CSC 7700: Scientific Computing

Module D: Simulations and Application Frameworks

Lecture 2: Simulating Complex Systems

Dr Peter Diener

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

November 15, 2013

Goals

Goals

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

Summary

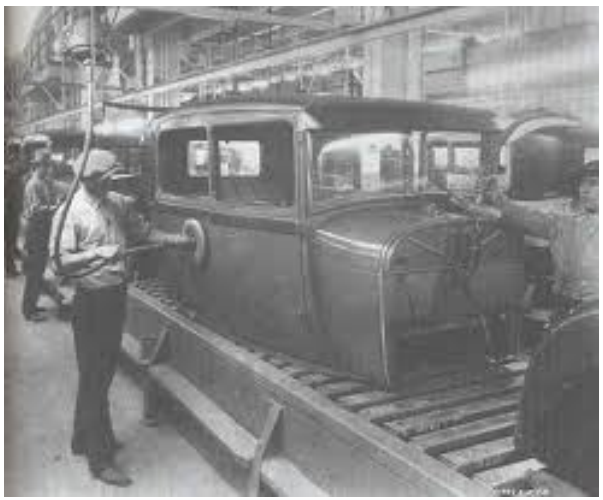
Summary

- 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.

Parallel Computing

HPC History

- Before MPI: *Vector architectures*, e.g. Cray Y-MP (until $\sim$ 1992).
- In the Y-MP, the vector length was 64 words (1 word is 64 bits) and it had 8 vector registers.
- Much more efficient than scalar processors (compare conveyor belt vs. hand assembly).
- Disadvantages: too inflexible for dynamic data structures, too expensive due to custom designed (low volume) hardware and memory system.



HPC History

- After vector machines *cluster architectures* became prevalent (e.g. Cray T3D, 1993 and CM-5, 1991).
- Basic idea: have many simple nodes connected by high-speed network.
- Nodes need to communicate (*exchange messages*) during computation.
- Also called *Beowulf architecture*, especially if only cheap commodity components are used.
- The need for communication lead to the development of MPI.

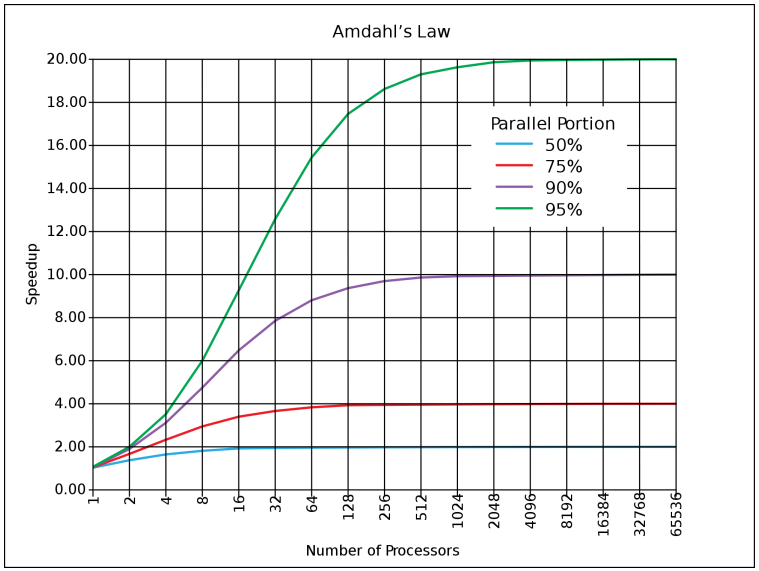
MPI: Message Passing Interface

- MPI is an Application Programming Interface (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.
- Note the emphasis on *possible* rather than *easy*.
- `www.mpi-forum.org`

Amdahl's Law

- P: Fraction of code that can be parallelised.
- N: Number of processors used.
- Amdahl's law:

$$S = \frac{1}{(1-P) + P/N}$$



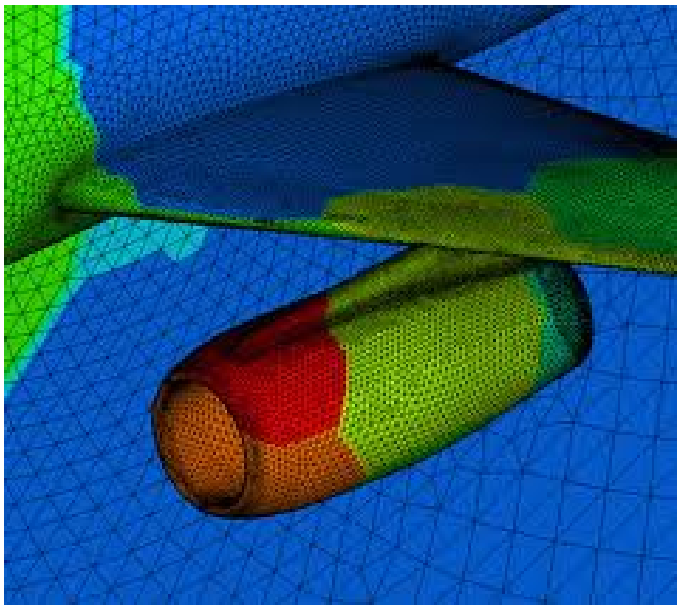
- When running on N processes, not necessarily N times as fast.
- Overhead and non-parallelisable part of the code determines maximum possible parallel speedup.
- 100,000-fold speedup requires $> 99.999\%$ parallelisation.

MPI Programming

- Efficient MPI parallelisation is complex and tedious.
- Requires re-designing data type layouts and APIs (and then rewriting the program).
- To ensure correctness, need good encapsulation of parallelism and understanding of advanced programming concepts.
- Design and implementation needs to be carefully thought out in order to ensure extensibility and portability.

