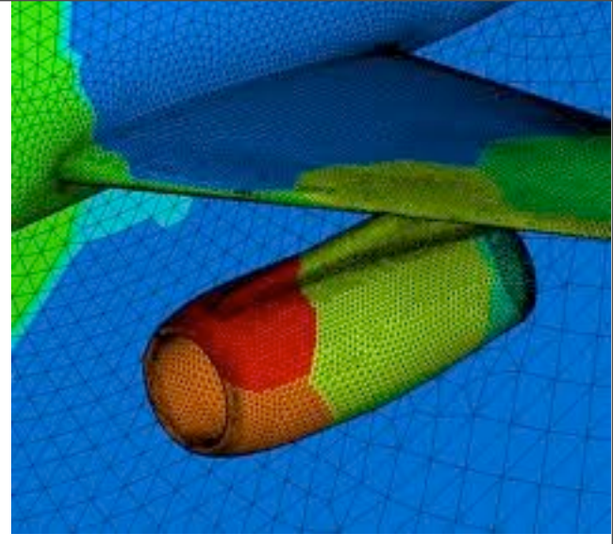
MPI Parallelisation Summary

- Efficient MPI parallelisation is complex and tedious
- Requires re-designing data type layouts and APIs (and then rewriting program)
- To ensure correctness, need good encapsulation of parallelism (and understanding of principles of *Algebraic Data Types*, "containers" in C++ STL)

Domain

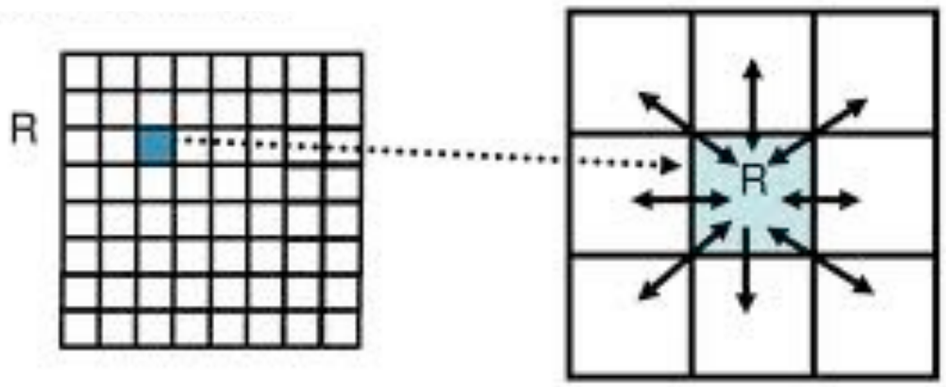
Decomposition

Fundamental Principle



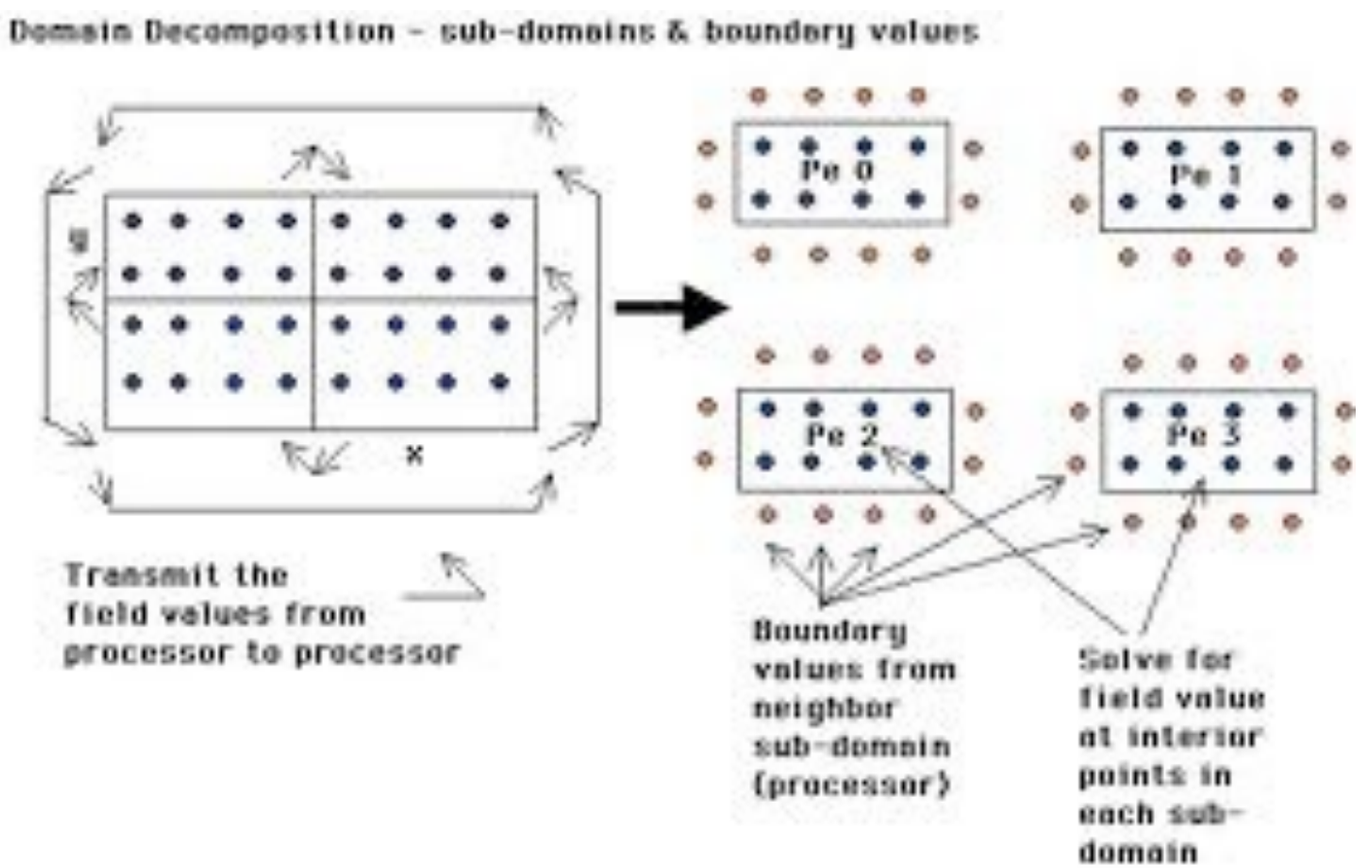
- In a domain decomposition, the discrete elements (points, cells, particles, ...) are distributed over all processes
- Each process handles only those that it *owns* (without requiring communication)
- Accessing neighbouring elements requires communication (e.g. at domain boundaries)

Typical Communication Pattern



2D: Each process communicates data to its 8-neighbors

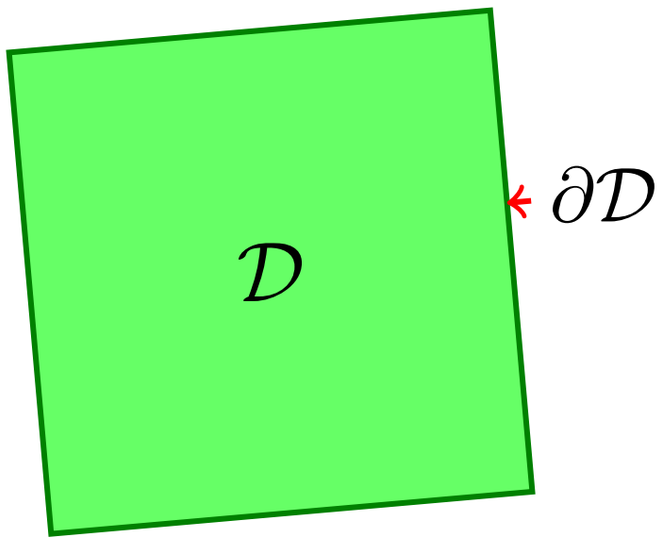
Ghost zones contain copies of neighbouring processes' grid points



Ghost zones are filled via inter-process communication from the corresponding *owner*

WaveToy Thorn: Wave Equation

For a given source function $S(x, y, z, t)$ find a scalar wave field $\varphi(x, y, z, t)$ inside the domain $\mathcal{D}$ with a boundary condition:



- inside $\mathcal{D}$:

$$\frac{\partial^2\varphi}{\partial t^2} = c^2\Delta\varphi + S$$

- on the boundary $\partial\mathcal{D}$:

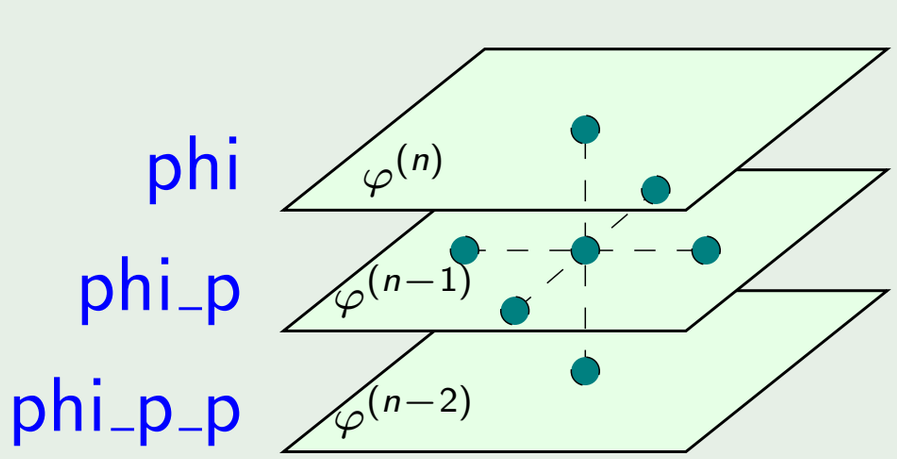
$$\varphi|_{\partial\mathcal{D}} = \varphi(t = 0)$$

