CSC 7700: Scientific Computing

Module D: Simulations and Application Frameworks

Lecture 4: Getting Science Out of Computing

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.
  - Supercomputers from the application scientist's point of view.
- Lecture 2 introduced:
  - Parallelization: data structures, load balancing, domain decomposition.
  - Software Engineering: multi-physics simulations, large projects, distributed code development.
- Lecture 3 introduced:
  - The component model as software architecture for real-world simulation codes.
  - The Cactus Software Framework as a specific example.
- In this lecture we will discuss:
  - Additional framework concepts.
  - Scientific programming.

Summary

Summary

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

Summary

- Parallel algorithms 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.

Summary

- We introduced the Cactus framework.
- Applications consist of many components (*thorns*) glued together by the framework (*flesh*).
- Cactus provides the main program while components are libraries.
- The end user can mix and match the thorns necessary for a specific problem and control which thorns are active at runtime.
- Thorns have implementation (regular code) and interface (ccl) files.
- Thorns "talk" to each other only through well-defined interfaces and an API provided by the flesh.
- The MPI parallellisation issues are (mostly) hidden from the application programmer (`SYNC` statements in schedule determines ghost zone updates).

Additional Framework Concepts

Cactus: Driver Thorn

- A *driver* is a special thorn in Cactus that implements parallelism and memory management.
- The driver implements the "grid function" data type (as well as "grid arrays").
- This *externalizes parallelism* so that other thorns don't have to implement parallel algorithms
- However, this places certain restrictions onto other thorns.
- There must be exactly one driver active (standard Cactus driver is PUGH).
- The driver can provide advanced discretisation methods, such as AMR or multi-block (e.g. the *Carpet* driver).
- The driver can be based on an existing parallel library (e.g. Chombo or Samrai).
- The driver (or closely related thorn) also provides I/O.

Application efficiency

Data Access

- Simulations handle large data sets (previous example: 1 billion elements).
- Cannot easily copy data:
  - Not enough memory.
  - It takes too much time.
- If possible, each process must compute with the data it owns ("bring computation to data").
- In Cactus, work routines are called on each process with access to the data owned by the process.

Data Sharing

- Different components may need to access the same data.
  - Example: A spacetime evolution thorn needs access to the stress energy tensor and a hydrodynamics evolution thorn needs access to the spacetime metric.
- If components are very independent, data need to be copied.
- If data cannot be copied, the components must interact in some (non-trivial) way.
- In Cactus this is done by inheritance: A thorn can have direct access to another thorn's data.

Component Coupling

- How closely are components coupled in a framework?

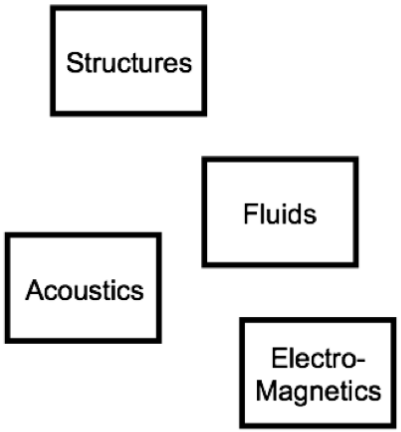
  No Coupling: Independently executing programs. Data "sharing" requires writing/copying/reading files.

  Loose Coupling: Independent data management and parallelism in each component. Data sharing requires memory transfers.

  Tight Coupling: Data are managed outside of components (or by a special component). Data sharing is efficient (components share access to the same memory), but components need to rely on an external data manager.