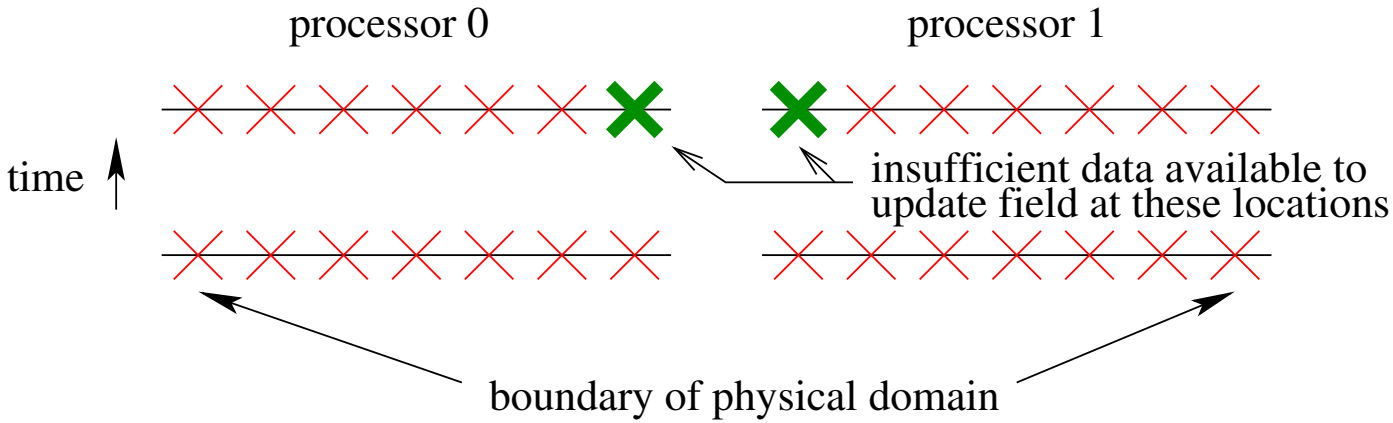
Domain Decomposition

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

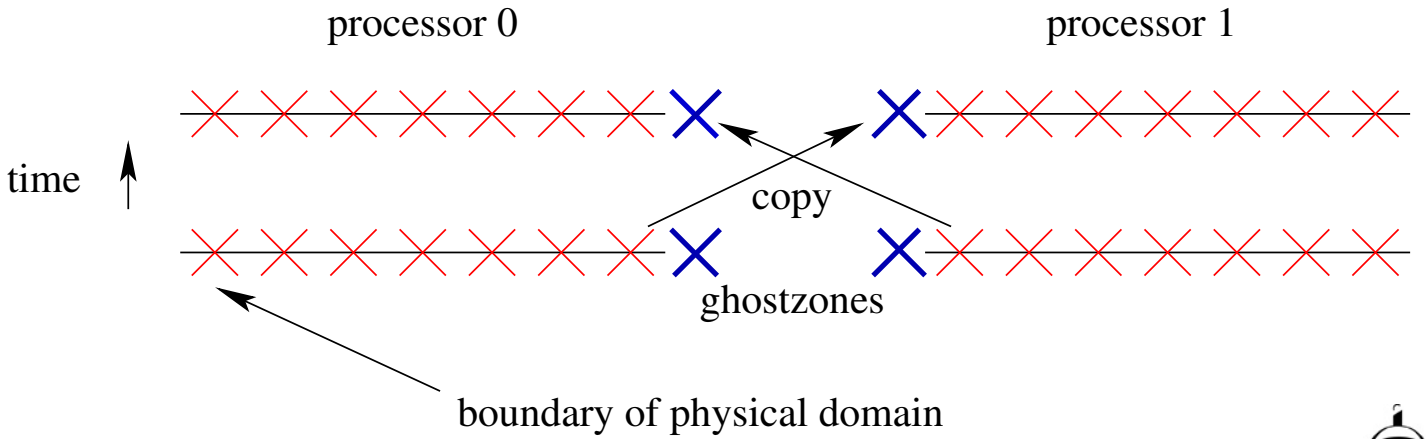


Domain Decomposition

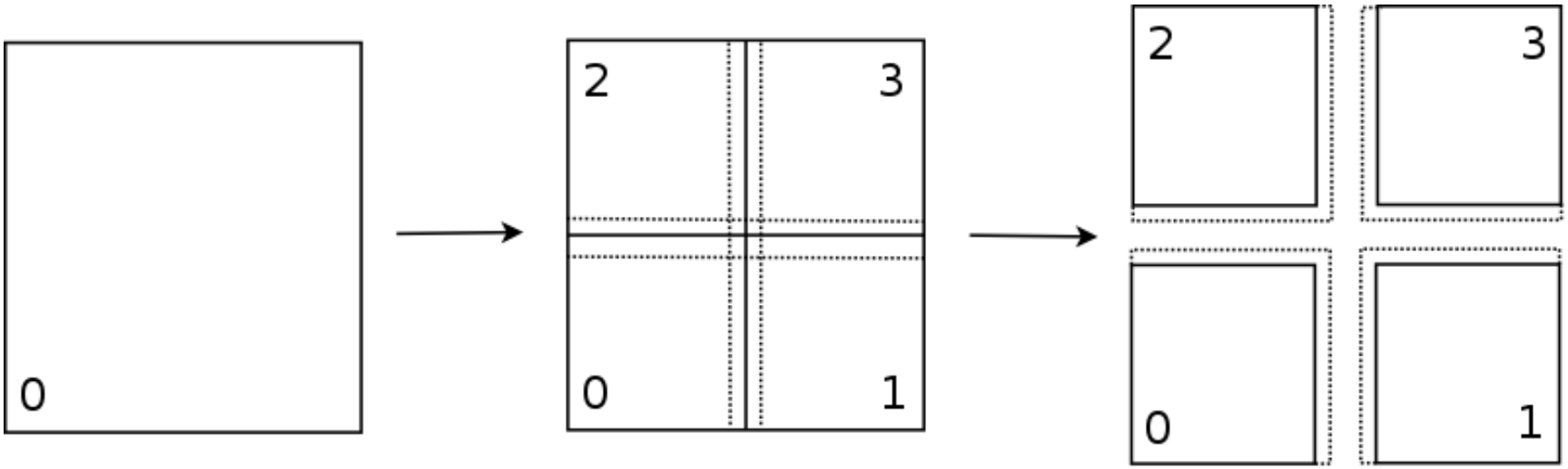
Without Ghostzones:



With Ghostzones:

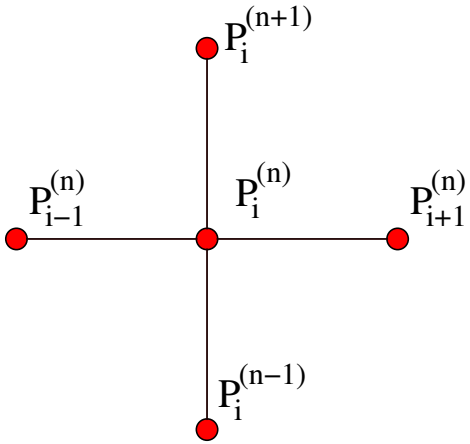


Domain decomposition



The Wave Equation

Approximating the pressure $P(x,t)$ with a grid function $P_i^{(n)}$.



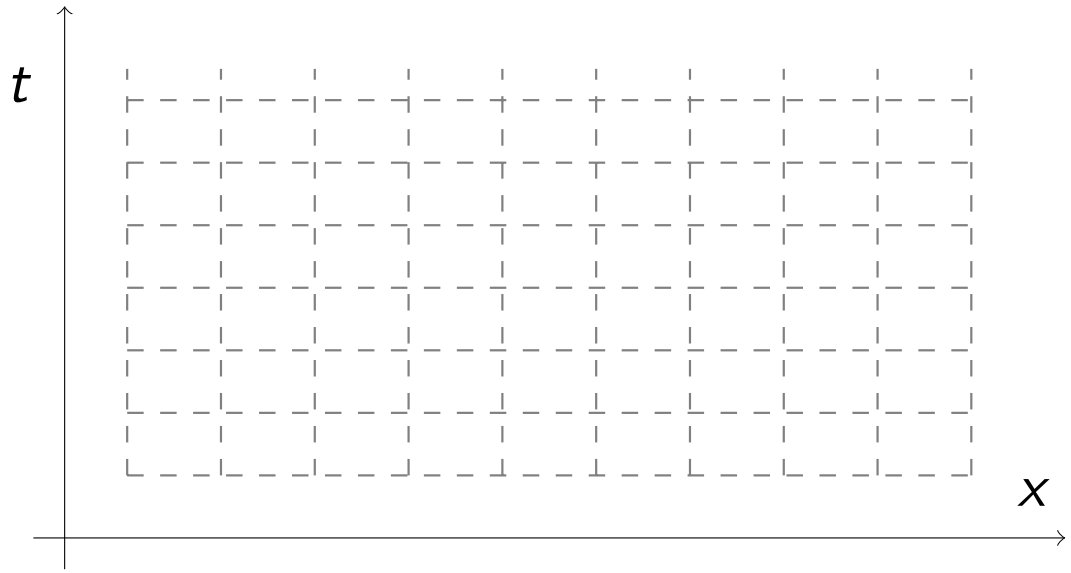
$$\frac{\partial^2 P}{\partial t^2} = v^2 \frac{\partial^2 P}{\partial x^2}$$

$$\Downarrow$$

$$\frac{P_i^{(n+1)} - 2P_i^{(n)} + P_i^{(n-1)}}{\Delta t^2} = v^2 \frac{P_{i+1}^{(n)} - 2P_i^{(n)} + P_{i-1}^{(n)}}{\Delta x^2}$$

The error from this time and space discretisation is $O(h^2)$.