WaveToy Thorn

Thorn structure:

interface.ccl

- grid function phi[3]:



- Boundary_SelectVarForBC

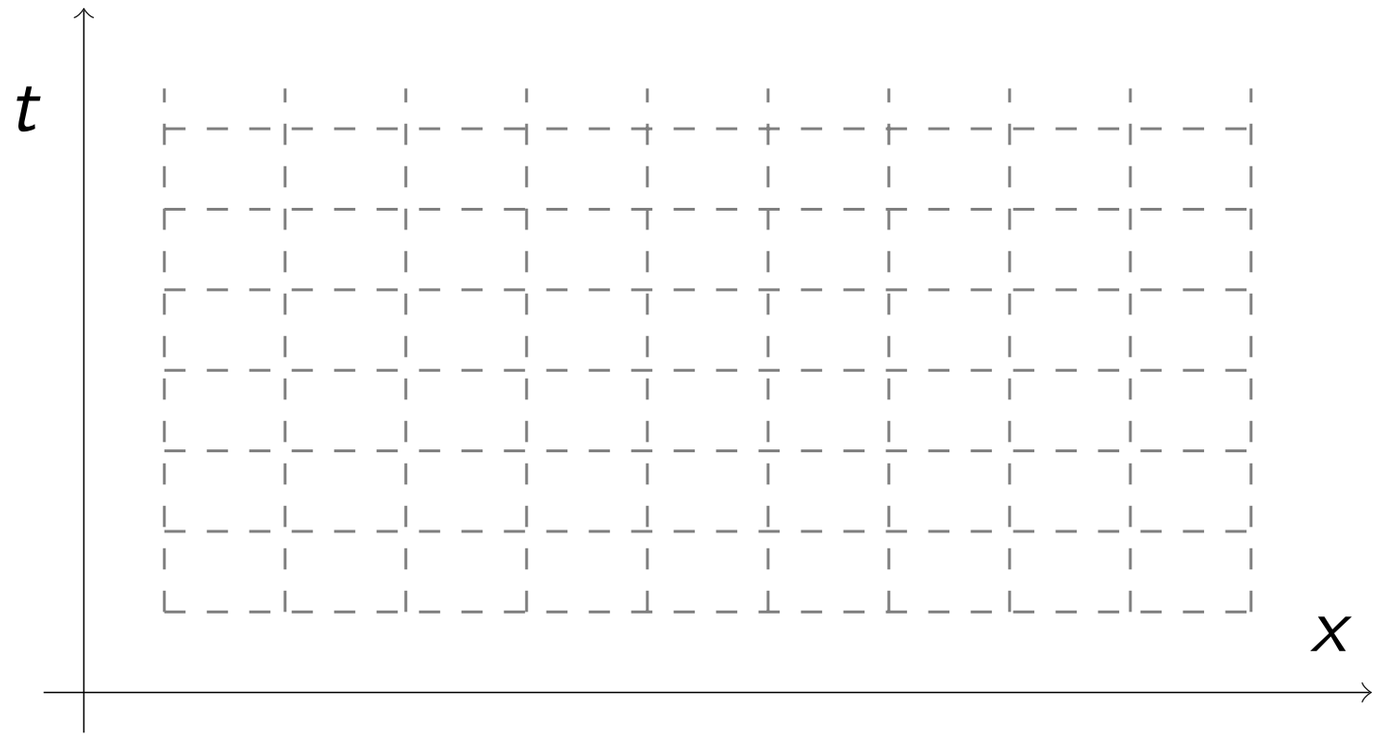
param.ccl

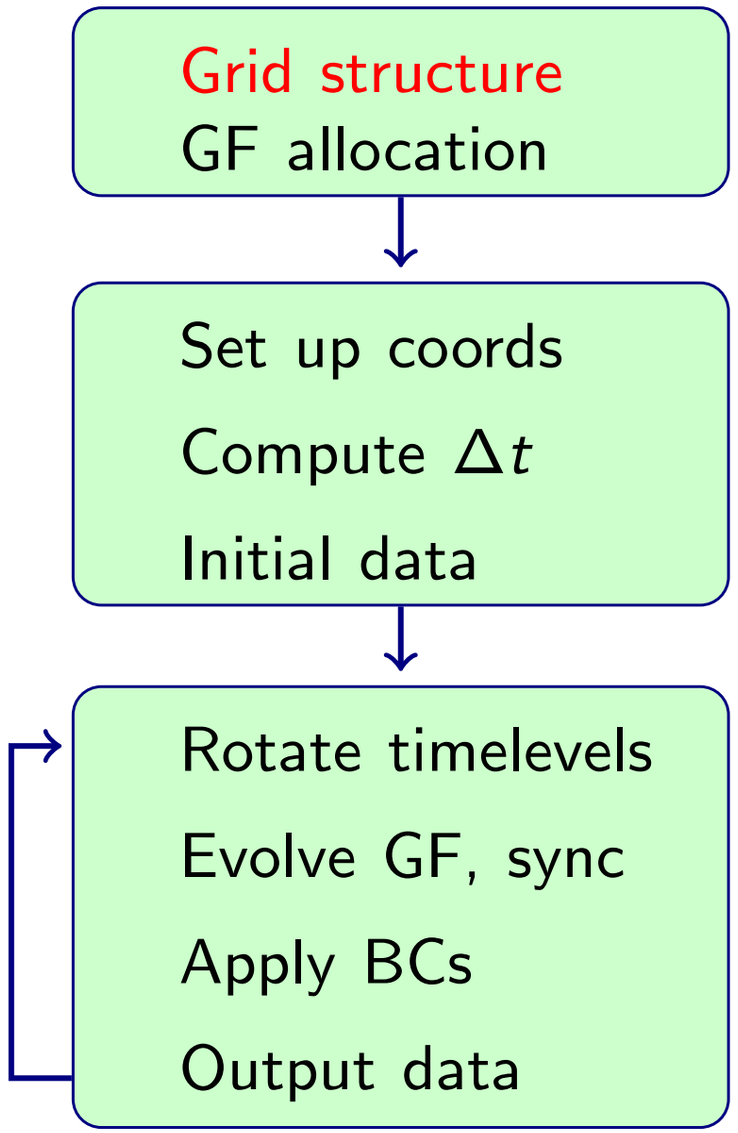
- Parameters of initial Gaussian pulse: amplitude A, radius R, width σ

schedule.ccl

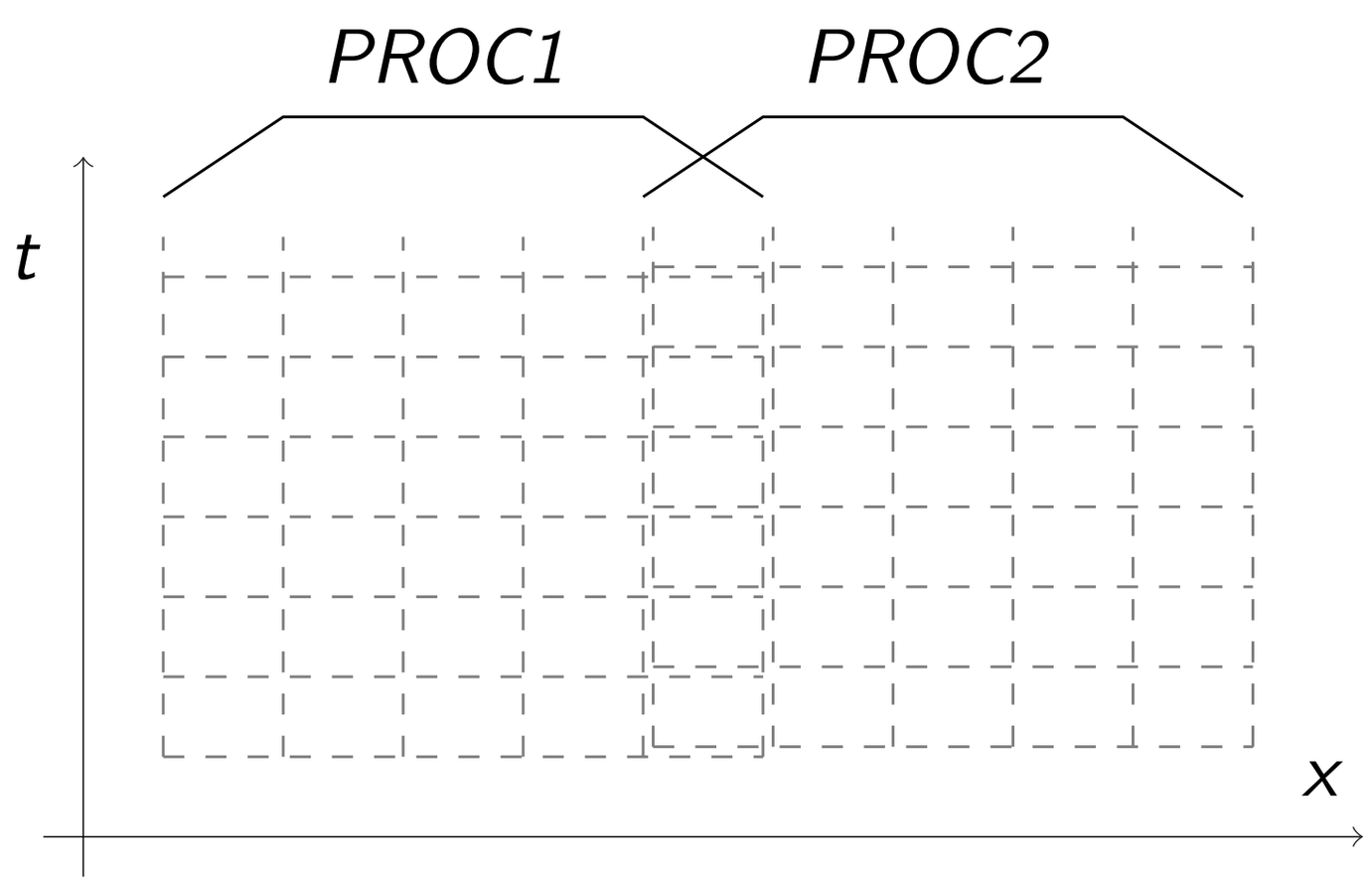
- WaveToy_InitialData
- WaveToy_Evolution
- WaveToy_Boundaries

