CSC 7700: Scientific Computing
Module C: Simulations and Application Frameworks
Lecture 4: Getting Science Out of Computing

Dr. Erik Schnetter

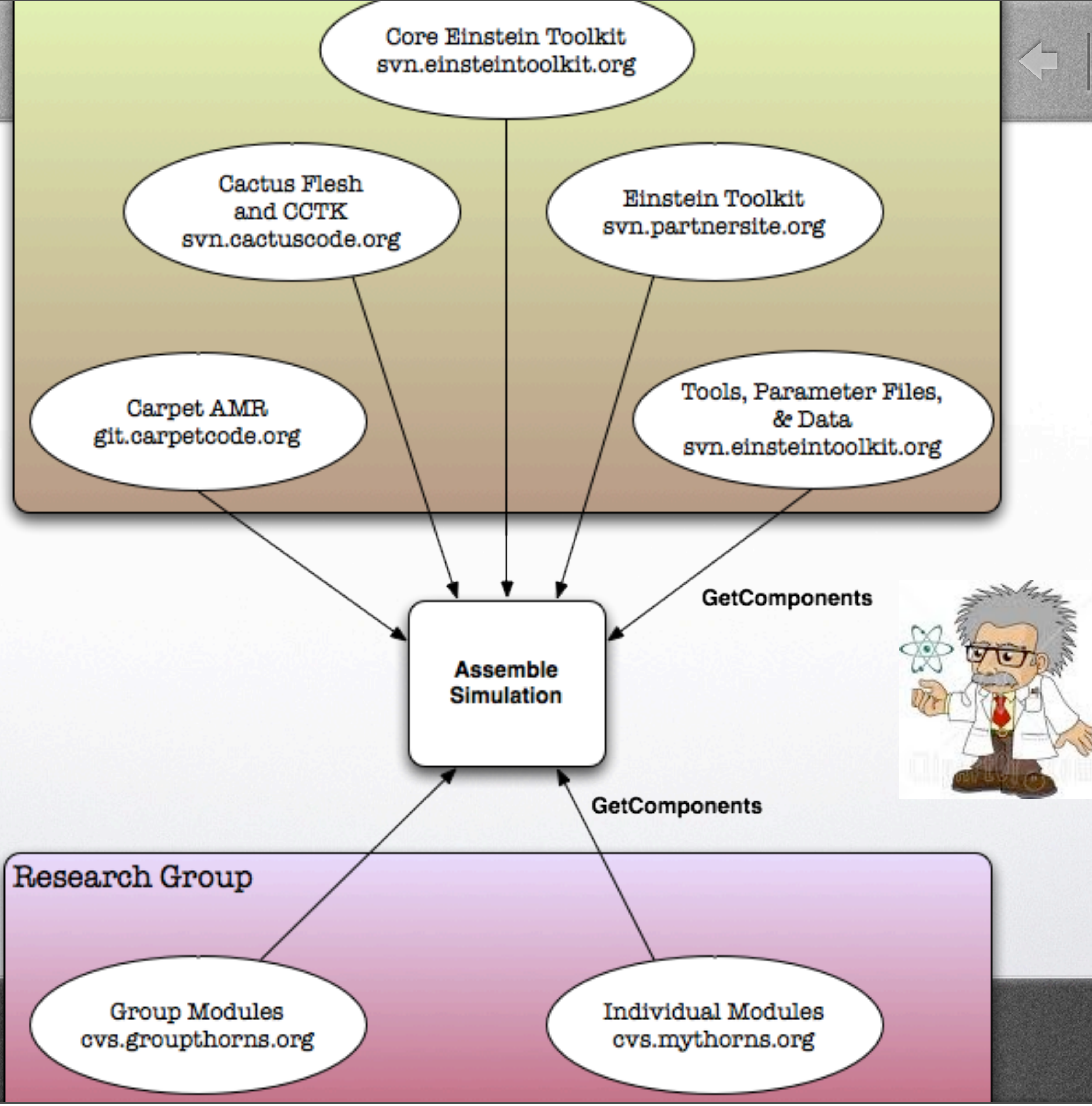
Today's Plan

- Review Cactus design principles
- Advanced framework concepts
  - computational efficiency
- Scientific programming
- Simulation data management

Cactus Design Principles Review

Basic Framework Design Idea

- Applications consist of many components (*thorns*), glued together by the framework (*flesh*)
- Framework provides main program, components are libraries (control inversion)
- End user assembles code, no central control, easy to add or replace components


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

Basic Component Design Idea

- Components are independent entities
- Ideally, a component can stand on its own, without requiring specific other components (higher level and more independent than e.g. C++ class)
- Components need to declare their interfaces (variables, routines, etc.) to the framework

Some Important Cactus Concepts

- <u>Grid function</u>: container for discretised quantity (e.g. kinetic energy density)
- <u>Thorn list</u>: list of components that should be built into a Cactus configuration (required for building)
- <u>Parameter file</u>: list of components that should be activated, plus their parameter settings (required for running)

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

Parallelisation in Cactus

- Implicit parallelisation: No explicit MPI calls in source code, simplifies programming significantly (allows beginners to write parallel code)
- Grid functions provide distributed data structure
- SYNC statements in schedule determine communication (when ghost zones need updating)

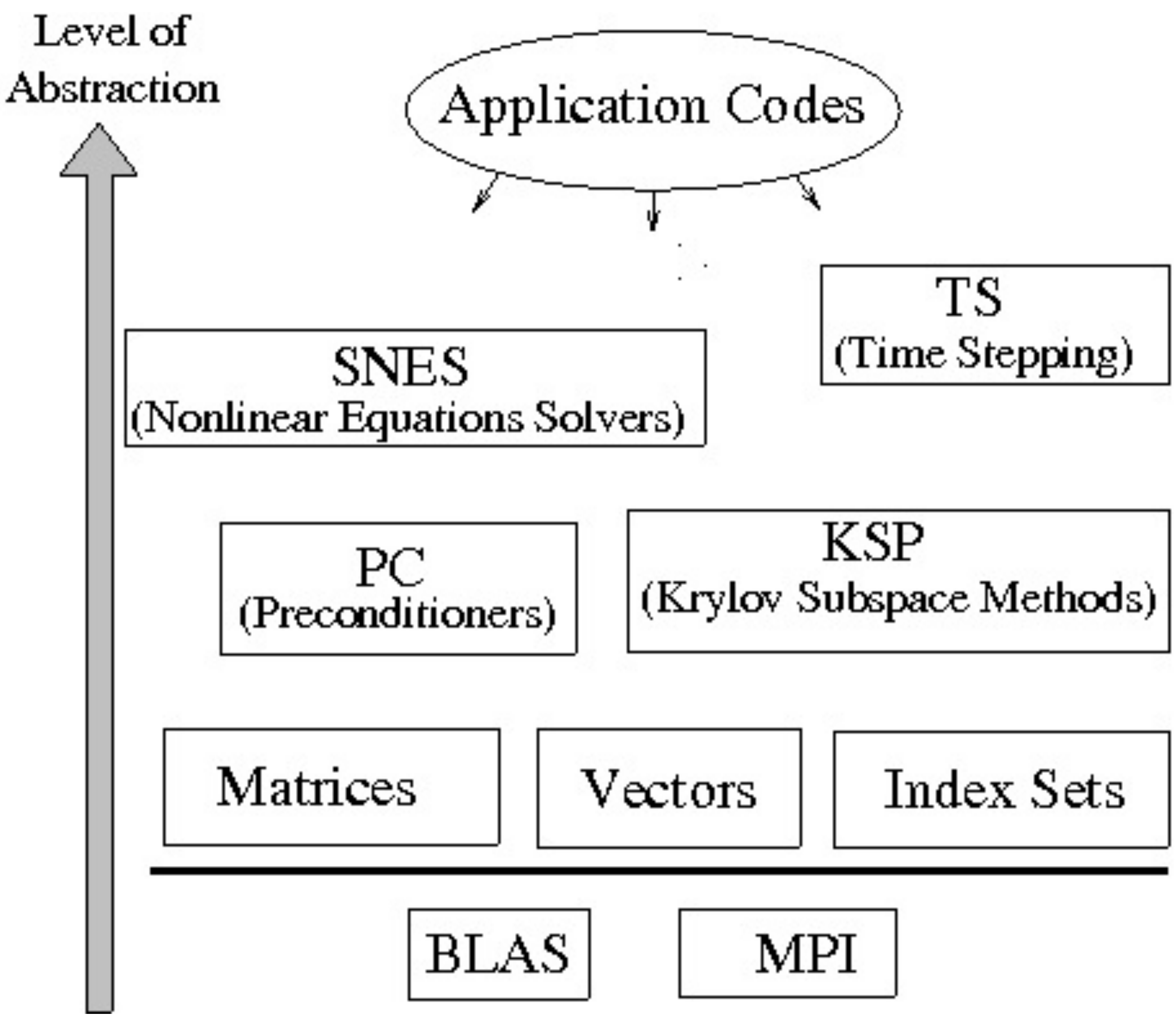
Questions about Cactus programming?