Wave Equation: Algorithm Illustration

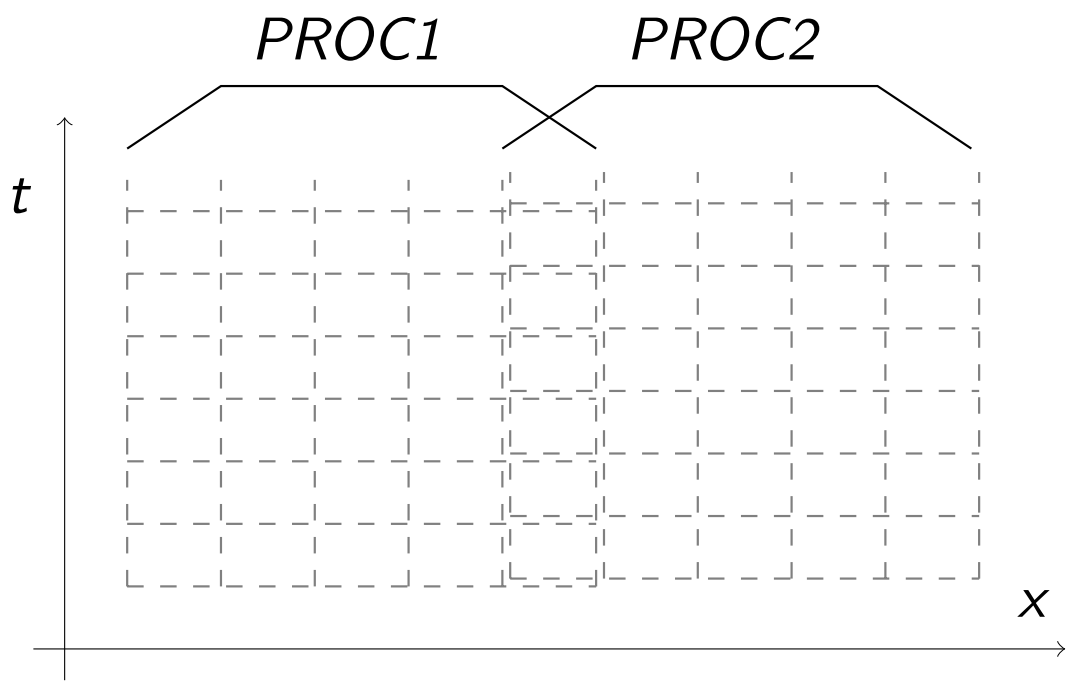


Grid structure
GF allocation

Set up coords
Compute Δt
Initial data

Rotate timelevels
Evolve GF, sync
Apply BCs
Output data

Wave Equation: Algorithm Illustration

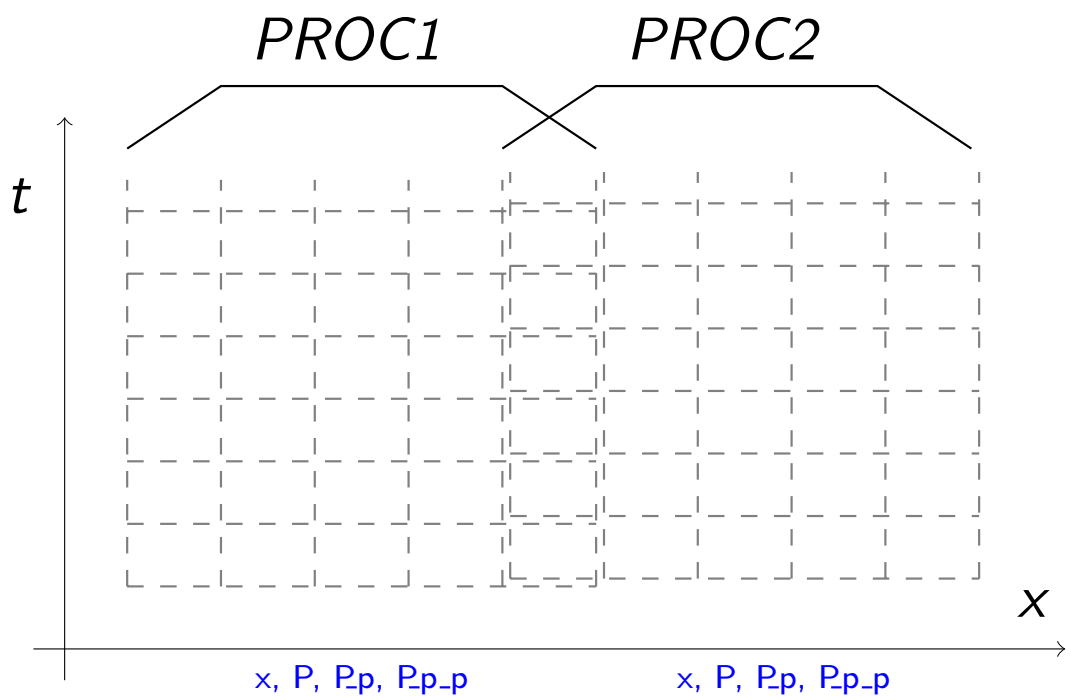


Grid structure
GF allocation

Set up coords
Compute Δt
Initial data

Rotate timelevels
Evolve GF, sync
Apply BCs
Output data

Wave Equation: Algorithm Illustration

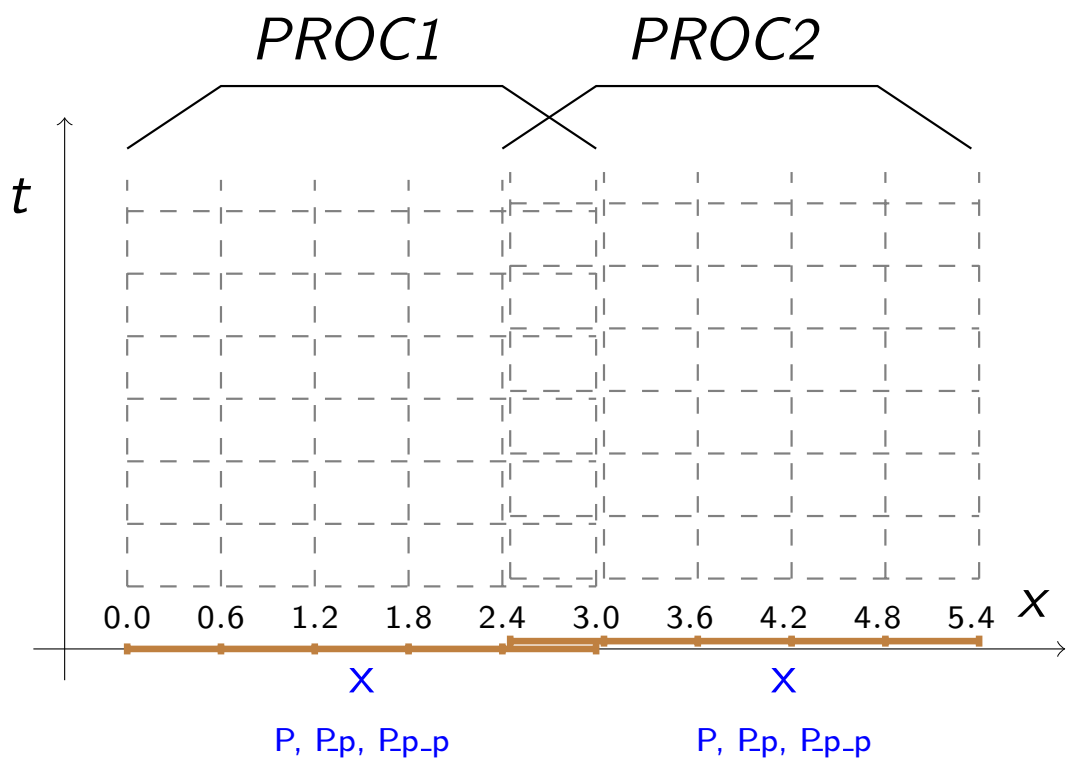


Grid structure
GF allocation

Set up coords
Compute Δt
Initial data

Rotate timelevels