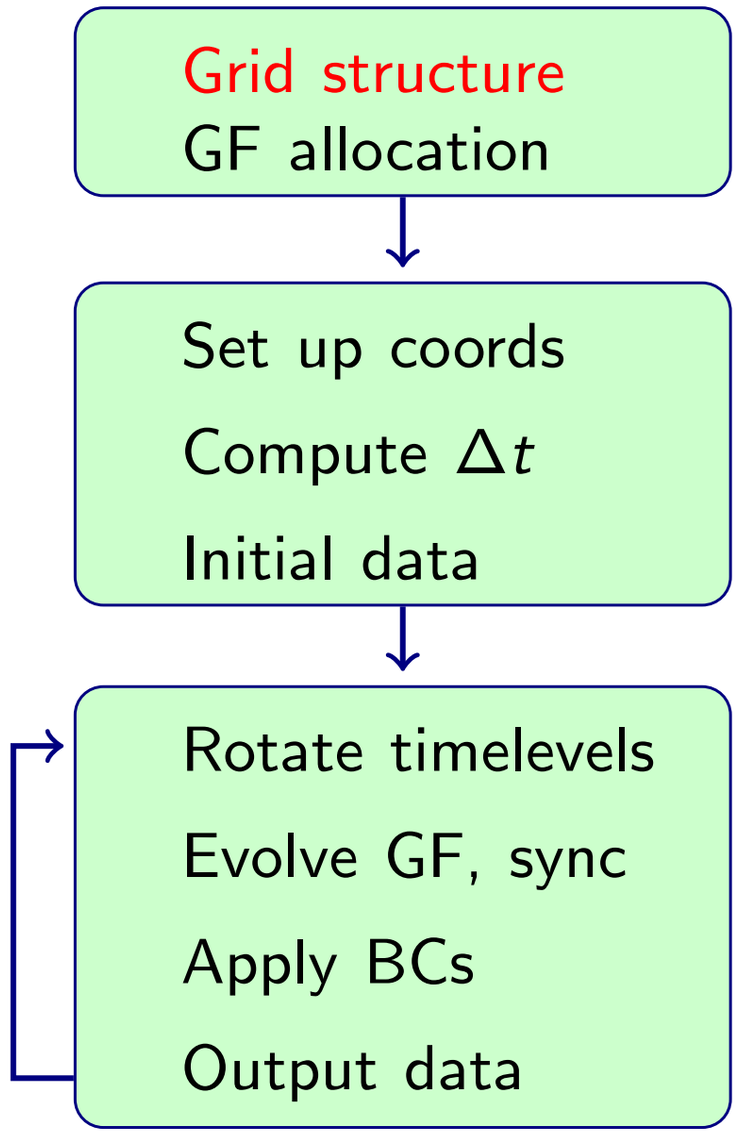
WaveToy Thorn: Algorithm Illustration



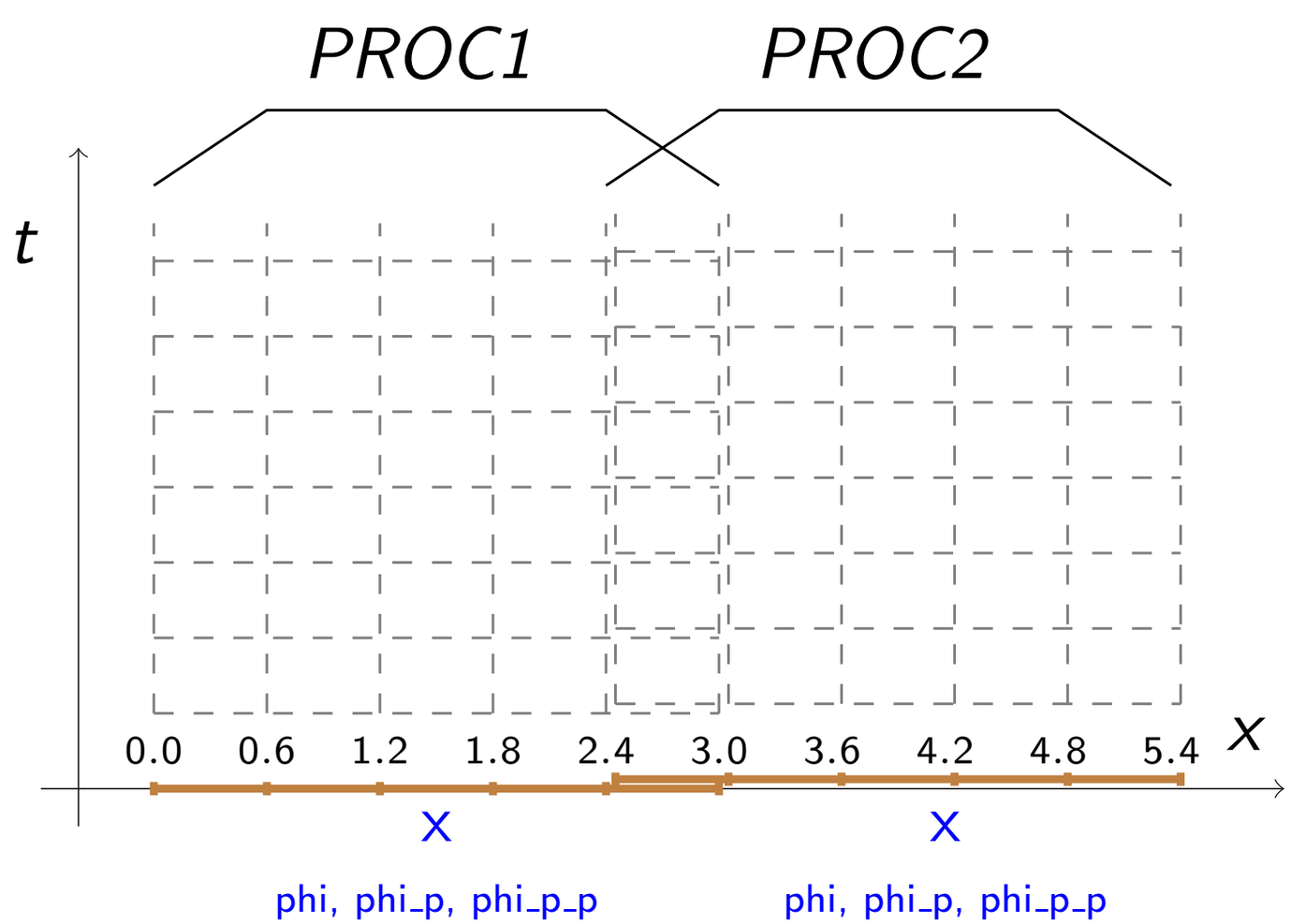


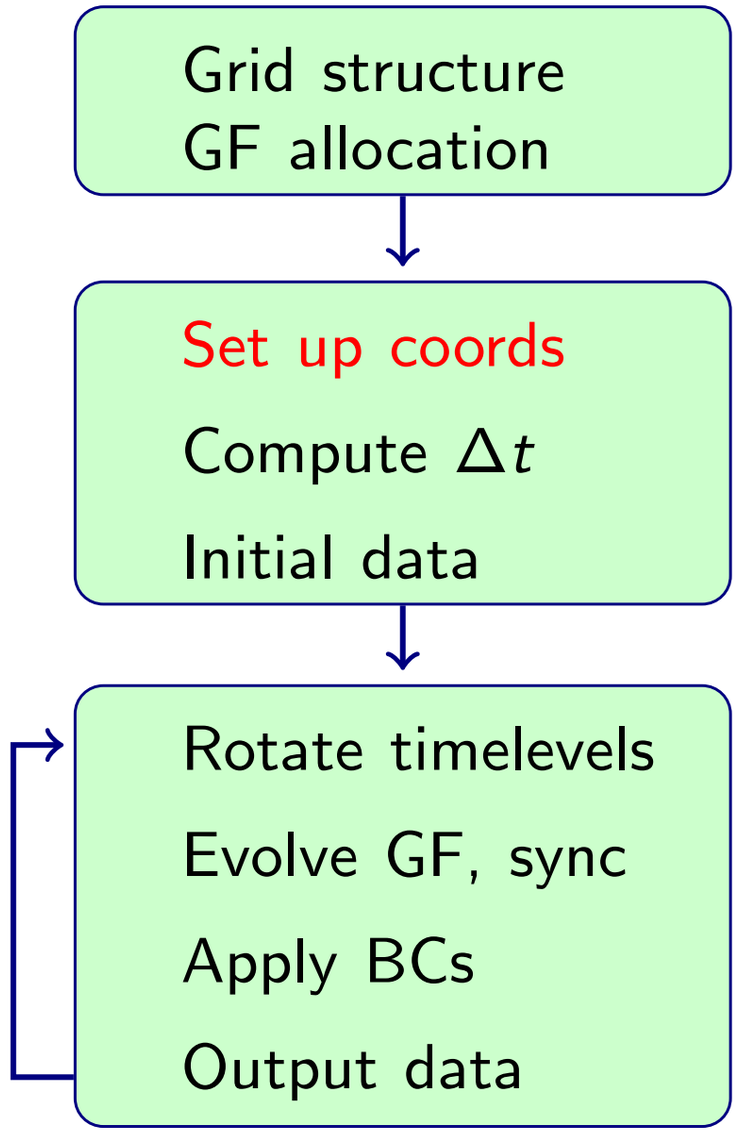
WaveToy Thorn: Algorithm Illustration



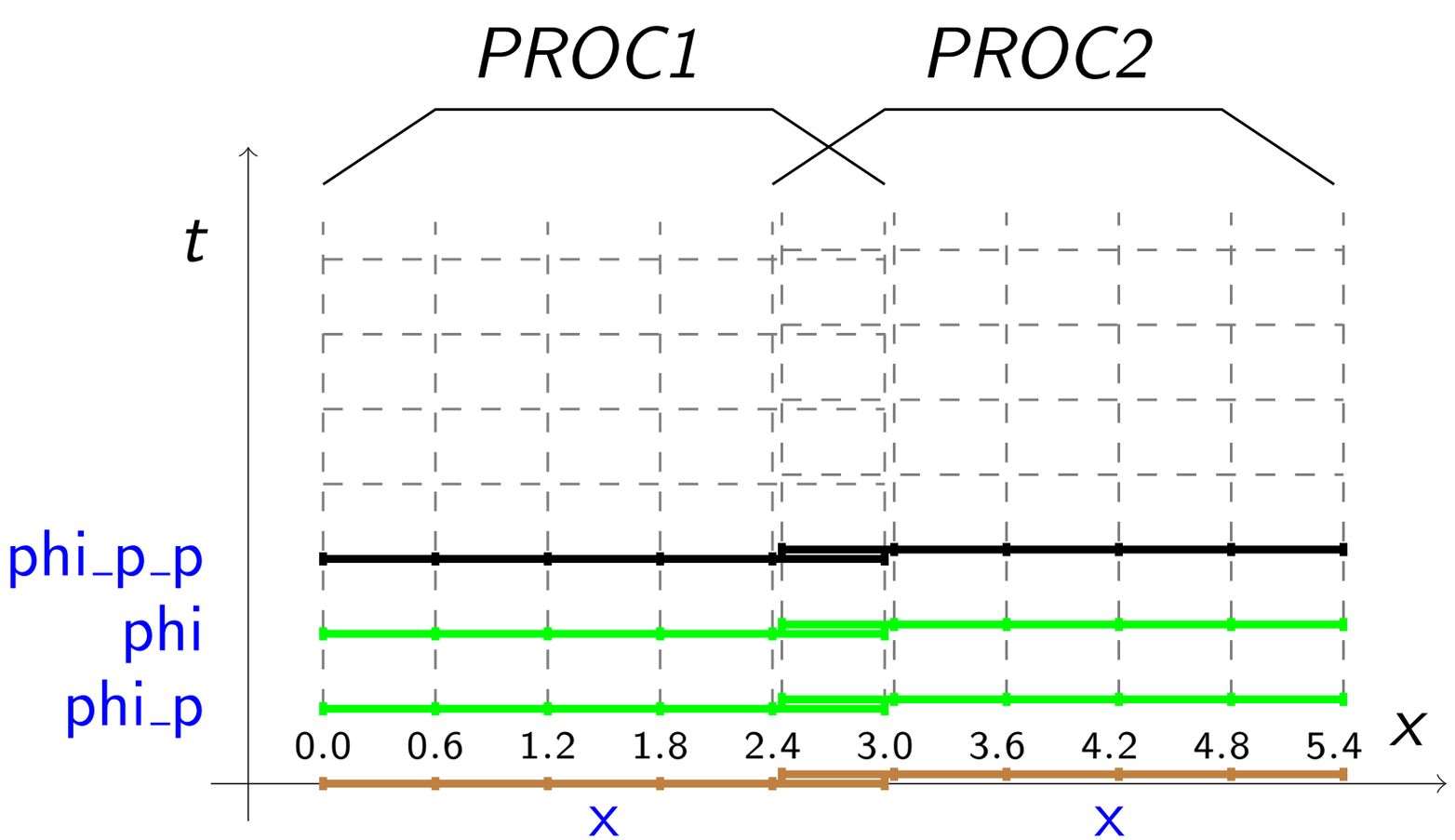