Advanced Framework Concepts

Other Frameworks

- Introducing and comparing several other frameworks for scientific computing
  - PETSc (not a framework)
  - MpCCI (industry standard, commercial)
  - SIERRA (Sandia, not public?)
  - CCA (not alive any more?)

PETSc (Portable, Extensible Toolkit for Scientific Computation)

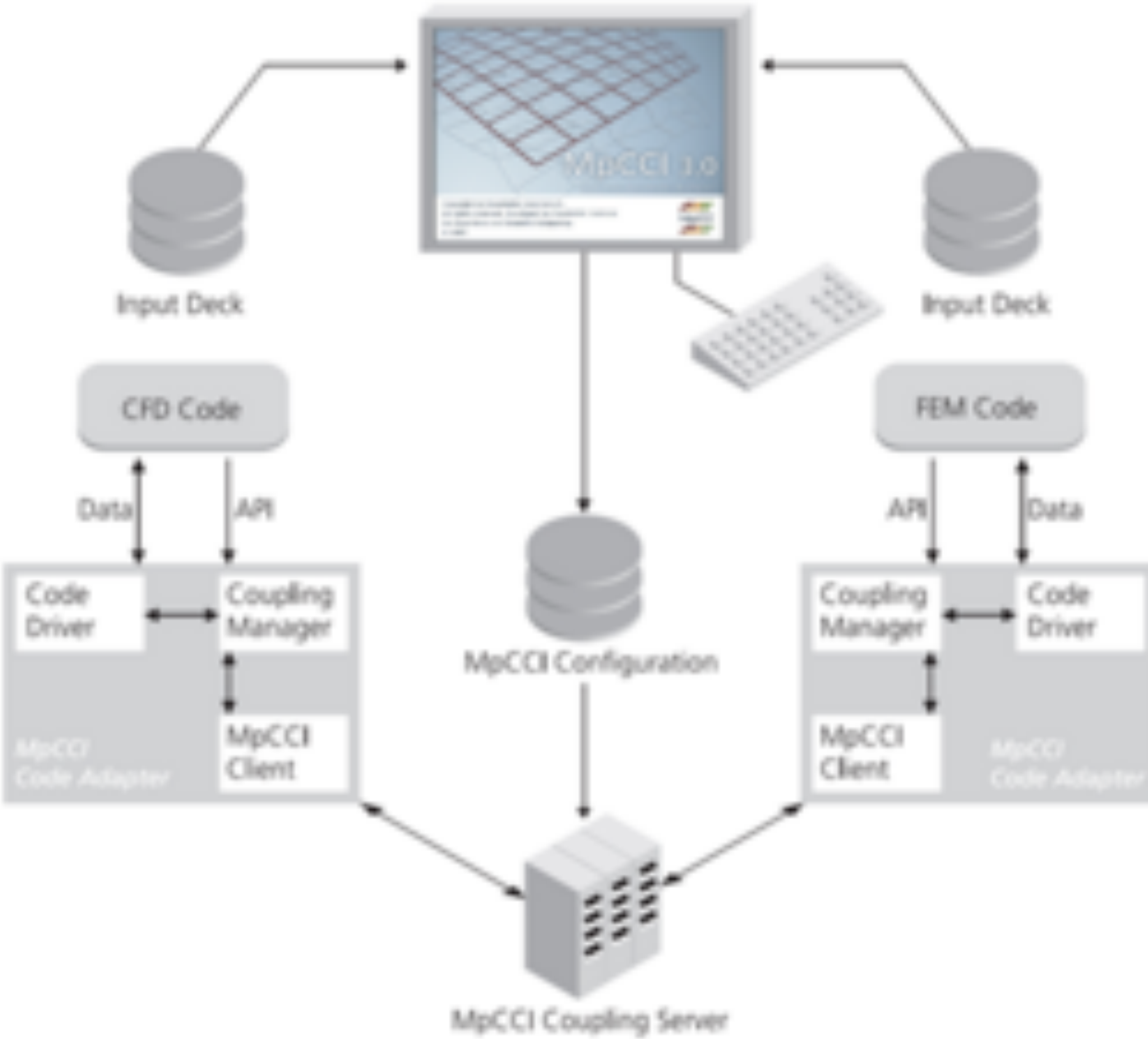


- Solver for sparse linear algebra problems
- Also other solvers based on the above
- Infrastructure for sparse linear algebra

PETSc

http://www.mcs.anl.gov/petsc/

MpCCI (Multi-physics Code Coupling Interface)