Evolve GF, sync
Apply BCs
Output data

Wave Equation: Algorithm Illustration

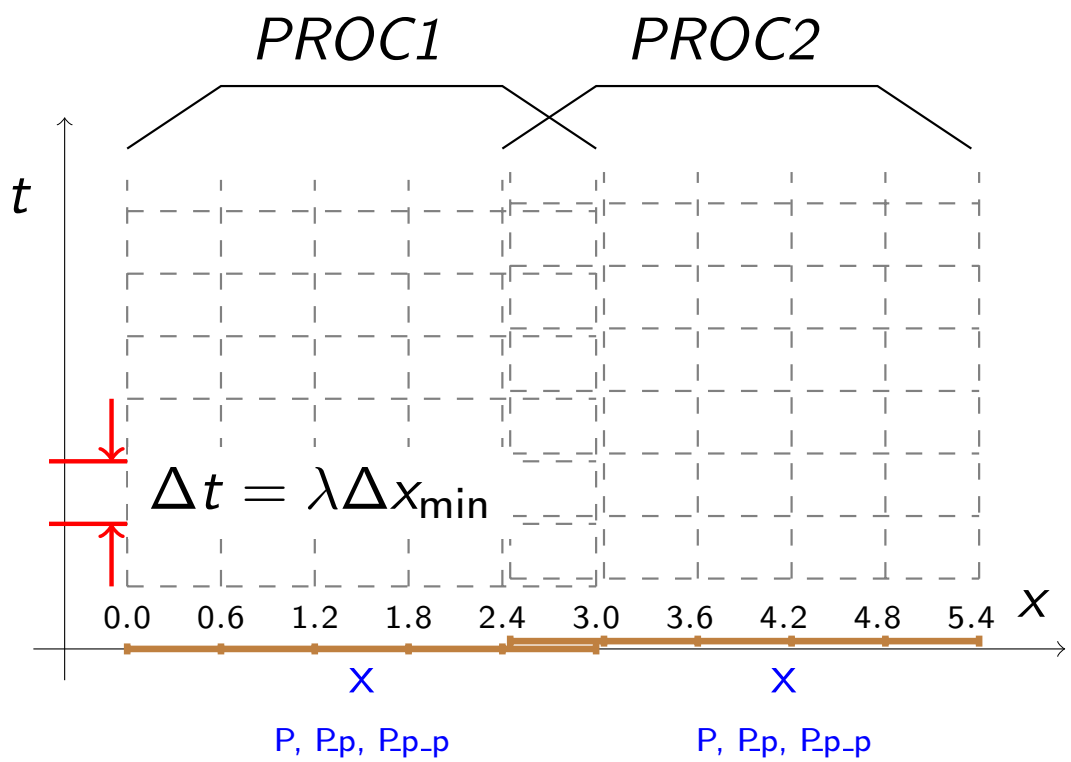


Grid structure
GF allocation

Set up coords
Compute Δt
Initial data

Rotate timelevels
Evolve GF, sync
Apply BCs
Output data

Wave Equation: Algorithm Illustration

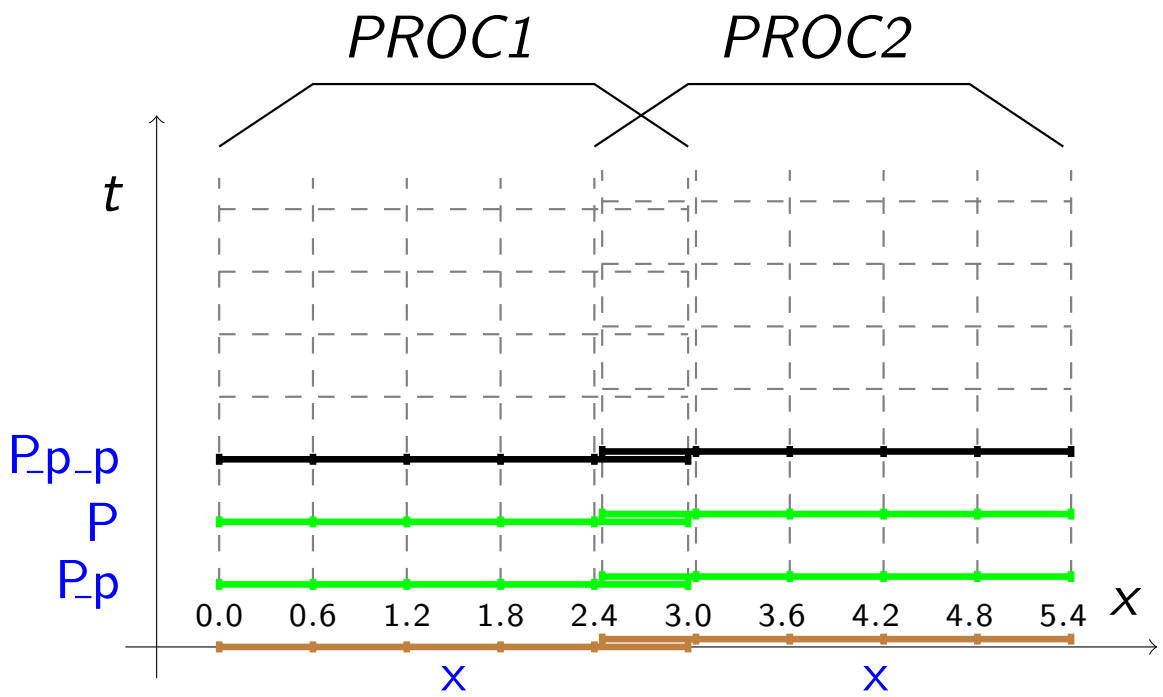


Grid structure
GF allocation

Set up coords
Compute Δt
Initial data

Rotate timelevels
Evolve GF, sync
Apply BCs
Output data

Wave Equation: Algorithm Illustration

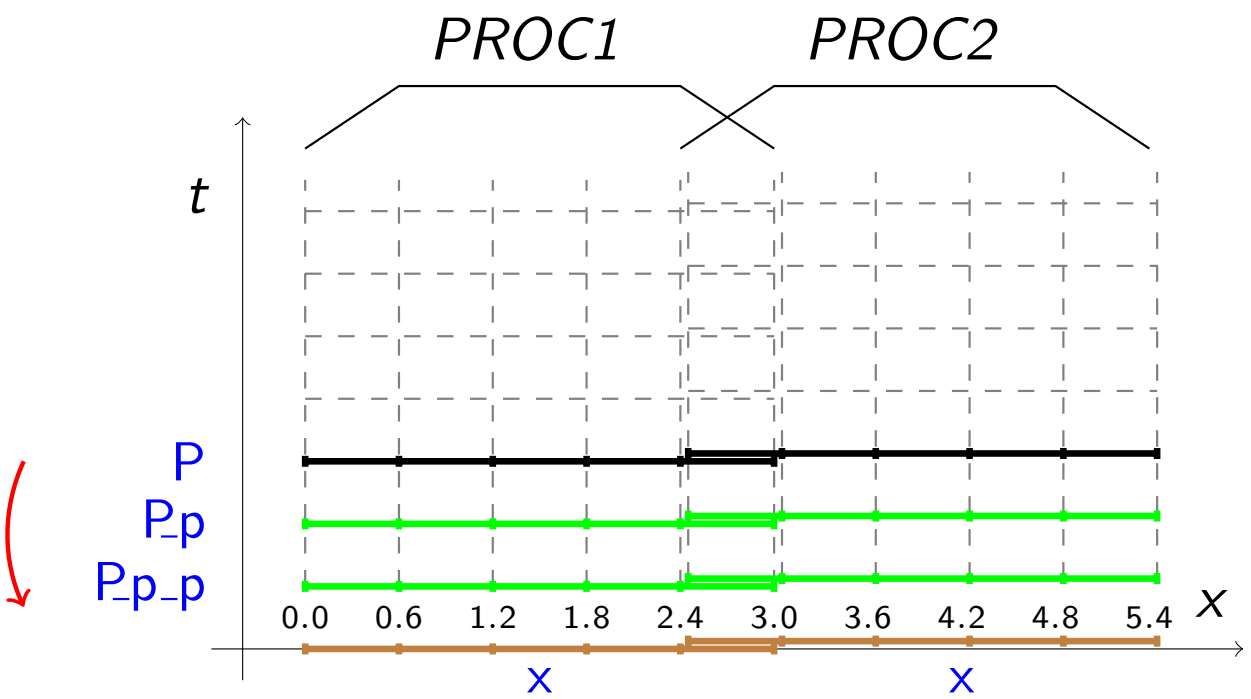


Grid structure
GF allocation

Set up coords