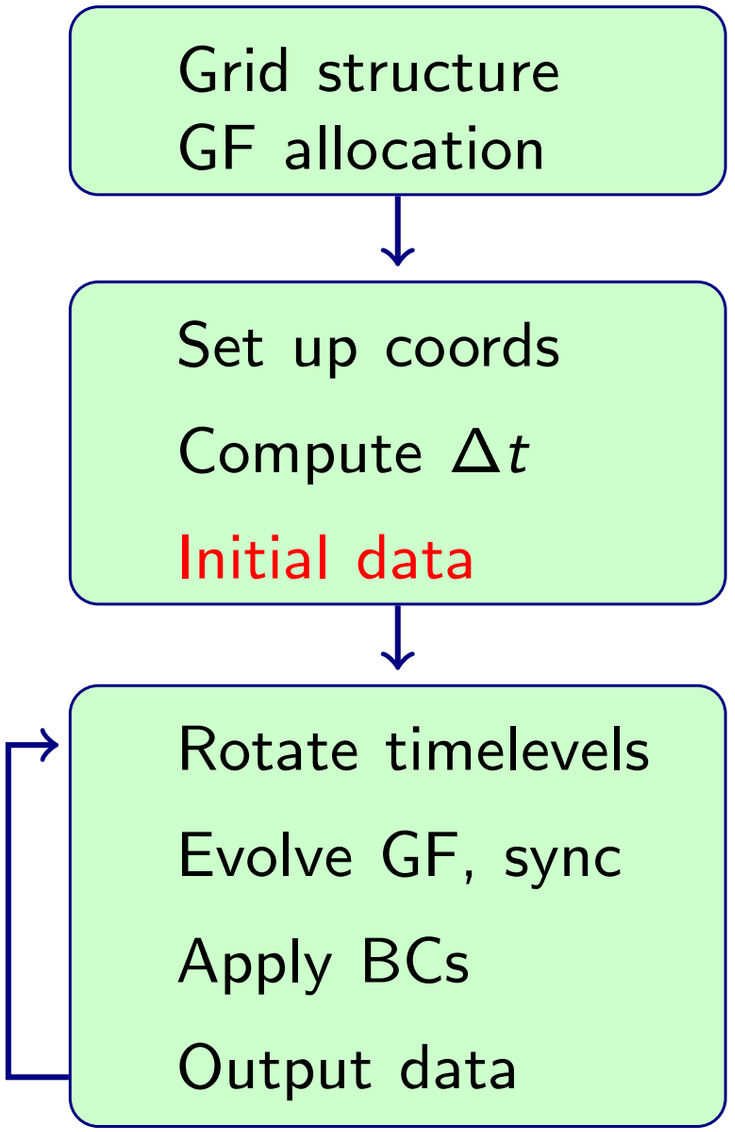
WaveToy Thorn: Algorithm Illustration



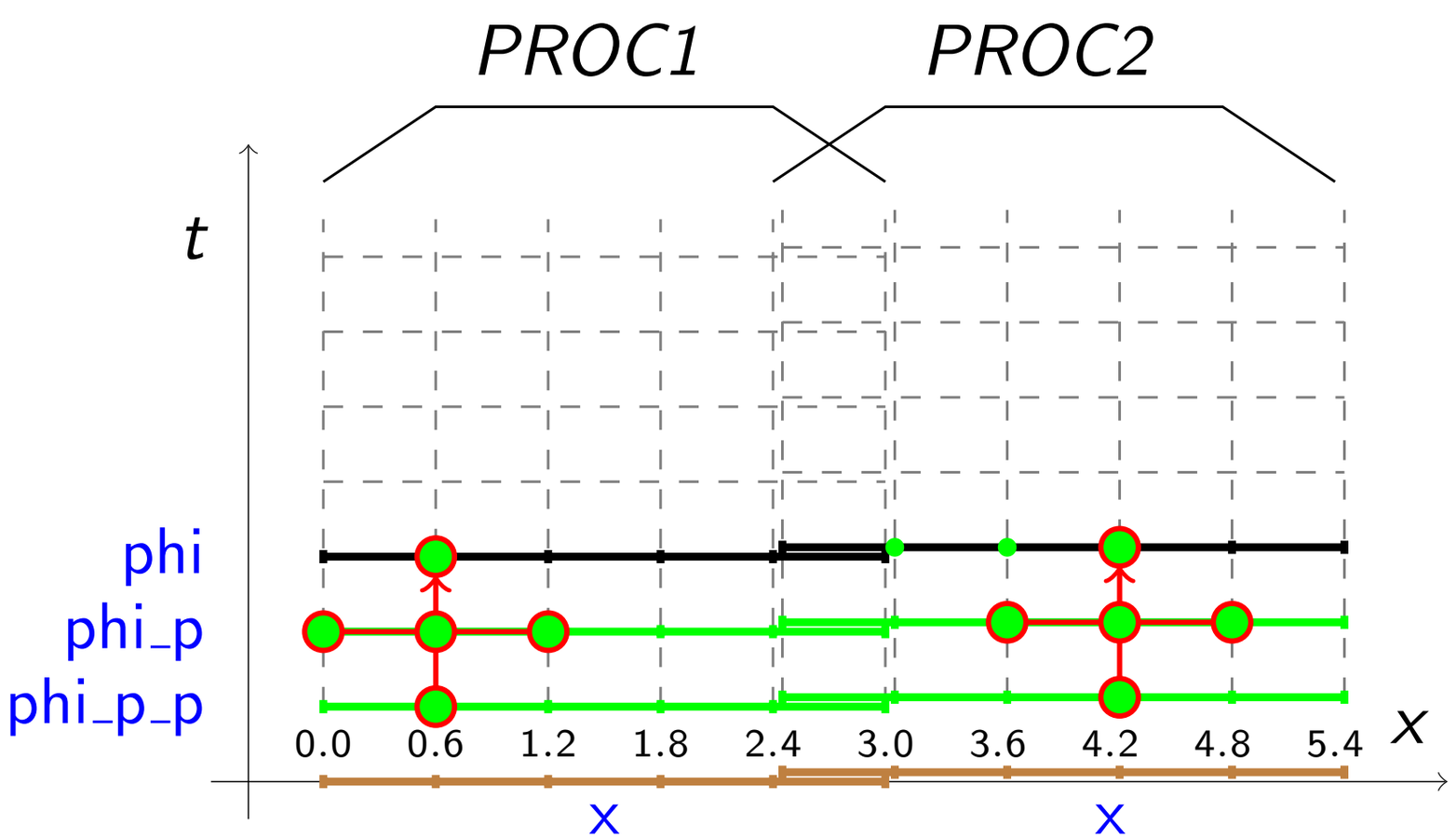


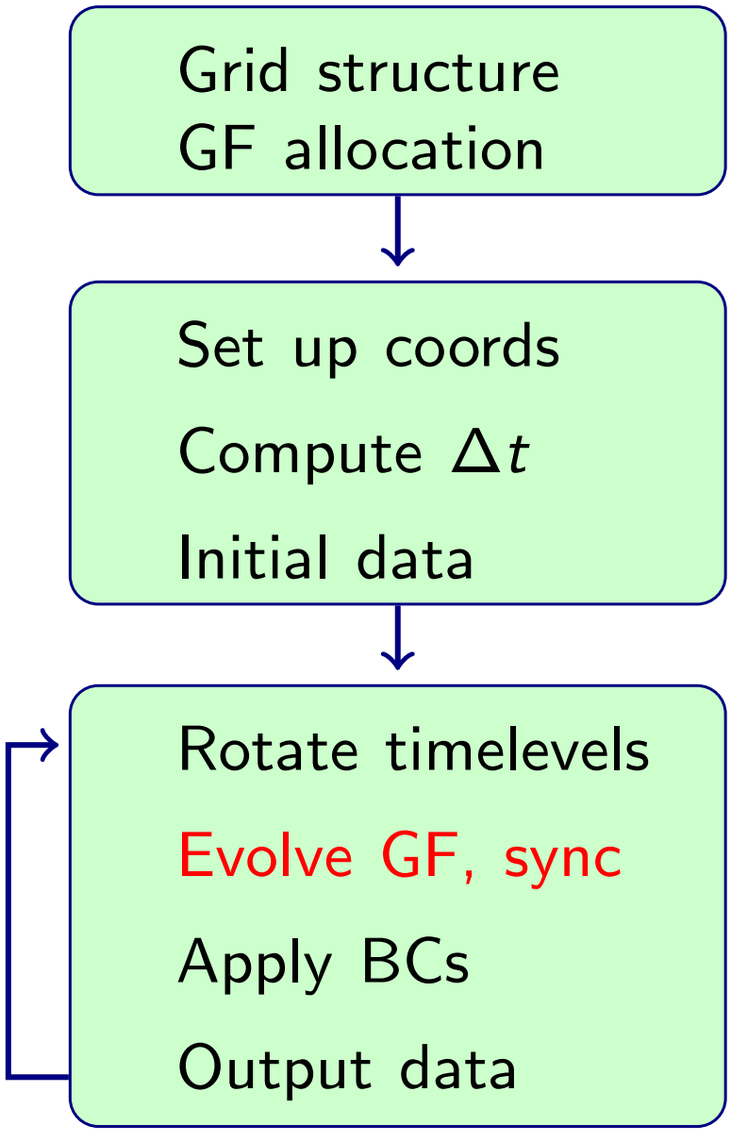
WaveToy Thorn: Algorithm Illustration



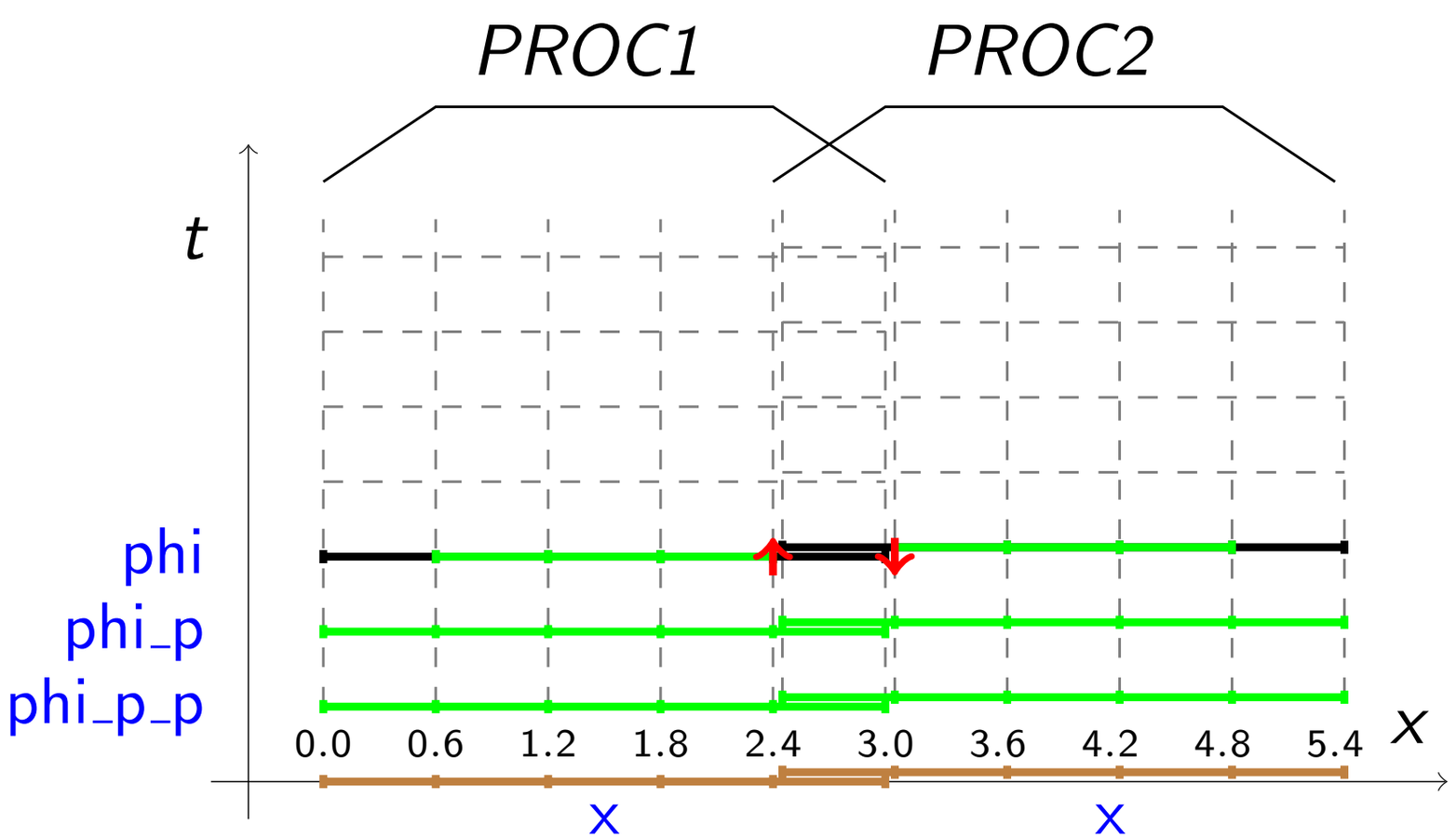