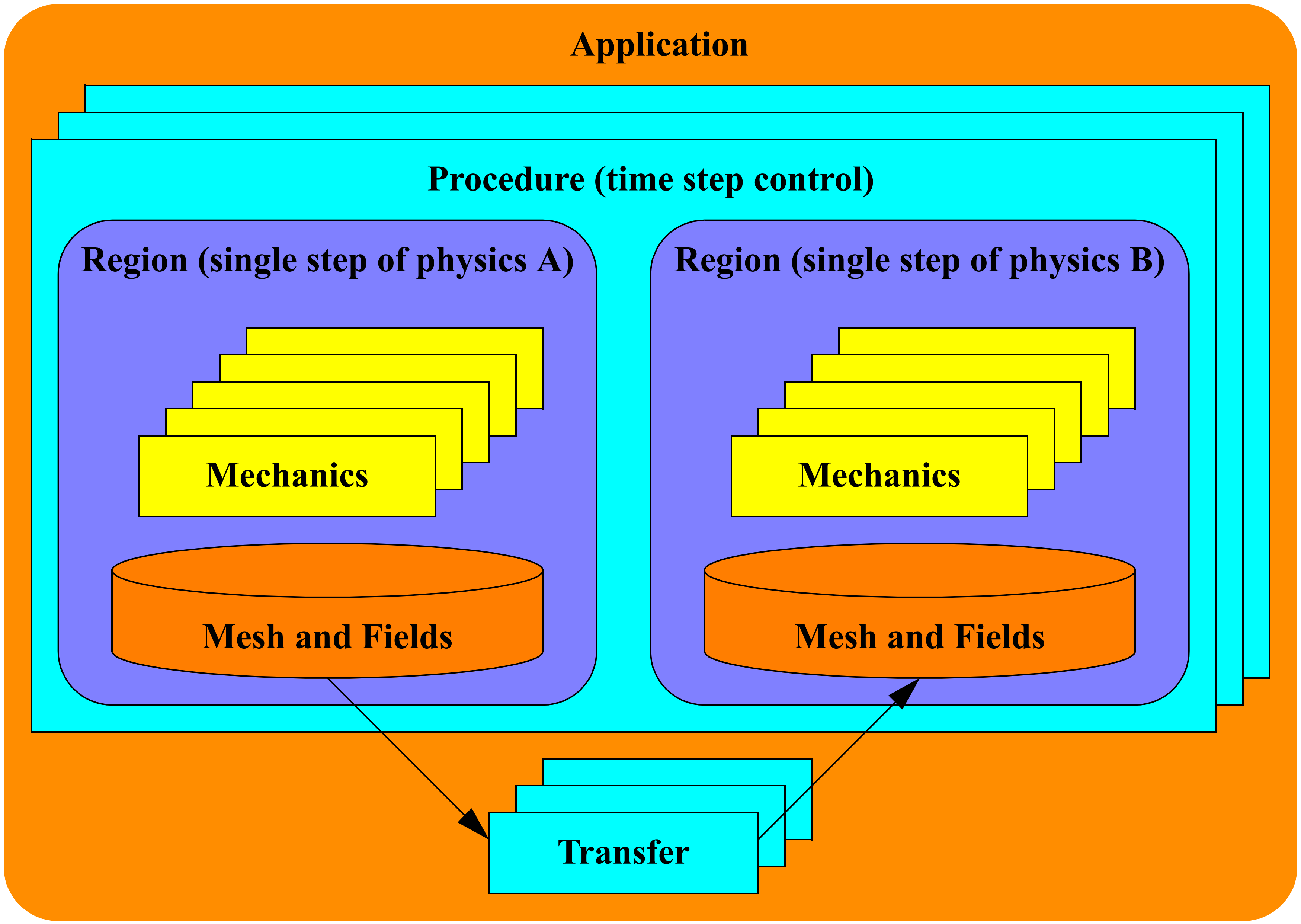


- Can couple arbitrary existing programs (that support some kind of external interface)
- Interface is implemented as additional program

Fraunhofer
SCAI

<u>http://www.mpcci.de/</u>

SIERRA



- Framework targeting Finite Elements (FEM)
- Simulation can have independent regions, coupled via common control mechanism

http://prod.sandia.gov/techlib/access-control.cgi/2002/023616.pdf

SIERRA

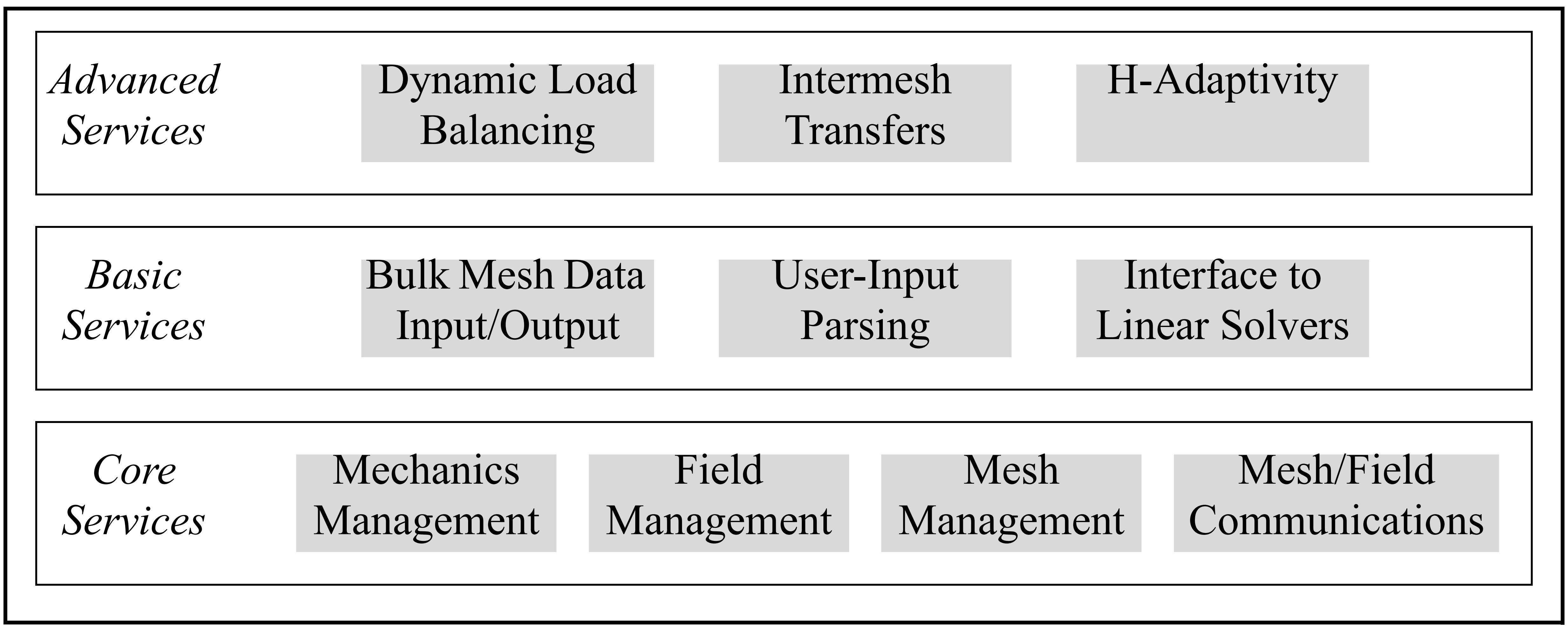
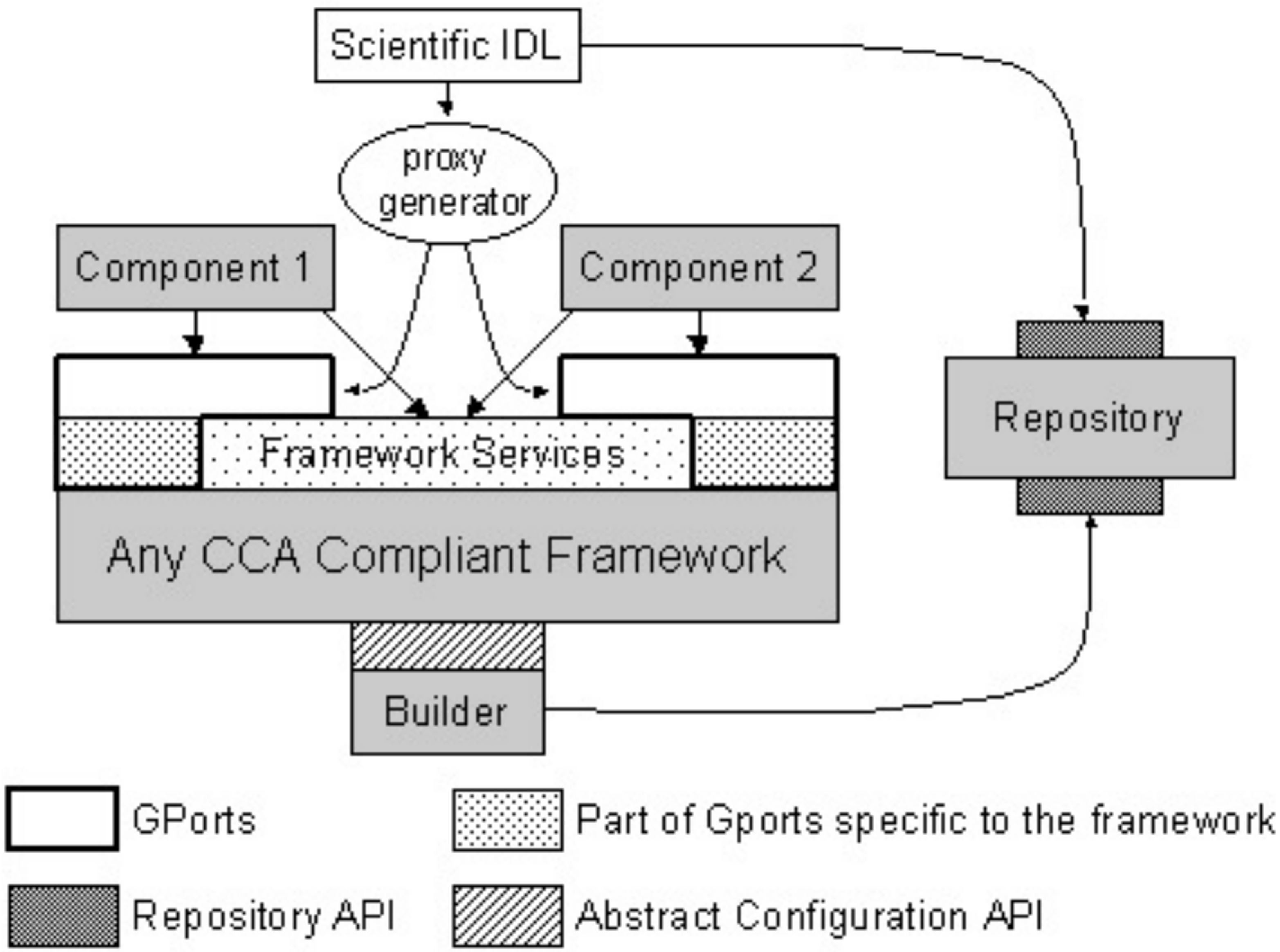


Figure 1.2. SIERRA Framework layered set of services.