Compute Δt
Initial data

Rotate timelevels
Evolve GF, sync
Apply BCs
Output data

Wave Equation: Algorithm Illustration

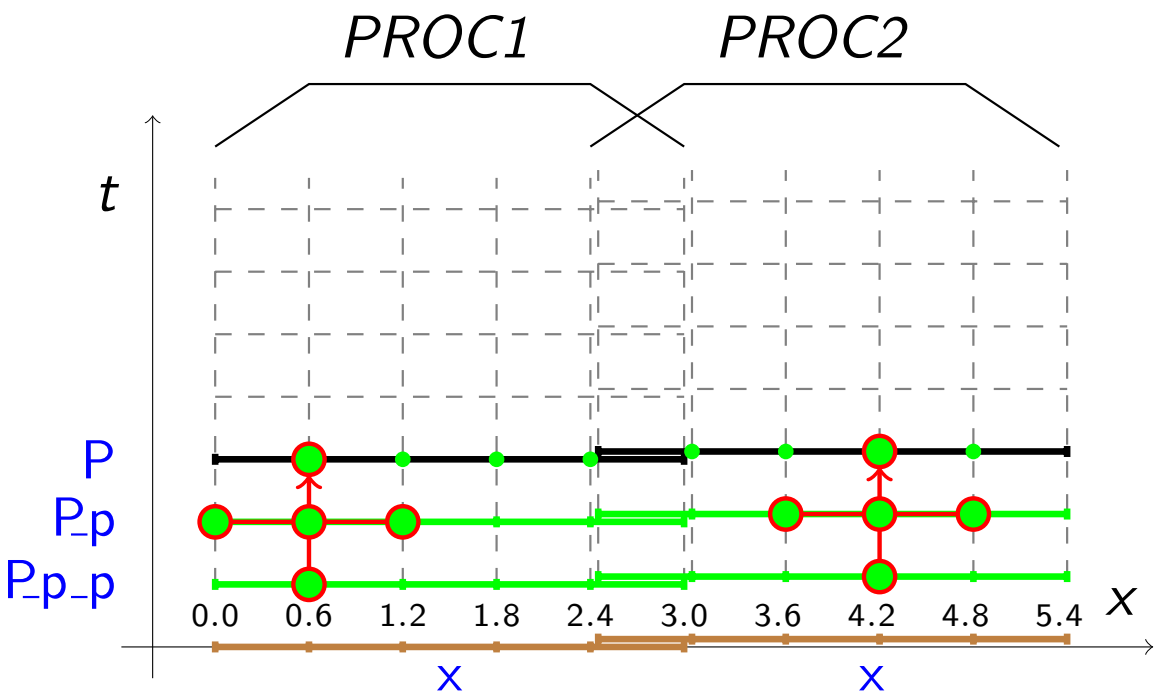


Grid structure
GF allocation

Set up coords
Compute Δt
Initial data

Rotate timelevels
Evolve GF, sync
Apply BCs
Output data

Wave Equation: Algorithm Illustration

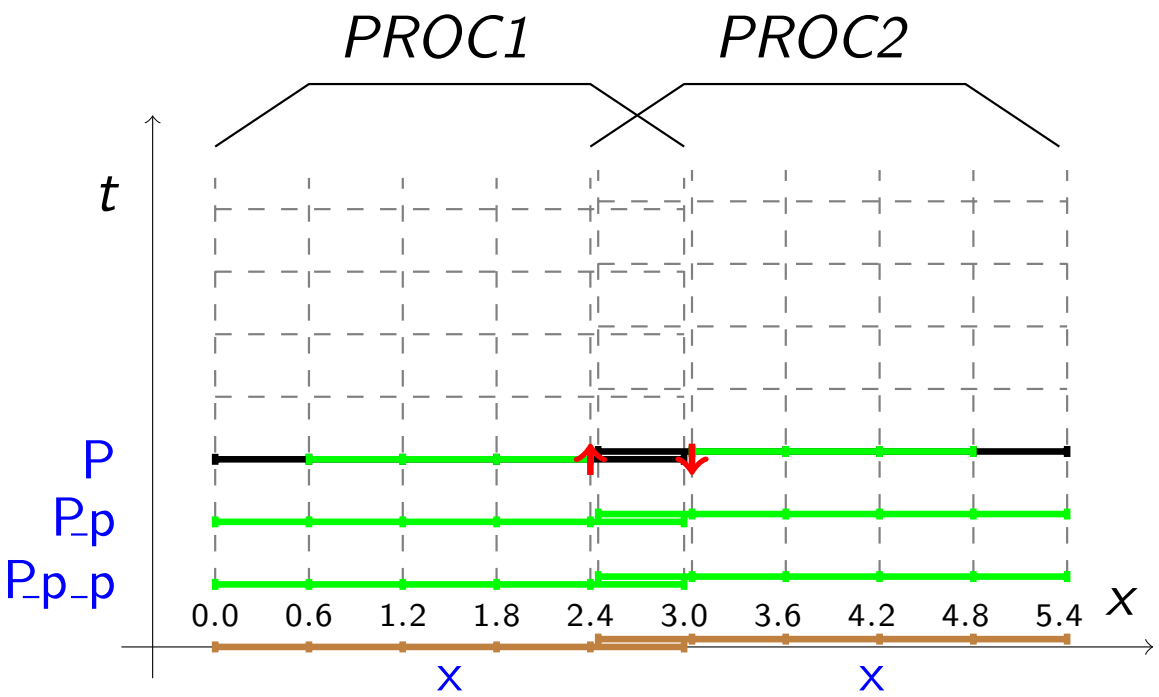


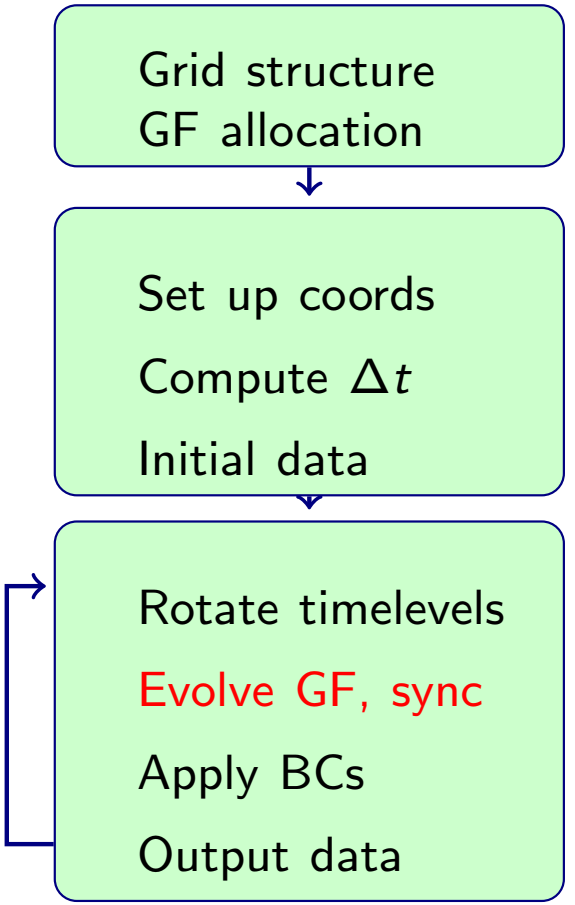
Grid structure
GF allocation

Set up coords
Compute Δt
Initial data

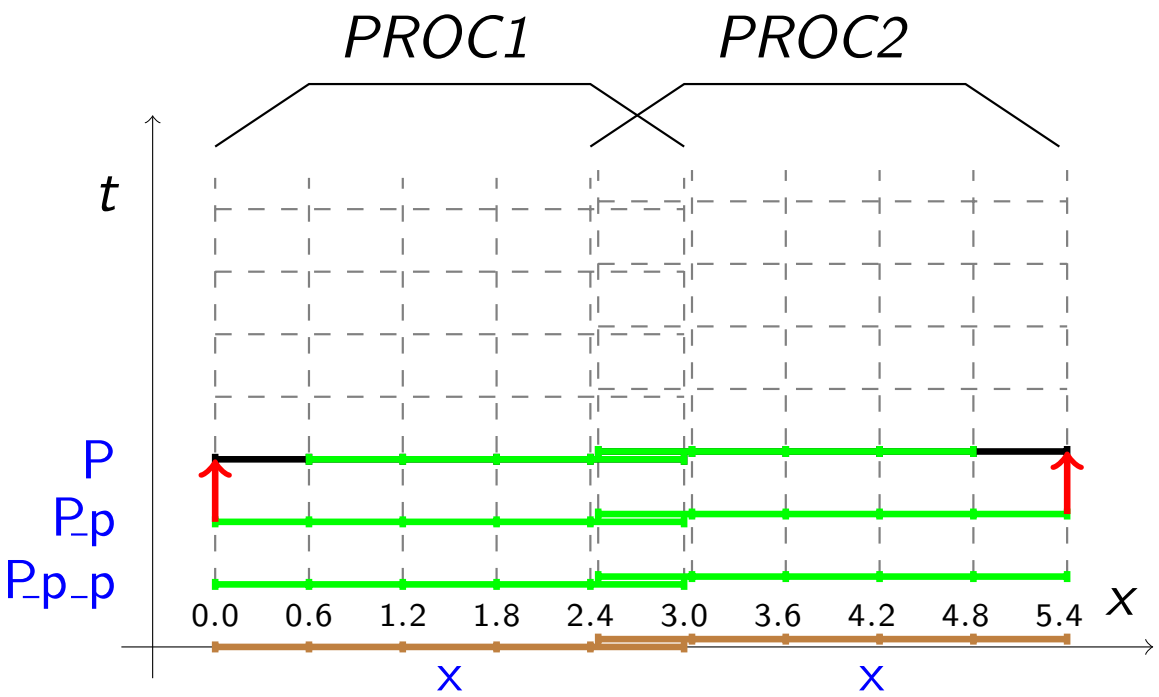
Rotate timelevels
Evolve GF, sync
Apply BCs
Output data