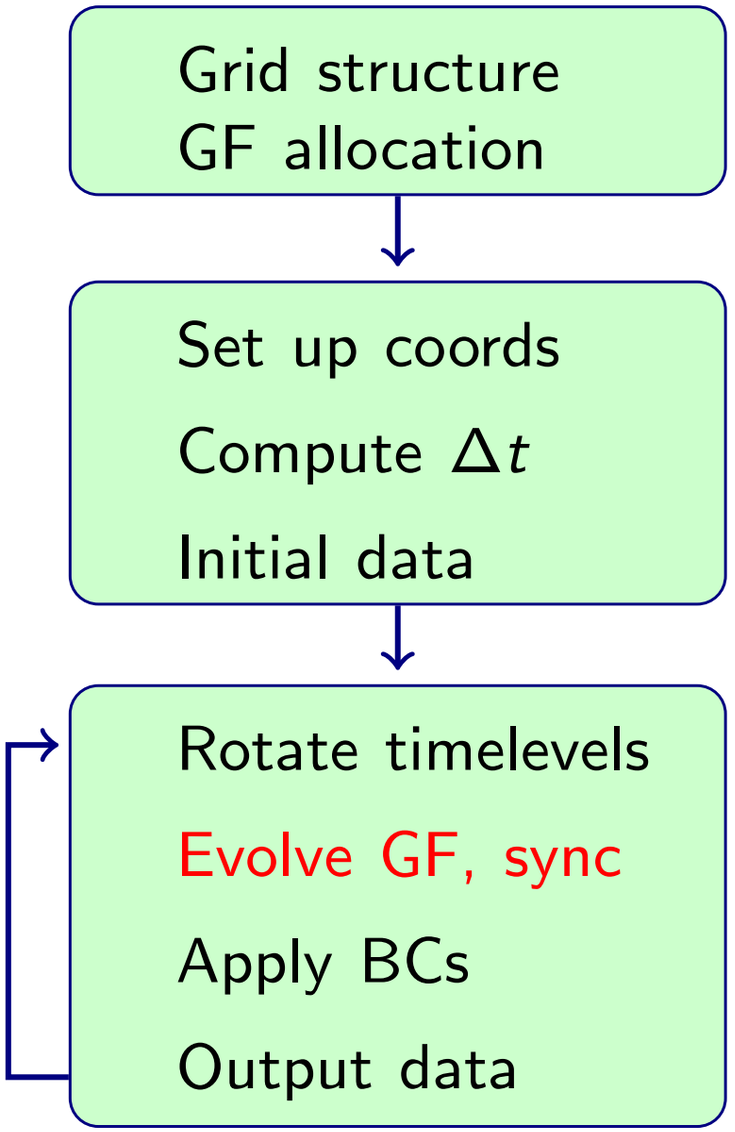
WaveToy Thorn: Algorithm Illustration



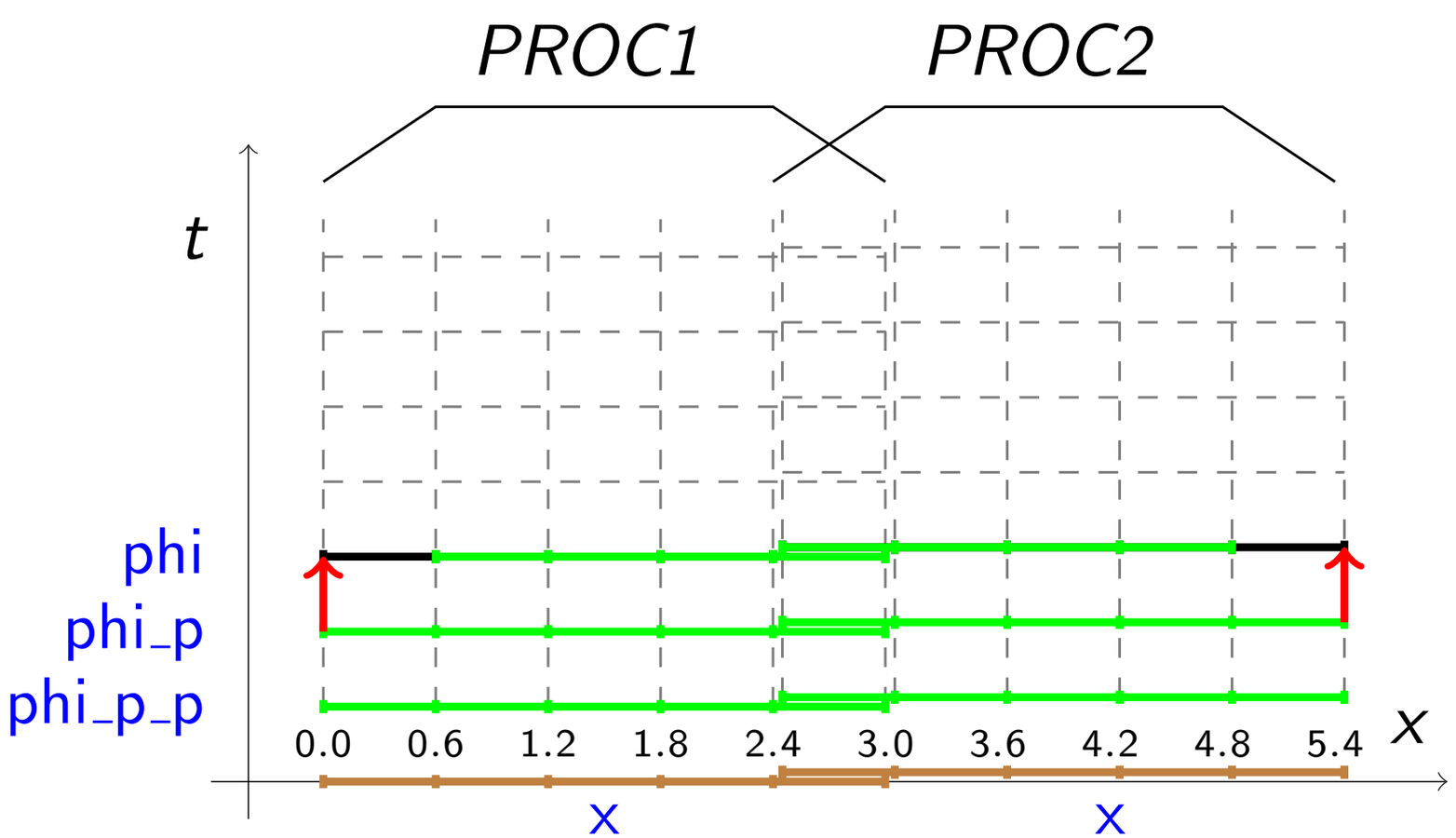


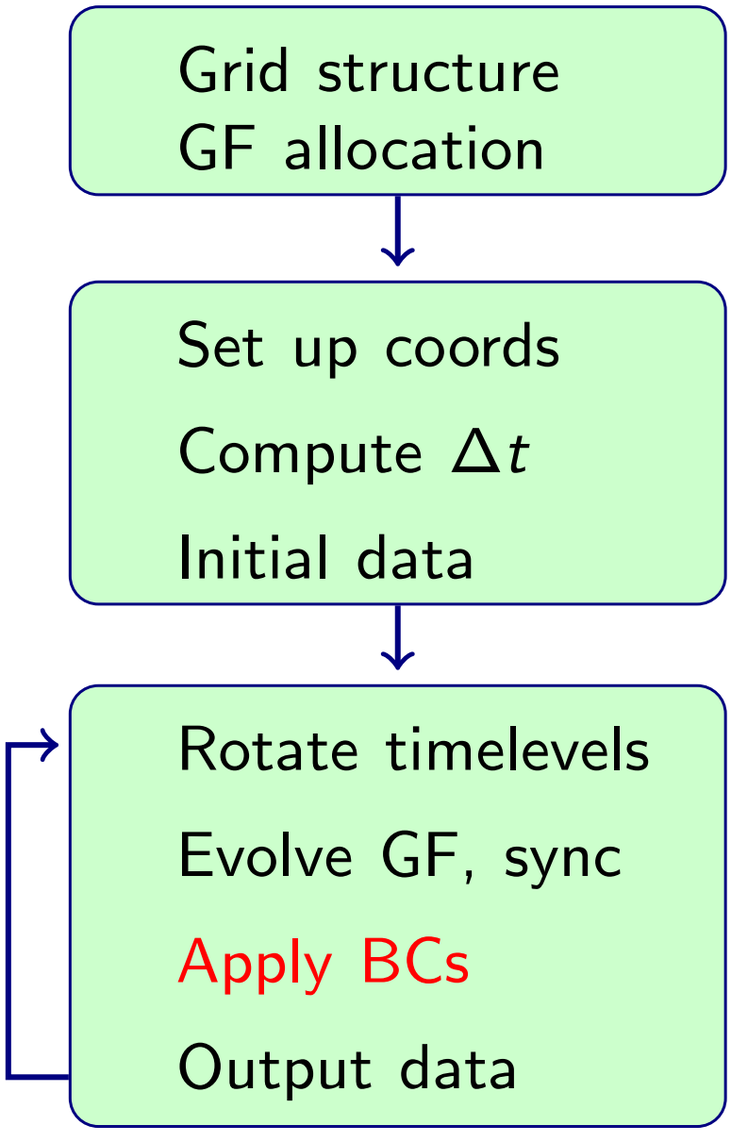
WaveToy Thorn: Algorithm Illustration



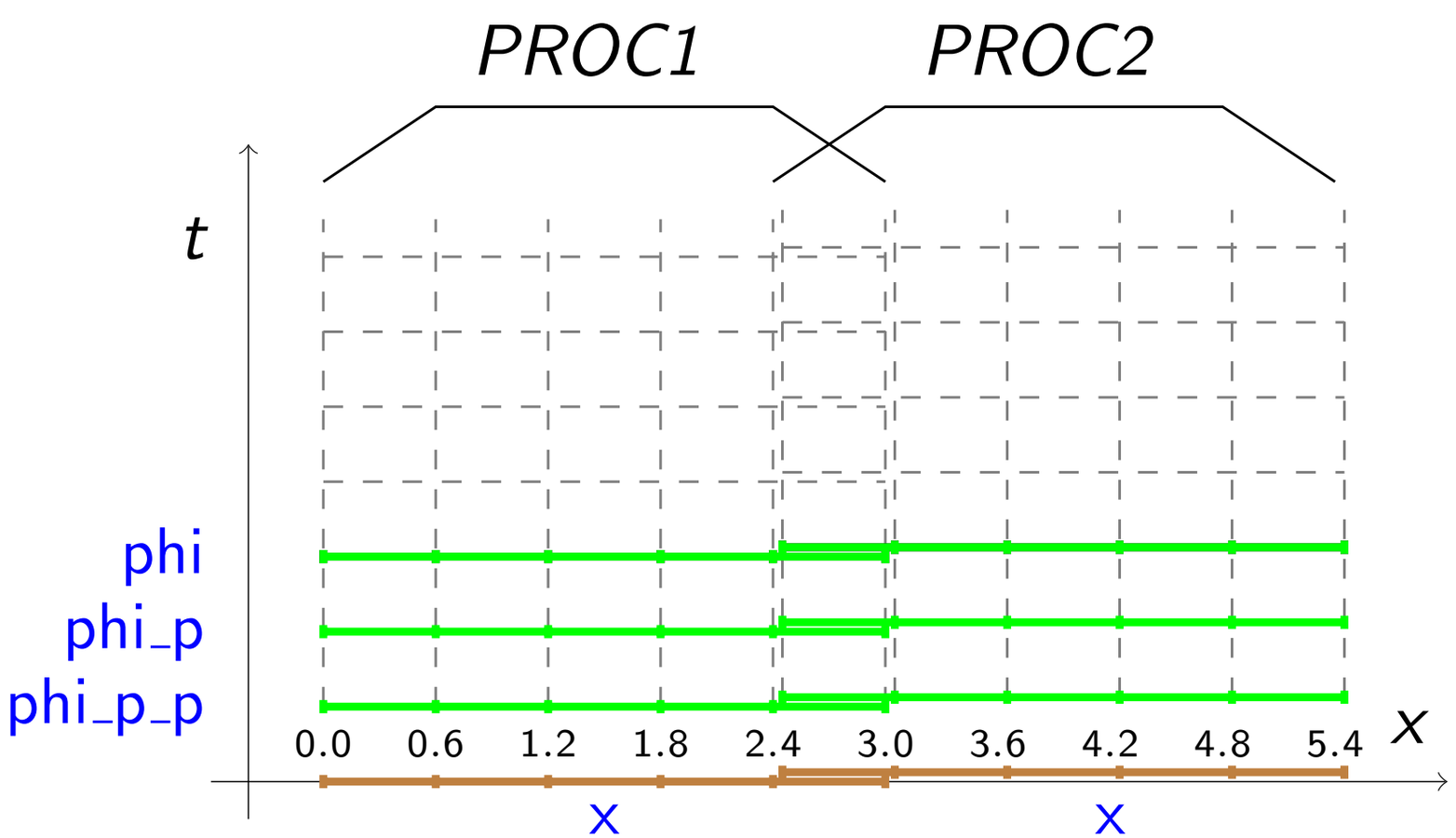