CCA (Common Component Architecture)



- Not a framework, but a specification for framework/ component interfaces
- High-level language for defining interfaces: IDL

CCA Common Component Architecture

http://www.cca-forum.org/

Cactus: Driver Thorn

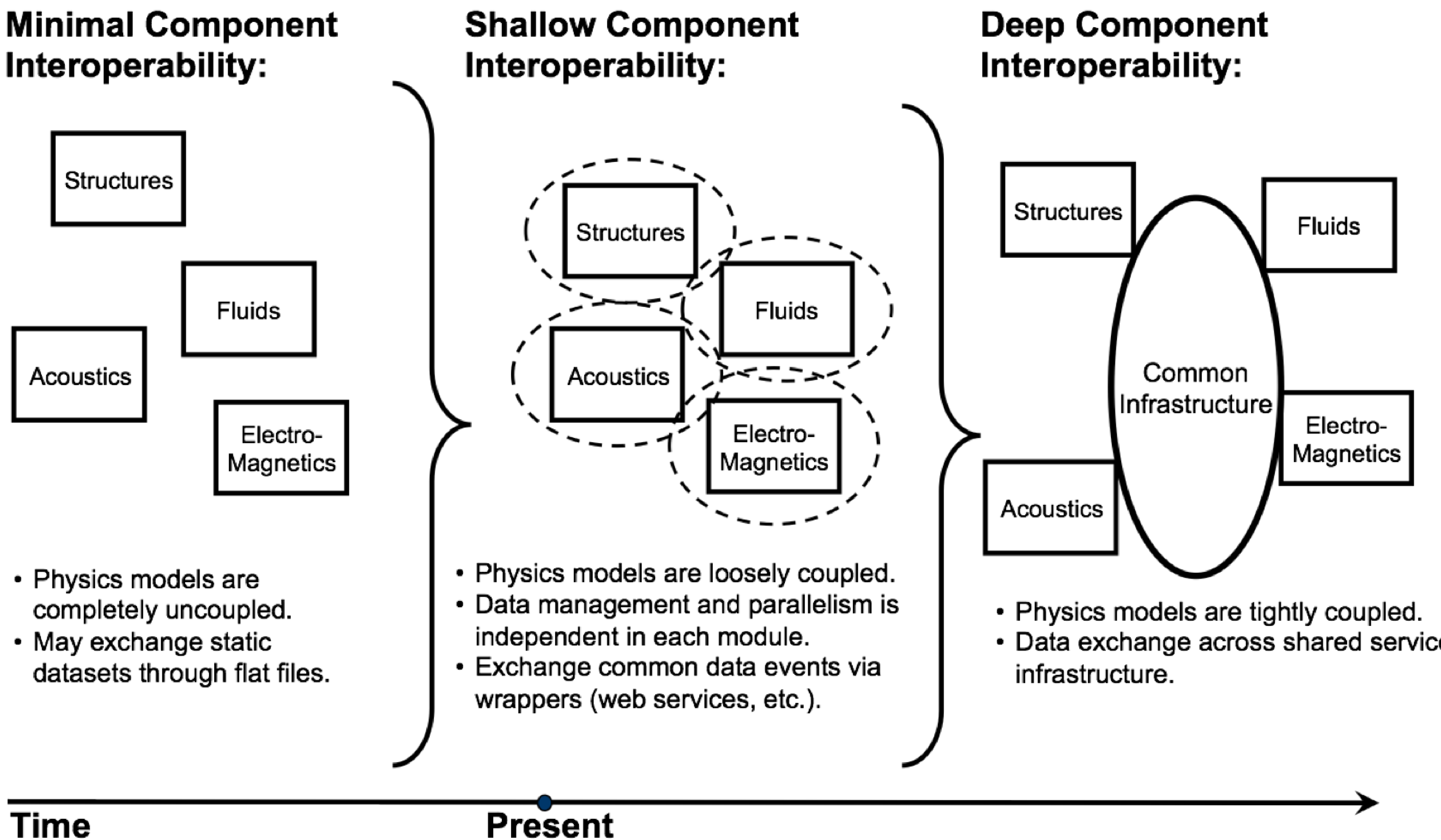
- A *driver* is a special thorn in Cactus that implements parallelism (and memory management)
- Implements "grid function" data type
- This *externalises parallelism*, so that other thorns don't have to implement parallel algorithms
- However, this places certain restrictions onto other thorns

Cactus: Driver Thorn

- There must be exactly one driver active (standard Cactus driver is *PUGH*)
- Driver can provide advanced discretisation methods, such as AMR or multi-block (e.g. *Carpet* driver)
- Driver can be based on existing parallel library (e.g. Chombo, Samrai)
- Driver (or related thorn) also provides I/O

Component Interoperability

[from HPC ASC White Paper]



Application Efficiency

Data Access

- Simulations handle much data (previous example: 1billion elements)
- Cannot easily copy data: not enough memory, takes too much time
- Therefore, if possible each process must compute with the data it owns ("bring computation to data", opposite of a function)

Data Sharing

- Different components may need to access the same data
  - (Example: your homework project thorn needs to access density, velocity from HydroBase)
- If components are very independent, then data need to be copied
- If data cannot be copied, then the components must interact in some (non-trivial) way

Component Coupling

- MpCCI: components don’t interact, data is exchanged (copied) via “interface process”
- SIERRA (within single region), Cactus: data are stored by framework/driver, different components can access same data
  - Components then have to be designed for a specific framework, are not general-purpose any more

Component Coupling

- No Coupling: independently executing programs, data “sharing” requires copying files
- Loose Coupling: *independent* data management in each component, data sharing requires copying
- Tight Coupling: data are managed outside of components (or by special component), data *sharing is efficient*, but components need to rely on external data manager

Component Safety

- Efficient data sharing between components requires running in the same address space
- This also means that thorns can (accidentally?) modify each other's data, errors (e.g. array index out of bounds) can propagate between components
- Only compile-time access control and coding standards can provide some safety

Advanced Framework Concepts Summary

- Many simulation frameworks with many different designs exist
- Fundamental design question is: how tight are components coupled?
- Tight coupling requires shared data management between components
- Trade-off between independence/ease-of-programming/safety and efficiency

Scientific Programming

Shared Code Development

- Developing a large code as a group is different from small-scale programming
    - There is old code (ten years old) that “belongs to nobody”
    - People use “your” code without understanding it
    - People make changes to “your” code without understanding it

Shared Code Development

- Best not to have “your” or “my” code; instead share responsibility
- Everything needs to be programmed defensively, so that wrong usage is always detected
- Functionality requires test cases, so that bad changes can be detected quickly

Test Cases

- Code can be ten years old and still very good
    - Cannot rewrite old code every year (and introduce new errors every year)
- But need to make sure old code is actually still good, in view of many other changes that may have happened in the mean time
- A *test case* stores program input and expected output, so that any change in behaviour can be detected

Recovering from Errors

- Mistakes happen, need to be able to undo bad changes
- Important approach: keep complete history of all changes to the code, undo changes when necessary
- Need to use source code management system (e.g. svn); this keeps track of who made when what change