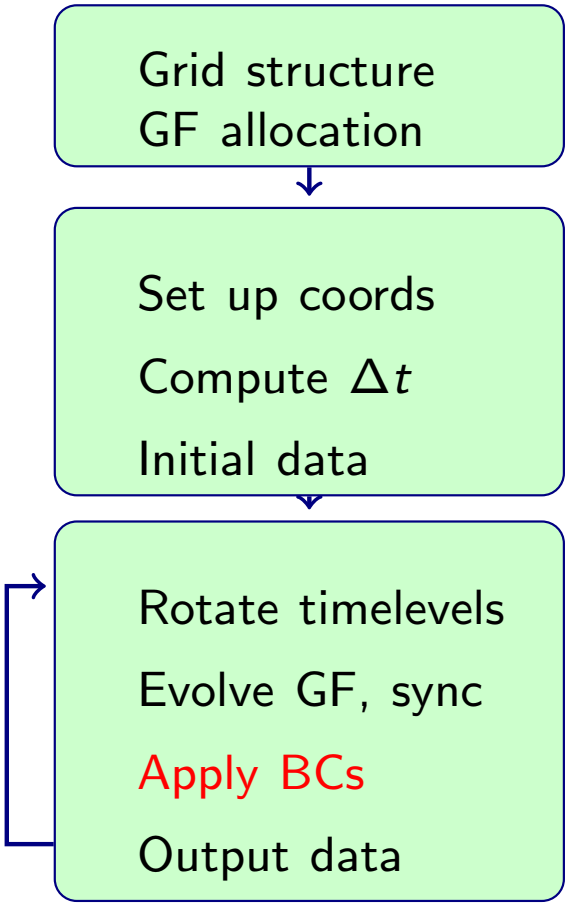
Wave Equation: Algorithm Illustration



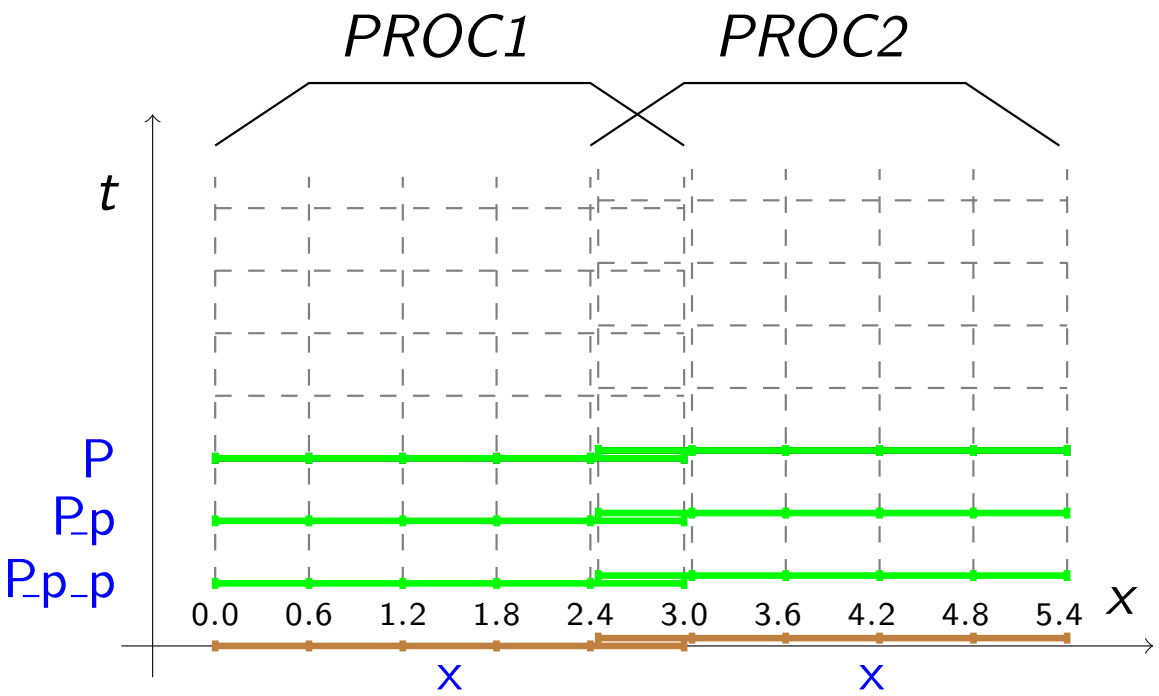


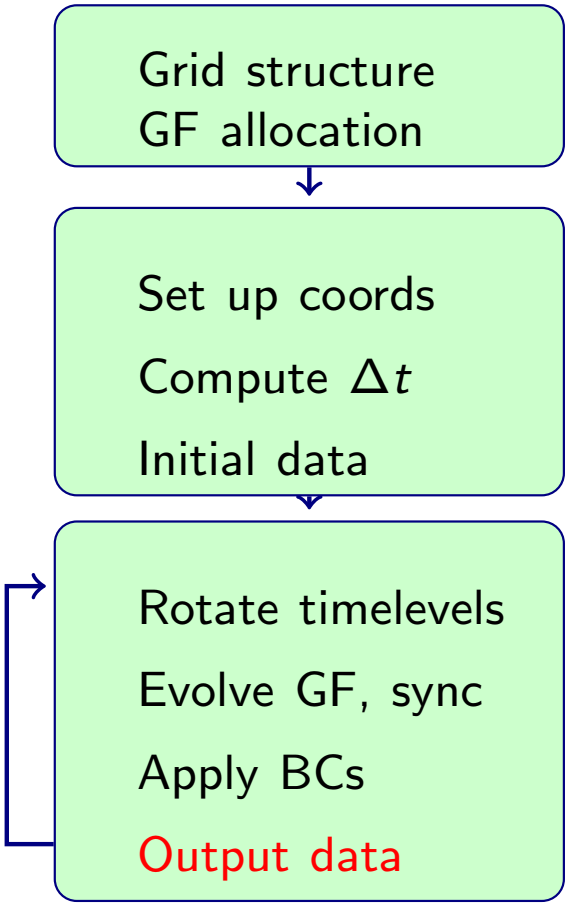
Wave Equation: Algorithm Illustration



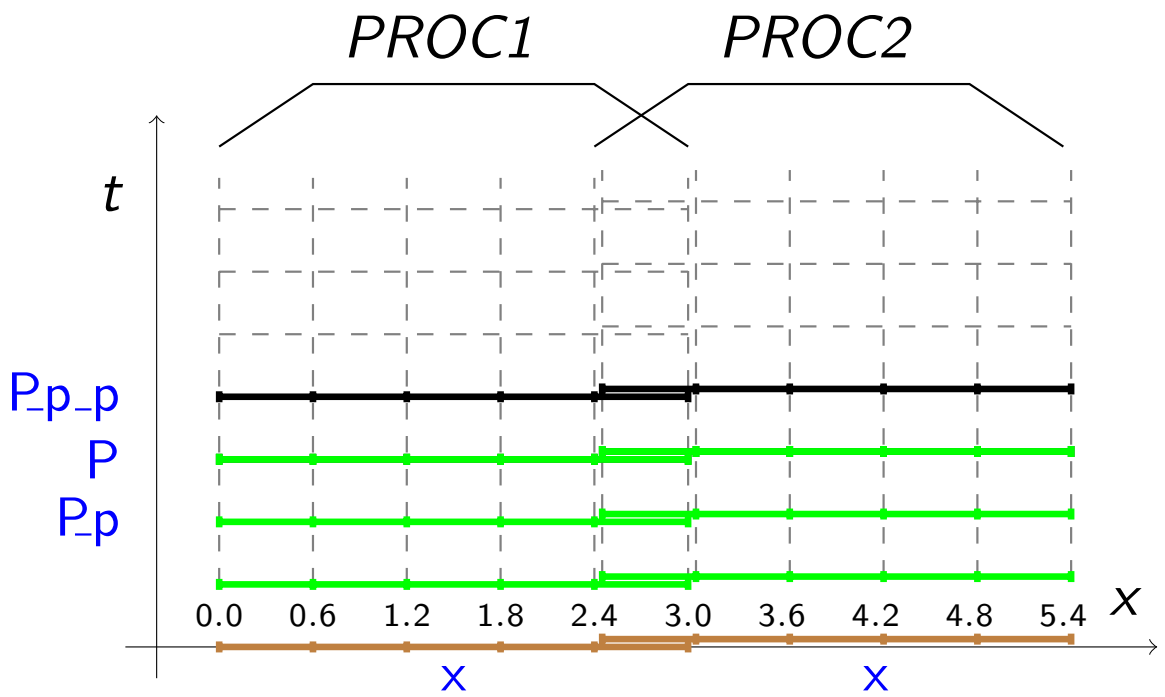


Wave Equation: Algorithm Illustration





Wave Equation: Algorithm Illustration

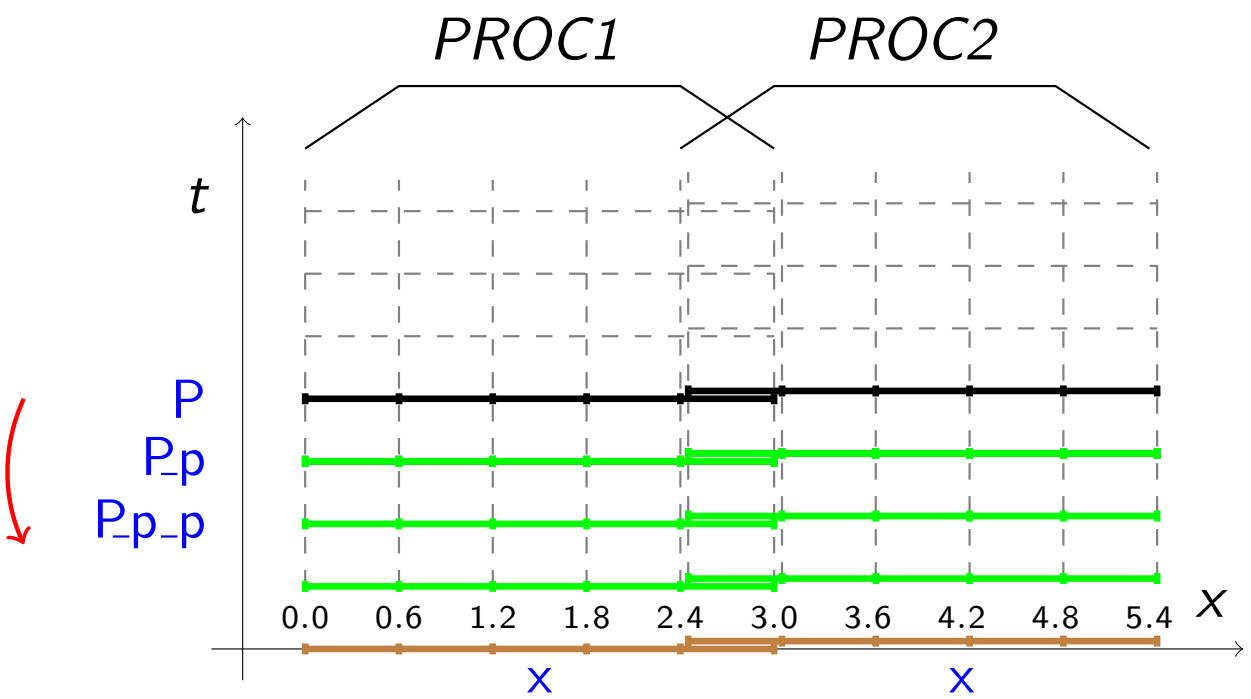


Grid structure
GF allocation

Set up coords
Compute Δt
Initial data

Rotate timelevels
Evolve GF, sync
Apply BCs
Output data

Wave Equation: Algorithm Illustration



Grid structure
GF allocation

Set up coords
Compute Δt
Initial data

Rotate timelevels
Evolve GF, sync
Apply BCs
Output data

Load Balancing

- In the previous example all processes have to perform the same operation at the same time, i.e. they progress in *lock step*.
  - If one process finish early, it has to idle (wastes time).
  - If one process finish late, all others have to idle (much worse!)
- Remedies: try to distribute load evenly (hard to do if different amount of work has to be done for different elements) or distribute load dynamically (results in overhead).
- 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 wave equation example the overhead was 20%.
- In a realistic example (e.g. binary black hole evolution) the overhead can be much larger:
  - Assume $30^3 = 27,000$ grid points per process (3D).
  - With 5 ghost zones (high order finite differencing) we then have $(30 + 2 \cdot 5)^3 = 40^3 = 64,000$ grid points with ghosts.
  - Thus we have $40^3 - 30^3 = 37,000$ ghost points per process.
  - This is an overhead of 137%!

Ghost Zone Overhead

- We need efficient parallel algorithms for current supercomputers in every corner of the program (Amdahl)!
- MPI is the 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 processors don't idle.
  - Potentially overlap communication with computation in order to hide the latency and overhead of the communication.