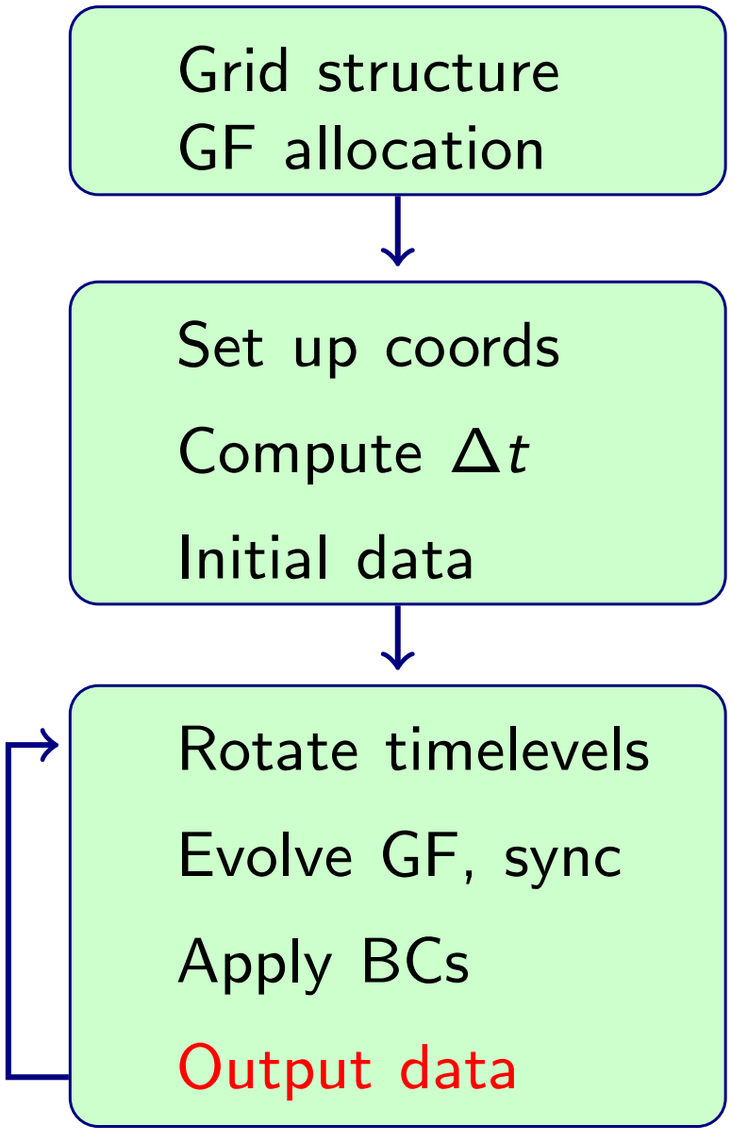
WaveToy Thorn: Algorithm Illustration



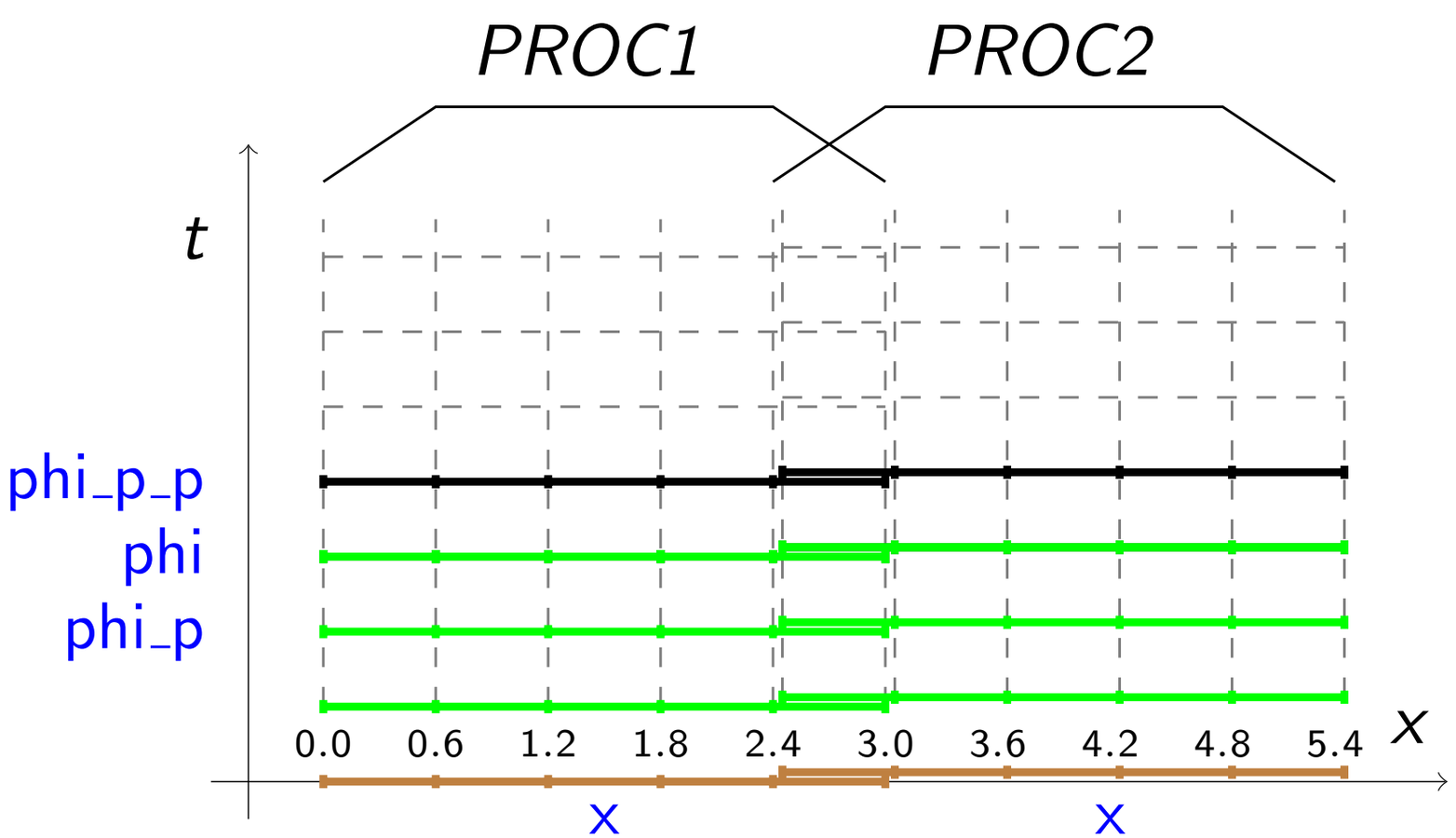


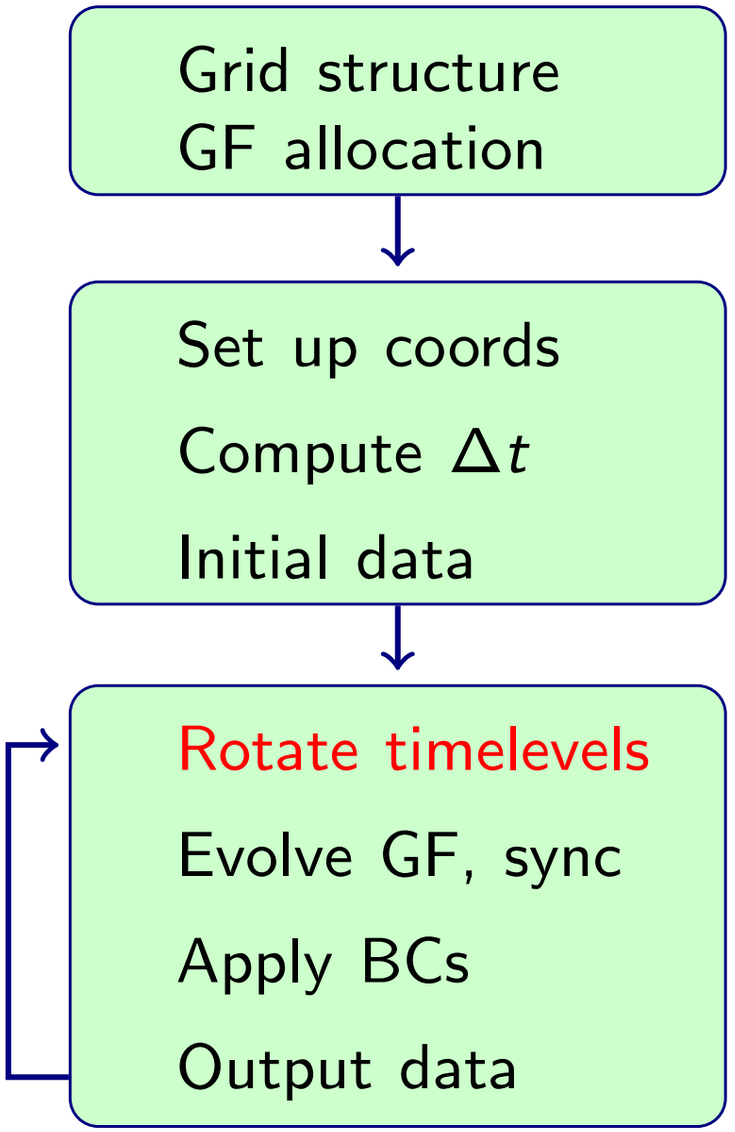
WaveToy Thorn: Algorithm Illustration





WaveToy Thorn: Algorithm Illustration





(Simple) Time Stepping Sequence

1. Evolve (step) interior of the grid, where all spatial derivatives can be calculated with the help of ghost zones
2. Apply outer boundary conditions
3. Synchronise inter-process ghost zones
4. Repeat!