Working Together

- Source code management system also defines a *single, standard version* of the code, on which everybody is working
- It would be too confusing to send source code around by email, or look into other people's directories
- See tutorials for svn etc.; svn and relatives are indispensable for scientific code development

Policies

- Working in a group requires policies since programming is a formal task; people need to know what is acceptable
  - Coding style (routine names, indentation, commit messages)
  - Access rights (using, modifying, adding, committing)
  - Testing standards before committing changes
  - Peer review before/after making changes

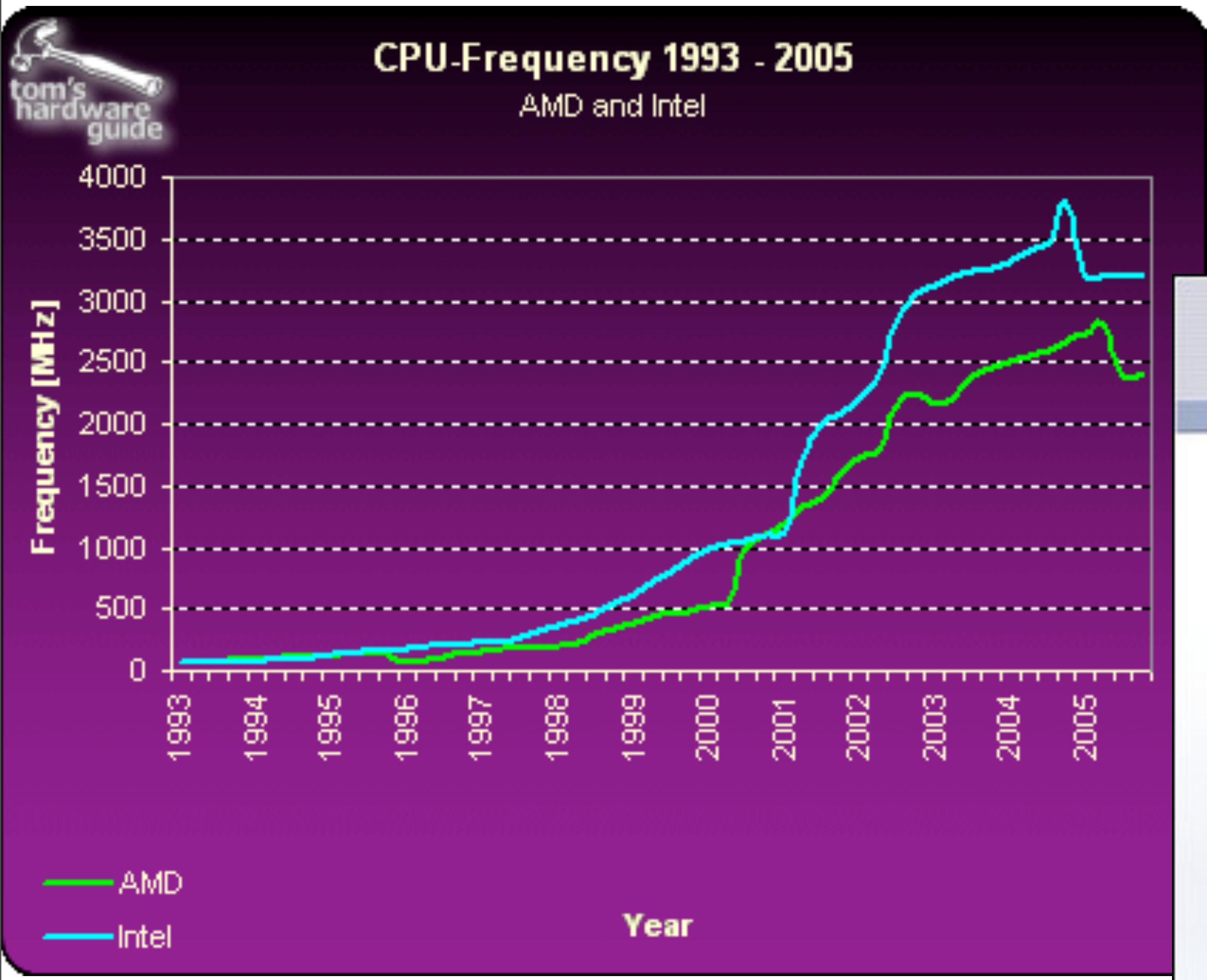
Component Life Cycle

1. Idea, experimental implementation
2. Prototype, useful for a single paper
3. Production use, more features added, most errors removed, useful for a series of papers
4. Mature, very useful, few changes
5. Outdated, used mostly for historic investigations (still somewhat useful)

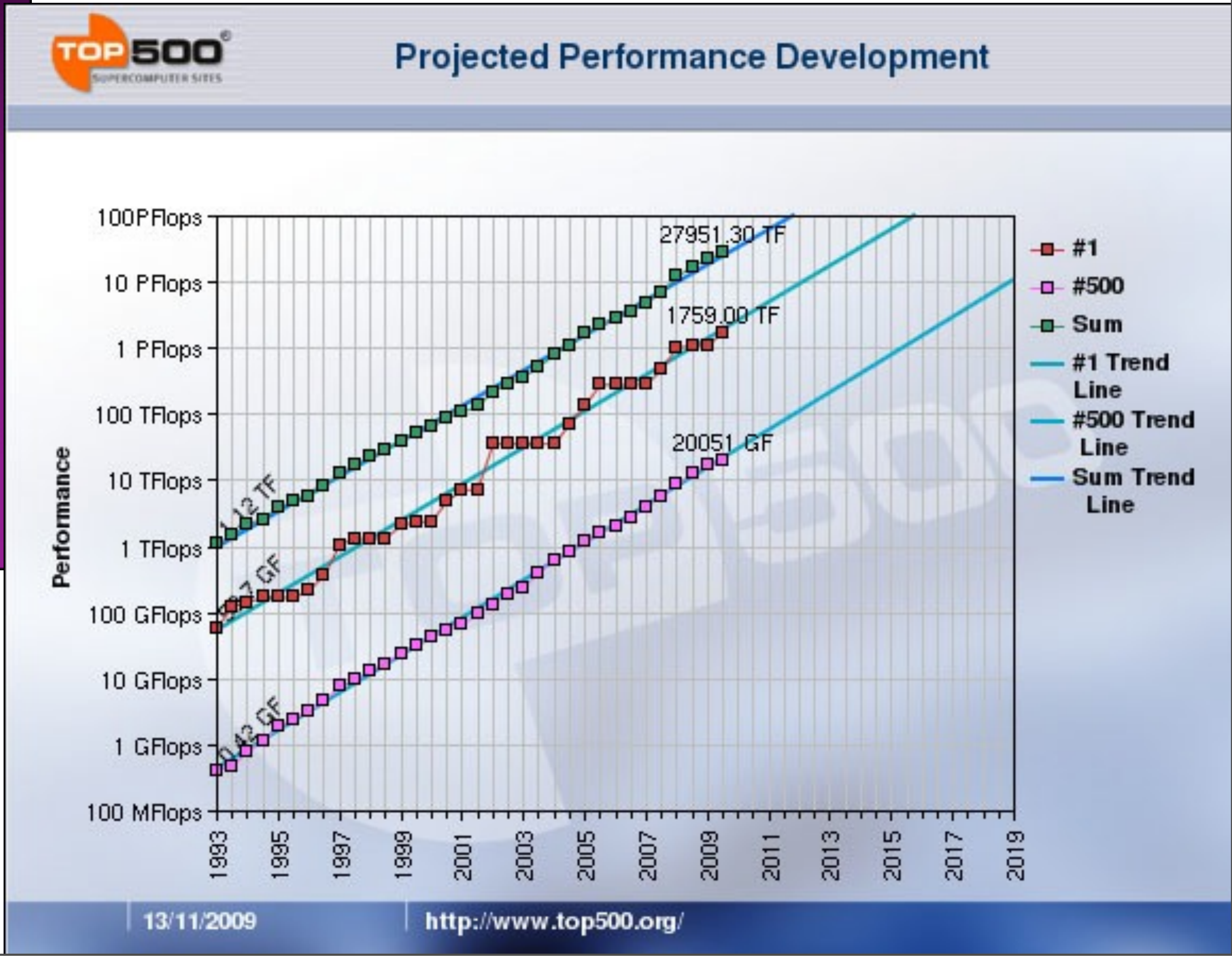
Portability

- Machines become old, outdated, and unreliable after a few years
- HPC systems frequently (once a week?) require maintenance, or are (once a year?) unavailable for a week due to system a upgrade
- Installed software (compilers) may have bugs, takes days or weeks to correct
- Therefore, scientific codes need to be portable, so that one can then quickly use other machines