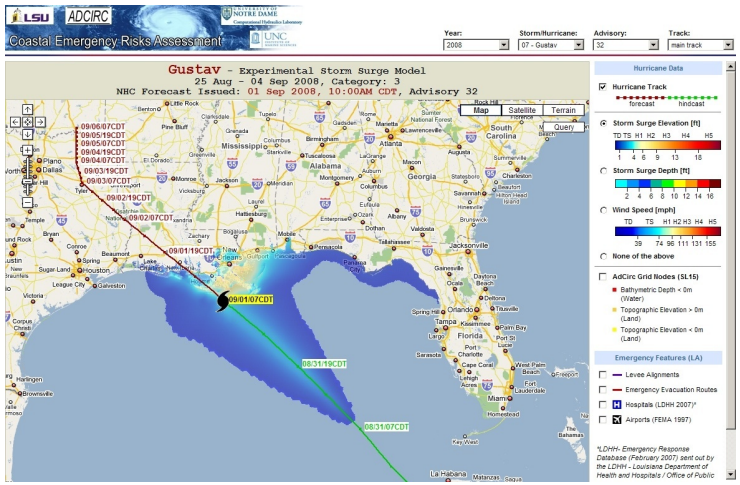
Component Model

Simulation Code Requirements

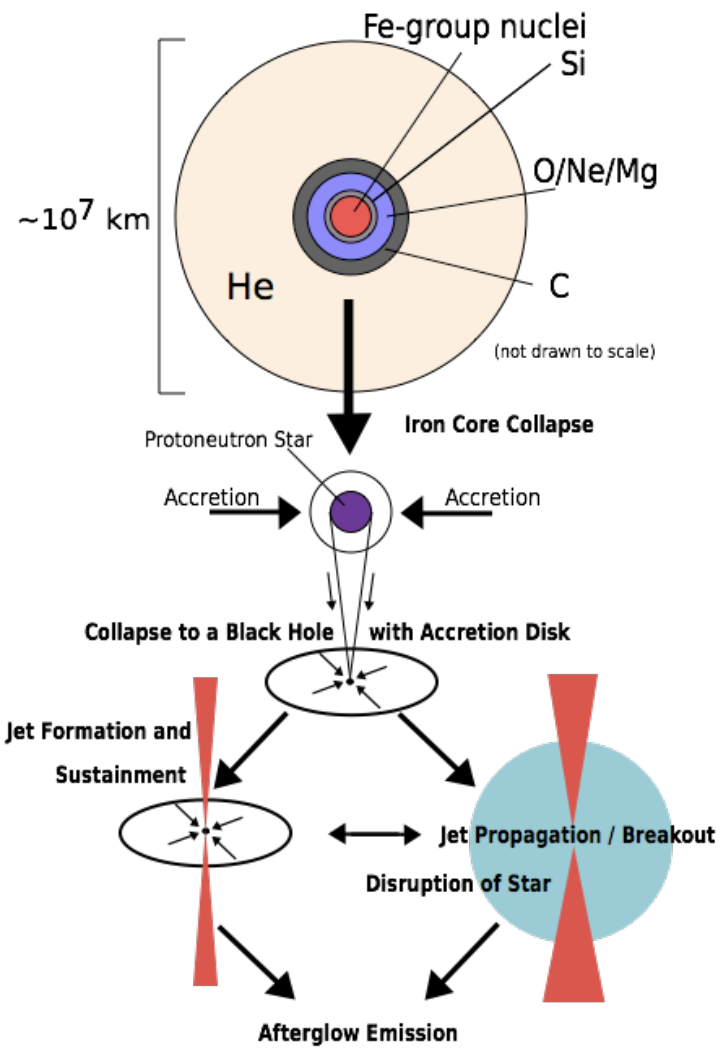
- Reliability: So that we can trust the results.
- Extensibility: So that researchers can add and experiment with new ideas.
- Usability: So that graduate (or under graduate) students don't waste too much time.
- Performance: So that we don't waste valuable resources.

Complex simulations

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



Example: Long Gamma-Ray Burst



- General Relativity (black hole).
- Relativistic hydrodynamics (star).
- Microphyscis, 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 discretisations methods.
- Computer scientists need to contribute:
  - A software architecture that makes this possible in a safe yet efficient manner.
- The physicist, mathematician and computer scientist need to work closely together (overcome language barrier) in order to achieve a good implementation.

Collaboration Problems

- Example: The Einstein Toolkit (not untypical)
  - Parts of the code is 13+ years old.
  - Graduate students leave after 3 productive years.
  - Post docs may only be around for 1 or 2 years before moving on.
  - Many original authors are not available anymore.
  - Developers distributed over many locations on several continents.
  - Most physicists/mathematicians are not good programmers.

Component Architecture

- Split program into independent *components*.
- A *framework* provides lean glue between these.
- Each component is developed independently by a small group of developers.
- The end user assembles the components needed to perform the simulations.
- There is no central control.
- There is no authoritative version.

Component Framework

- Basic principle: *control inversion* where main program is provided by framework and components look like libraries.
  - No component is "more important".
- The Framework itself does no real work. It just glues components together.
  - Components don't interact directly with each other. Only via the framework and the rules set by the framework.

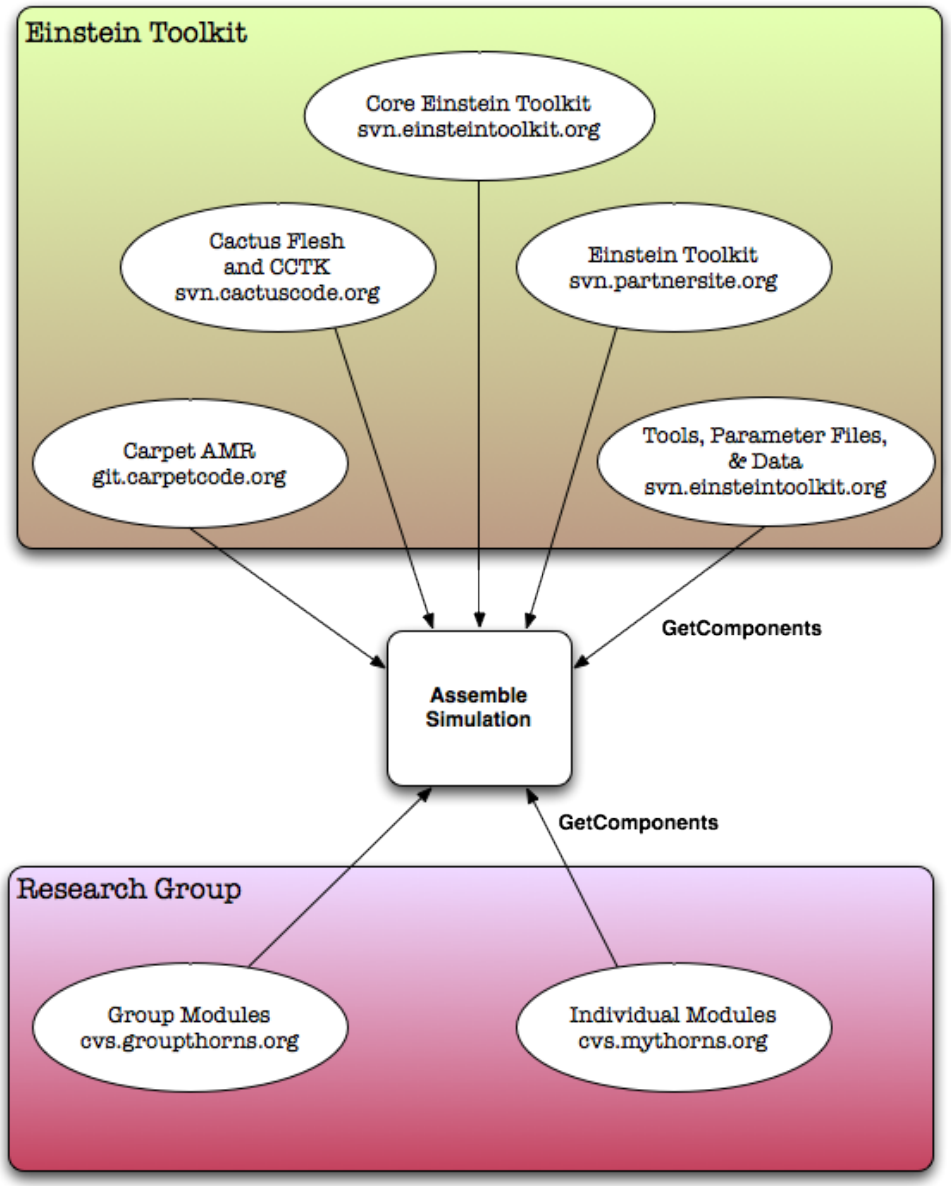
Example: The Einstein Toolkit

einstein toolkit

- Goal: Have state-of-the-art set of tools for Numerical Relativity available as open source.
- Organized by the Einstein Consortium.
- Open to everyone.
- See `http://einsteintoolkit.org`

LSU

Example: The Einstein Toolkit



The Einstein Toolkit: People

- About 50 contributors over the past decade, both from physics and computer science. Many has left the field.
- Currently 95 members from 50 different groups and institutions in 15 different countries.
- Currently 9 maintainers from 6 different sites.
- $>$ 250 publications, $>$ 40*theses* in physics, astronomy and computer science building on these components.
- Countless talks at the major astrophysics conferences/meetings are based on these components.

The Einstein Toolkit: Components sampler

- Evolution systems (both spacetime and matter).
- Boundary conditions (both symmetry and physical).
- Initial conditions (both spacetime and matter).
- Excision/Turduckening.
- Wave extraction.
- Horizon finding.
- Time stepping methods (multiple).
- Finite differencing.
- Adaptive Mesh Refinement (AMR) driver.
- I/O methods (output and checkpoint/restart).
- Web server.
- Twitter client.

Component Model Summary

- Modern simulation codes are complex and can contain multiple physics models.
- The component model can provide the necessary abstraction and encapsulation.
- The software *framework* provides the glue between components and allow the definition of clean interfaces.
- Important for research: Enables loosely coupled long-distance collaborations.