- Note: No process can advance before all other processes have completed their work – *lock stepping*

Load Balancing

- Lock stepping consequences:
  - If one process finishes early, it has to idle (waste time)
  - If one process finishes late, all others have to idle (much worse!)
- Remedies: try to distribute load evenly, distribute load dynamically
- Typical resource allocation problem, very computer sciency, requires complex (parallel) data structures

Ghost Zone Overhead

- Ghost zones require a memory overhead, since the same array element is stored on multiple processes

- In the example above, the overhead was 20%

- In a realistic example (GRB calculation), the overhead can be much larger:

- Assume $30^3 = 27{,}000$ grid points per process (3D)

- With 5 ghost zones (higher order FD), have $(30+2 \cdot 5)^3 = 40^3$ grid points with ghosts

- Thus $40^3 - 30^3 = 37{,}000$ ghost points per process

- Overhead >35%

Parallelisation Summary

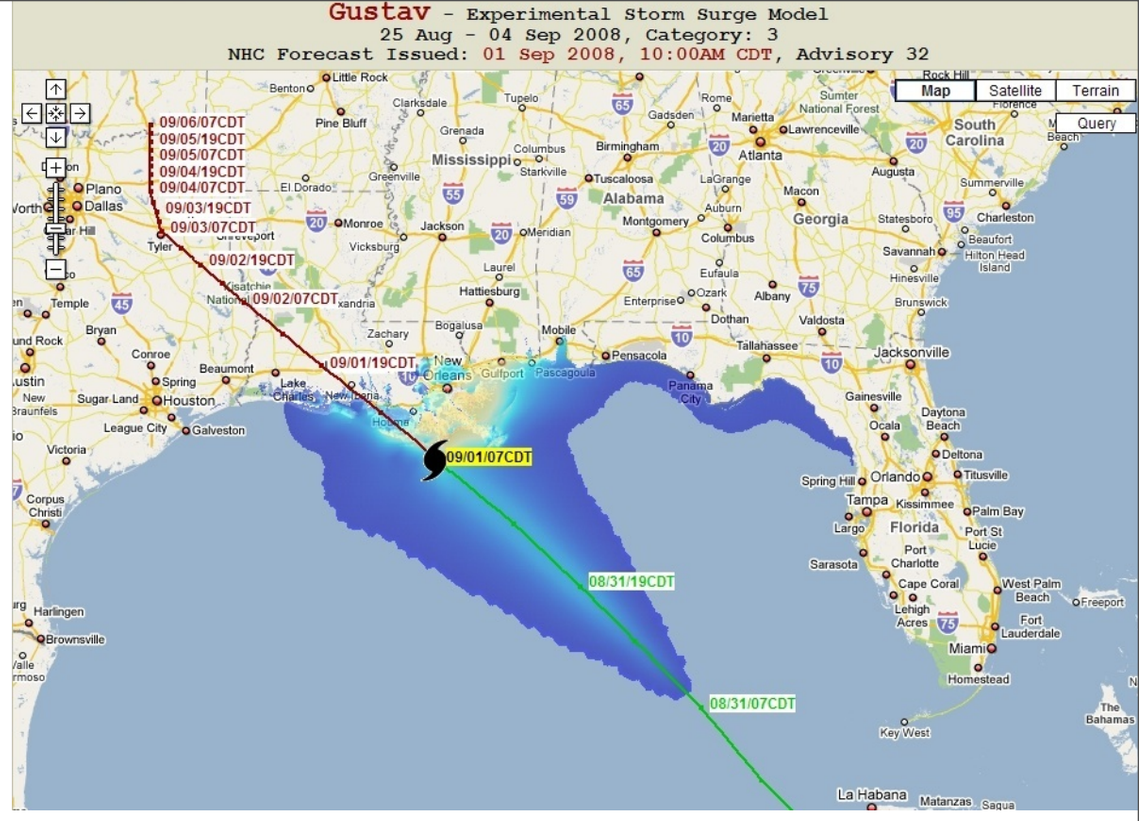
- Need efficient parallel algorithms for current supercomputers, in every corner of the program (Amdahl!)
- MPI is first choice for implementing parallel algorithms
- Domain decomposition (e.g. with ghost zones) distributes simulation data over nodes
- Some important computer science aspects:
  - Designing and implementing efficient distributed data types
  - Load balancing and scheduling to ensure processes don't idle

Part 2: Component Model

Simulation Code Requirements

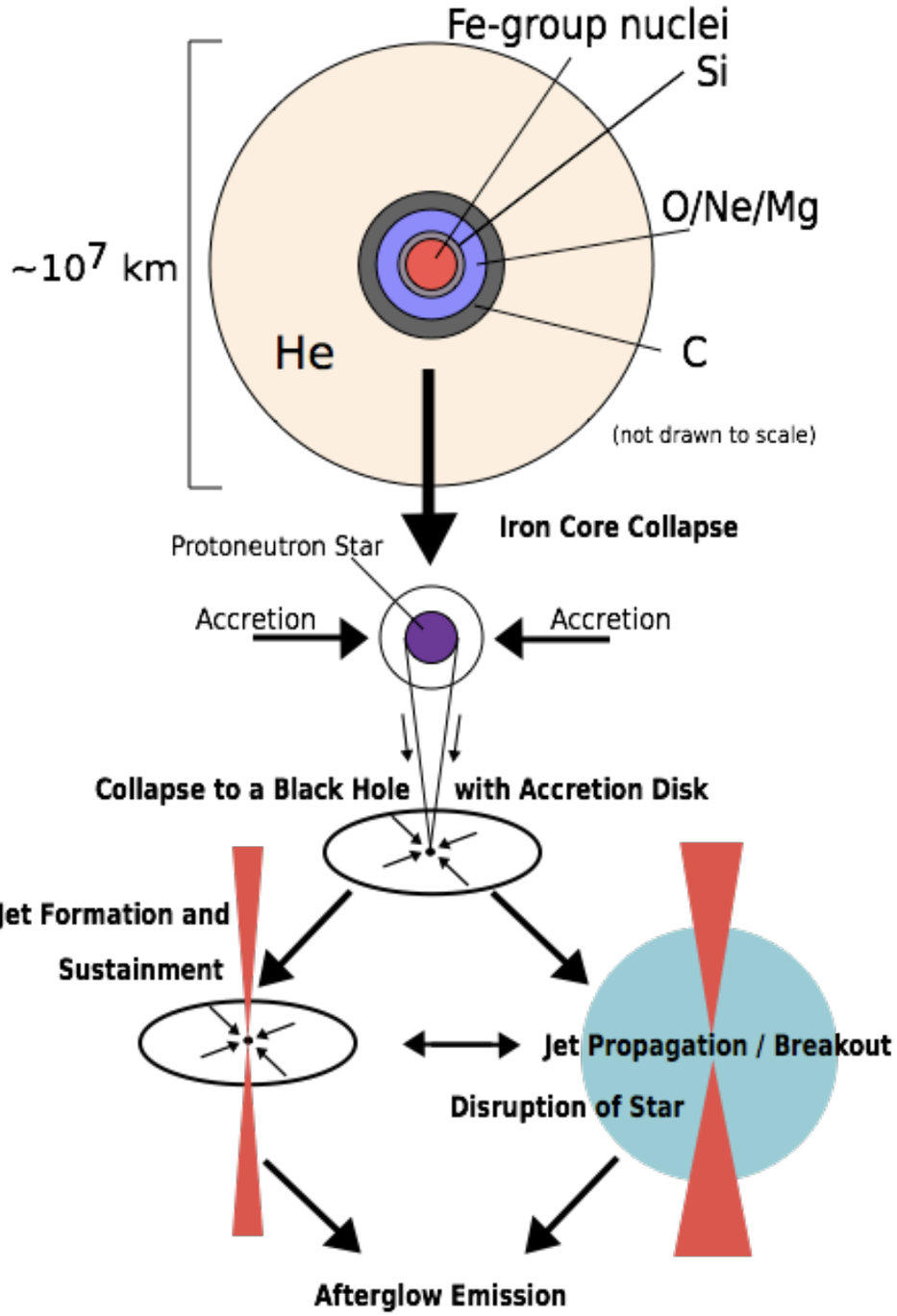
- Reliability, so that one trusts its results
- Extensibility, so that researchers can add and experiment with new ideas
- Usability, so that graduate students don't waste too much time
- Performance, because supercomputing time is expensive

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

No Homework