New Hardware Architectures



Early multi-core revolution graph



New Hardware Architectures

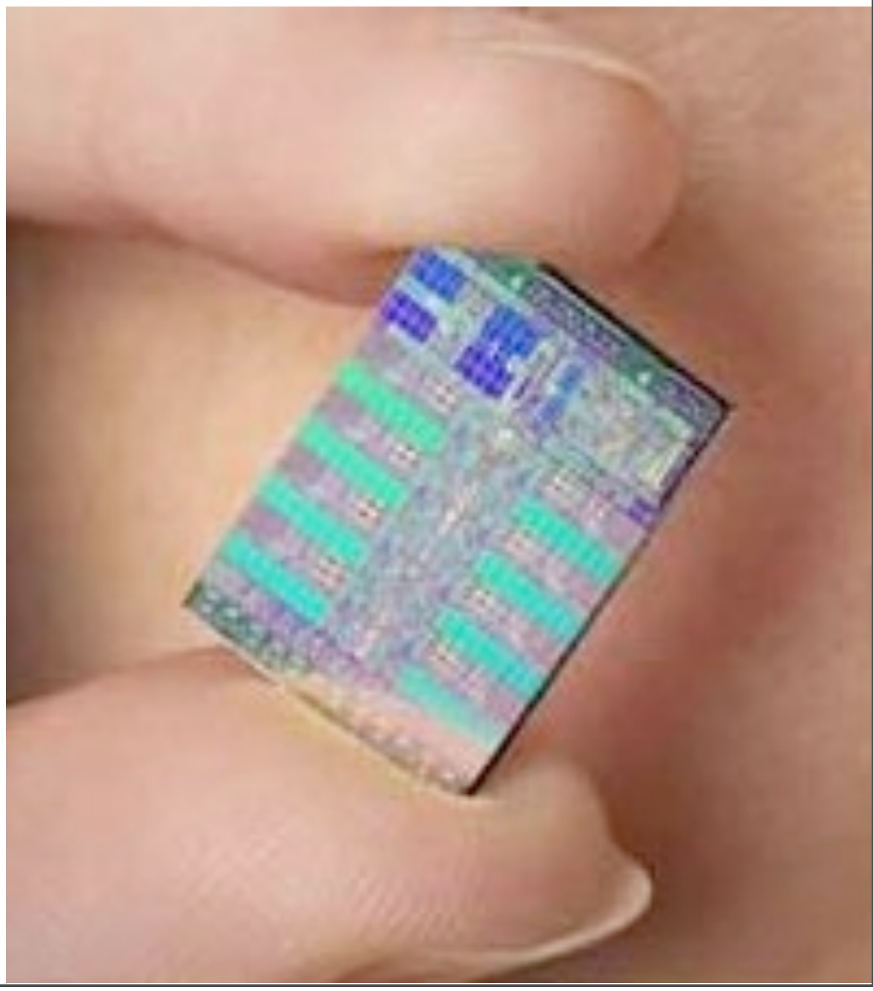
- Software stays around longer than hardware
  - software: >15 years (Cactus)
  - hardware: 3 years at most? (Moore's law)
- Software design must not only be portable, but also architecture independent

GPU Computing

- Increasing the speed of a chip is expensive
- Rule of thumb: Can either have 1 full-speed or 4 half-speed chips for same price
- Multi-core revolution: Every year, put more cores into a computer to make it more powerful
  - Does this also make it faster?

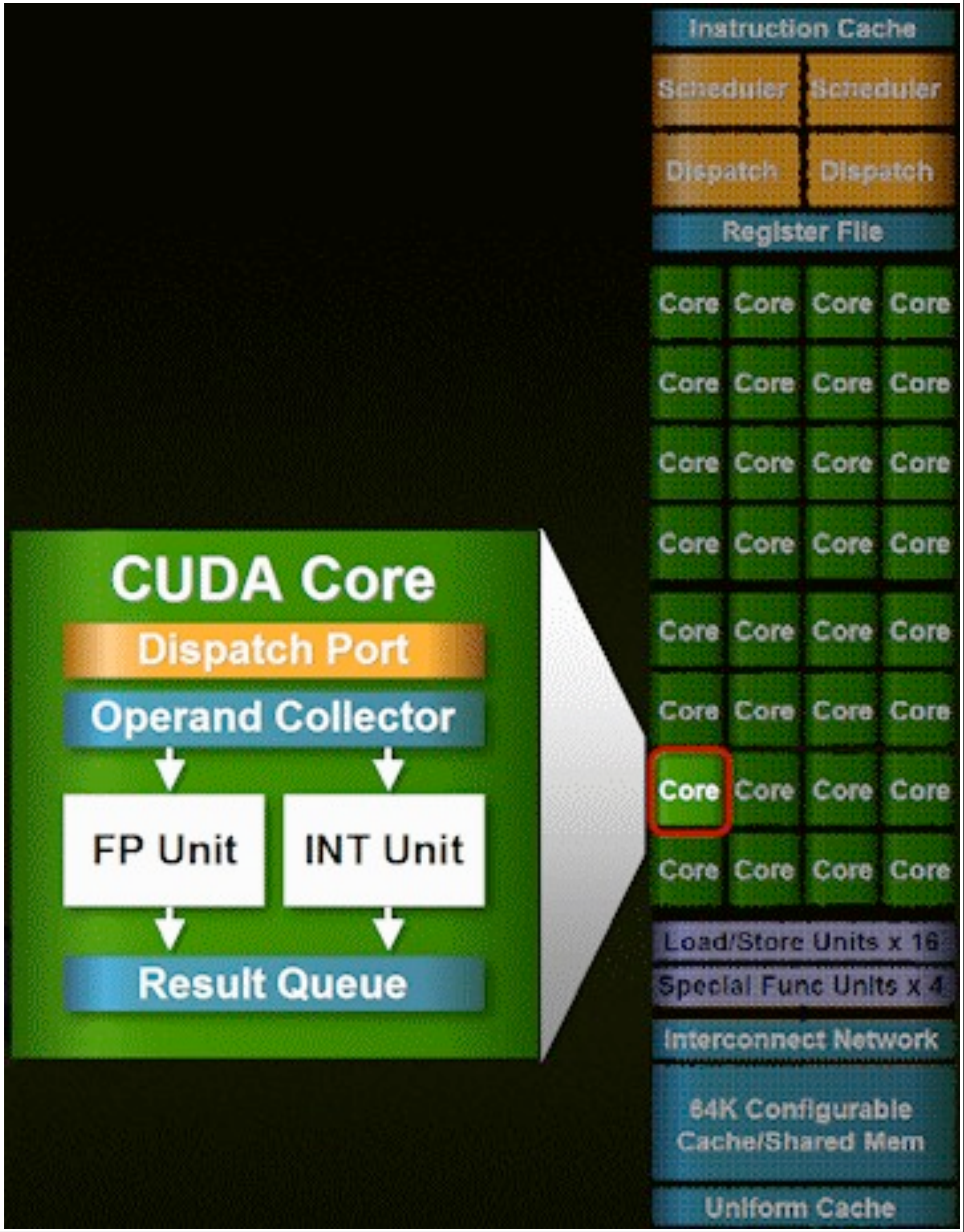
GPU Computing

- Started in 2001 with Sony/ Toshiba/IBM collaboration for *Cell Processor*
- Has 1 "real" core and 8 slower "additional" cores on a single chip
- Used in Sony's Playstation



GPU Computing

- Graphics cards (GPU) have a similar architecture with many cores
- (although these "cores" are, technically, only vector units and not really independent)



GPU Computing Challenges

- Each core is small and slow, need to use many cores just to be as fast as regular CPU
  - Slower clock speed
  - Less memory per core
- Need to use even more cores to get good speed
- But, if this is possible, then total speed is much larger than that of regular CPUs (of same total cost)

Framework Architecture Challenges

- Don't want to re-design framework and re-write code for new architectures; framework needs to isolate programmer from architectural changes
- Old architecture (clusters, MPI) has been around for 20 years, but something new is coming now (Roadrunner, first petaflop system in 2008, uses Cell processors)

Cactus Approach to Architecture Independence

- Separate physics code from computer science code (different thorns, driver)
- Each thorn "sees" only a small part of the overall problem (information hiding)
  - ideally, each physics thorn acts on a single grid point at a time, but that would have too much overhead
  - this externalises parallelism, load balancing, data distribution, data sharing (*largest Cactus success*)
  - this also significantly complicates programming (e.g. calculating total mass) (*largest Cactus problem*)

Scientific Programming Practices Summary

- Use a source code management system (strongly suggest to do this for homeworks as well)
- Keep track of software versions for each simulation, have test cases to ensure correctness
- Need portability and architecture independence to make programs future-proof

Simulation Data Management

Scientific Integrity

- Scientific results must be repeatable
- That is, one must document and preserve the exact software version for each simulation
  - Can either be very careful about this, or use automated tools to help
- Time span is at least months for single paper (review process), years for research groups, difficult for longer times (decades) since hardware changes

Simulation Data

- Simulations produce much data
- Never enough disk space to keep it all
- Often, auto-clean will delete "old" files
  - see Queen Bee "Work Volume Purge" messages you should be be receiving, e.g. this morning

TABLE 1

BINARY SEQUENCES FOR WHICH NUMERICAL SIMULATIONS HAVE BEEN CARRIED OUT, WITH DIFFERENT COLUMNS REFERRING TO THE PUNCTURE INITIAL LOCATION $\pm x/M$, THE LINEAR MOMENTA $\pm p/M$, THE MASS PARAMETERS m_i/M, THE DIMENSIONLESS SPINS a_i, THE NORMALIZED ADM MASS $\widetilde{M}_{\rm ADM} \equiv M_{\rm ADM}/M$ MEASURED AT INFINITY, AND THE NORMALIZED ADM ANGULAR MOMENTUM $\widetilde{J}_{\rm ADM} \equiv J_{\rm ADM}/M^2$. FINALLY, THE LAST SIX COLUMNS CONTAIN THE NUMERICAL AND FITTED VALUES FOR $|v_{\rm kick}|$ (IN km/s), $a_{\rm fin}$ AND THE CORRESPONDING ERRORS.

	$\pm x/M$	$\pm p/M$	m_1/M	m_2/M	a_1	a_2	$\widetilde{M}_{\rm ADM}$	$\widetilde{J}_{\rm ADM}$	$\|v_{\rm kick}\|$	$\|v^{\rm fit}_{\rm kick}\|$	err. (%)	$a_{\rm fin}$	$a^{\rm fit}_{\rm fin}$	err. (%)
$r0$	3.0205	0.1366	0.4011	0.4009	-0.584	0.584	0.9856	0.825	261.75	258.09	1.40	0.6891	0.6883	0.12
$r1$	3.1264	0.1319	0.4380	0.4016	-0.438	0.584	0.9855	0.861	221.38	219.04	1.06	0.7109	0.7105	0.06
$r2$	3.2198	0.1281	0.4615	0.4022	-0.292	0.584	0.9856	0.898	186.18	181.93	2.28	0.7314	0.7322	0.11
$r3$	3.3190	0.1243	0.4749	0.4028	-0.146	0.584	0.9857	0.935	144.02	146.75	1.90	0.7516	0.7536	0.27
$r4$	3.4100	0.1210	0.4796	0.4034	0.000	0.584	0.9859	0.971	106.11	113.52	6.98	0.7740	0.7747	0.08
$r5$	3.5063	0.1176	0.4761	0.4040	0.146	0.584	0.9862	1.007	81.42	82.23	1.00	0.7948	0.7953	0.06
$r6$	3.5988	0.1146	0.4638	0.4044	0.292	0.584	0.9864	1.044	45.90	52.88	15.21	0.8150	0.8156	0.07
$r7$	3.6841	0.1120	0.4412	0.4048	0.438	0.584	0.9867	1.081	20.59	25.47	23.70	0.8364	0.8355	0.11
$r8$	3.7705	0.1094	0.4052	0.4052	0.584	0.584	0.9872	1.117	0.00	0.00	0.00	0.8550	0.855	0.00
$ra0$	2.9654	0.1391	0.4585	0.4584	-0.300	0.300	0.9845	0.8250	131.34	132.58	0.95	0.6894	0.6883	0.16
$ra1$	3.0046	0.1373	0.4645	0.4587	-0.250	0.300	0.9846	0.8376	118.10	120.28	1.85	0.6971	0.6959	0.17
$ra2$	3.0438	0.1355	0.4692	0.4591	-0.200	0.300	0.9847	0.8499	106.33	108.21	1.77	0.7047	0.7035	0.17
$ra3$	3.0816	0.1339	0.4730	0.4594	-0.150	0.300	0.9848	0.8628	94.98	96.36	1.46	0.7120	0.7111	0.13
$ra4$	3.1215	0.1321	0.4757	0.4597	-0.100	0.300	0.9849	0.8747	84.74	84.75	0.01	0.7192	0.7185	0.09
$ra6$	3.1988	0.1290	0.4782	0.4602	0.000	0.300	0.9850	0.9003	63.43	62.19	1.95	0.7331	0.7334	0.04
$ra8$	3.2705	0.1261	0.4768	0.4608	0.100	0.300	0.9852	0.9248	41.29	40.55	1.79	0.7471	0.7481	0.13
$ra10$	3.3434	0.1234	0.4714	0.4612	0.200	0.300	0.9853	0.9502	19.11	19.82	3.72	0.7618	0.7626	0.11
$ra12$	3.4120	0.1209	0.4617	0.4617	0.300	0.300	0.9855	0.9750	0.00	0.00	0.00	0.7772	0.7769	0.03
$s0$	2.9447	0.1401	0.4761	0.4761	0.000	0.000	0.9844	0.8251	0.00	0.00	0.00	0.6892	0.6883	0.13
$s1$	3.1106	0.1326	0.4756	0.4756	0.100	0.100	0.9848	0.8749	0.00	0.00	0.00	0.7192	0.7185	0.09
$s2$	3.2718	0.1261	0.4709	0.4709	0.200	0.200	0.9851	0.9251	0.00	0.00	0.00	0.7471	0.7481	0.13
$s3$	3.4098	0.1210	0.4617	0.4617	0.300	0.300	0.9855	0.9751	0.00	0.00	0.00	0.7772	0.7769	0.03
$s4$	3.5521	0.1161	0.4476	0.4476	0.400	0.400	0.9859	1.0250	0.00	0.00	0.00	0.8077	0.8051	0.33
$s5$	3.6721	0.1123	0.4276	0.4276	0.500	0.500	0.9865	1.0748	0.00	0.00	0.00	0.8340	0.8325	0.18
$s6$	3.7896	0.1088	0.4002	0.4002	0.600	0.600	0.9874	1.1246	0.00	0.00	0.00	0.8583	0.8592	0.11
$t0$	4.1910	0.1074	0.4066	0.4064	-0.584	0.584	0.9889	0.9002	259.49	258.09	0.54	0.6868	0.6883	0.22
$t1$	4.0812	0.1103	0.4062	0.4426	-0.584	0.438	0.9884	0.8638	238.37	232.62	2.41	0.6640	0.6658	0.27
$t2$	3.9767	0.1131	0.4057	0.4652	-0.584	0.292	0.9881	0.8265	200.25	205.21	2.48	0.6400	0.6429	0.45
$t3$	3.8632	0.1165	0.4053	0.4775	-0.584	0.146	0.9879	0.7906	174.58	175.86	0.73	0.6180	0.6196	0.26
$t4$	3.7387	0.1204	0.4047	0.4810	-0.584	0.000	0.9878	0.7543	142.62	144.57	1.37	0.5965	0.5959	0.09
$t5$	3.6102	0.1246	0.4041	0.4761	-0.584	-0.146	0.9876	0.7172	106.36	111.34	4.68	0.5738	0.5719	0.33
$t6$	3.4765	0.1294	0.4033	0.4625	-0.584	-0.292	0.9874	0.6807	71.35	76.17	6.75	0.5493	0.5475	0.32
$t7$	3.3391	0.1348	0.4025	0.4387	-0.584	-0.438	0.9873	0.6447	35.36	39.05	10.45	0.5233	0.5227	0.11
$t8$	3.1712	0.1419	0.4015	0.4015	-0.584	-0.584	0.9875	0.6080	0.00	0.00	0.00	0.4955	0.4976	0.42
$u1$	2.9500	0.1398	0.4683	0.4685	-0.200	0.200	0.9845	0.8248	87.34	88.39	1.20	0.6893	0.6883	0.15
$u2$	2.9800	0.1384	0.4436	0.4438	-0.400	0.400	0.9846	0.8249	175.39	176.78	0.79	0.6895	0.6883	0.17
$u3$	3.0500	0.1355	0.3951	0.3953	-0.600	0.600	0.9847	0.8266	266.39	265.16	0.46	0.6884	0.6883	0.01
$u4$	3.1500	0.1310	0.2968	0.2970	-0.800	0.800	0.9850	0.8253	356.87	353.55	0.93	0.6884	0.6883	0.01

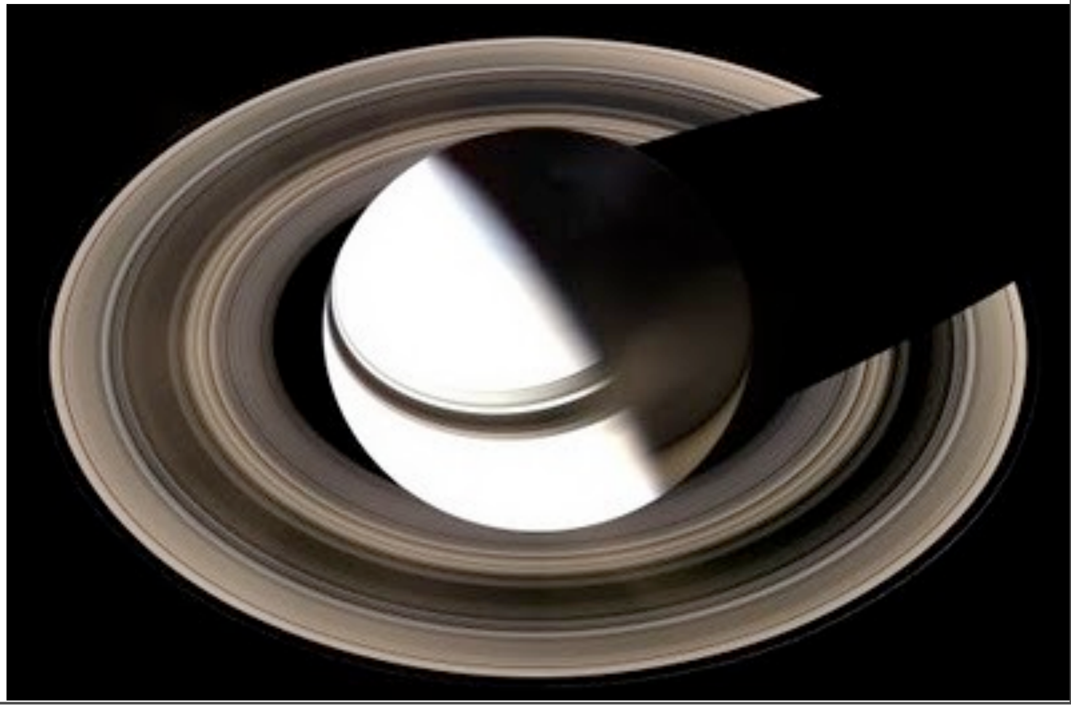
[2008]

Simulation Laboratory

- Publications usually based on multiple simulations
  - ...performed by different people at different times on different systems...
  - ...with different versions of the code...
- Tedious to keep track of everything, administrative burden

Data Curation

- Simulations are expensive ($1M for previous table?)
- Scientific results should be "eternal", must (?) be able to go back and re-analyse data even after years
  - Can you?
- What if project is interesting and takes years to complete?
  - How long will your class projects survive?
  - If they don't, then why are they useful?



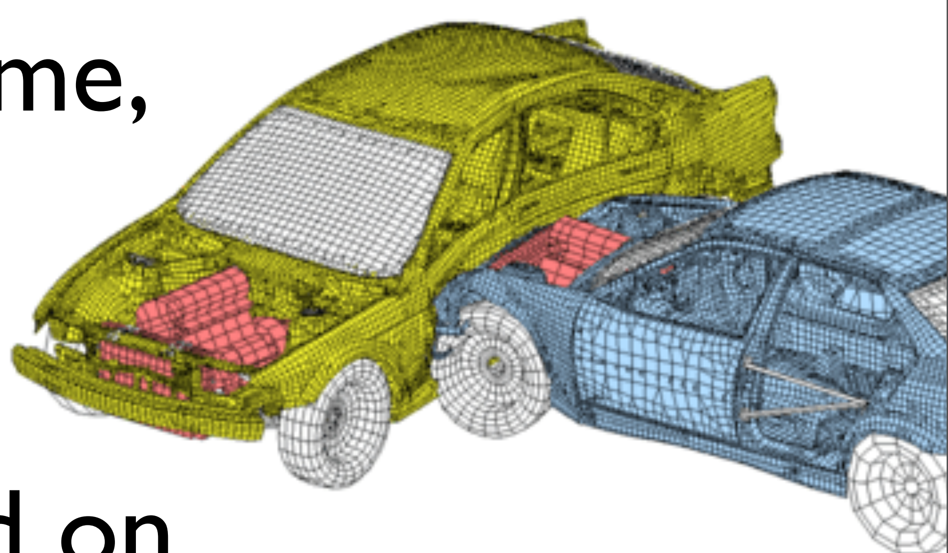
Finding Data

- If you indeed keep all old school programming projects around, how do you find something later if you need it?
- How do you find simulation data?
  - If machines are shut down after several years?
  - If they were performed by others who left the group years ago?
  - Do you just spend another $1M to re-run the simulations? Does the old code still work? (How much does it cost to archive the data? To keep the code working?)

Summary

Simulation Science Basics

- Physics model described by PDEs
- PDEs discretised in space and time, leading to many (billions) of elements



- Discrete equations implemented on large supercomputers
- A simulation iterates many (millions of) times to produce its result

Simulating Complex Systems

- Require parallelism for efficient executing
  - These days, most supercomputers have cluster architecture programmed with MPI
  - Domain decomposition is the prevalent strategy for parallelism
- Real-world simulation codes are large, framework model can provide good architecture for this

Framework Architecture

- A framework provides glue to combine independent components
  - This allows large-scale, loosely-coupled program development, ideal for today's international research groups
- Cactus is such a framework
  - Components need to declare their interfaces to the framework

Getting Science Out of Computing

- Components can be loosely or tightly coupled, depending on independence/ efficiency trade-off
- On HPC systems, data placement defines efficiency since moving data is expensive
- Consider new/future system architectures
- Scientific integrity requires archiving simulation data