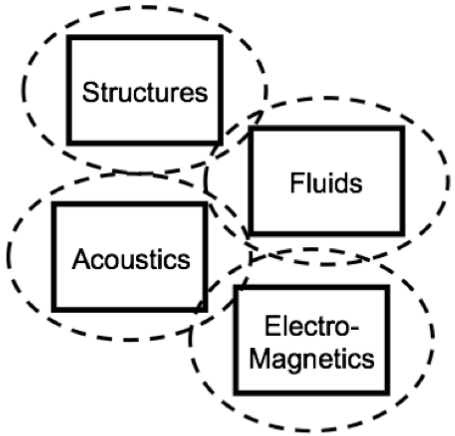
Component Coupling

Minimal Component Interoperability:



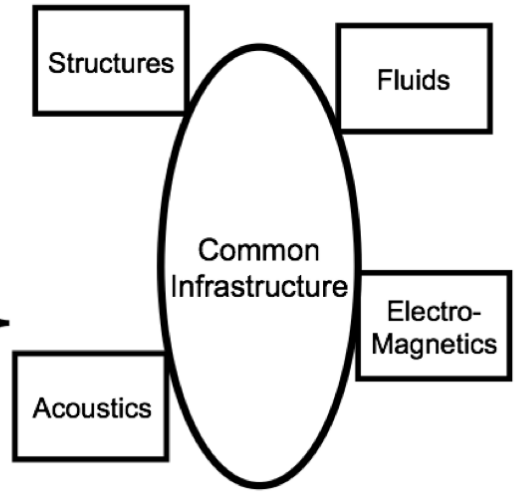
- Physics models are completely uncoupled.
- May exchange static datasets through flat files.

Shallow Component Interoperability:



- Physics models are loosely coupled.
- Data management and parallelism is independent in each module.
- Exchange common data events via wrappers (web services, etc.).

Deep Component Interoperability:



- Physics models are tightly coupled.
- Data exchange across shared service infrastructure.

Component Safety

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

Additional 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 (or community) is different from small-scale programming.
  - There is old code ($>$ 10 years old) that "belongs to nobody".
  - People use "your" code without understanding it.
  - People make changes to "your" code without understanding it.
- Best not to have "your" or "my" code. Instead share responsibility.
- Program defensively, so that wrong usage is (always) detected.
- There need to be a testing mechanism so that bad changes can be detected quickly.

Test Cases

- Code can be > 10 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 working, despite the many other changes to the framework and other components that it interacts with.
- A *test case* stores program input and expected output so that any change in behavior can be detected.
- Test cases can also be used to test portability. Should get the same result on different architectures to within roundoff error.

Recovering From Errors

- Mistakes happen (bugs) and it should be possible to undo bad changes to the code.
- It is important, therefore, to keep the complete history of all changes to the code in order to be able to undo changes when necessary.
- Need to use source code management tools such as subversion, darcs, git, mercurial... This not only keeps track of the changes to the code but also who made them.

Working Together

- A source code management system also defines a *single standard version* of the components on which everybody is working.
- It would be too confusing to send source code around by email or look into other directories.
- Source code management systems also allows for temporary branches for heavy development when adding new features without disturbing people doing production runs.
- Source code management systems are indispensable for scientific code development.
- Tutorials for source code management systems are available online.

Policies

- Working in a group on a code base requires some policies regarding:
  - Coding style (routine names, indentation, commit messages).
  - Access rights (using, modifying, adding, committing).
  - Testing standards before committing changes.
  - Peer review before/after making changes.
- It is necessary to know what is acceptable behavior.

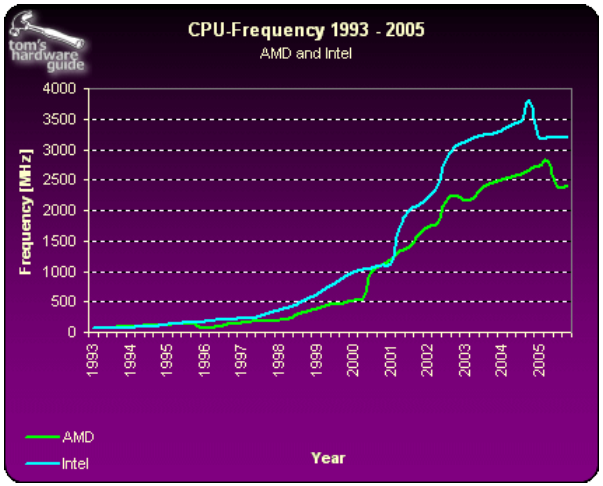
Component Life Cycle

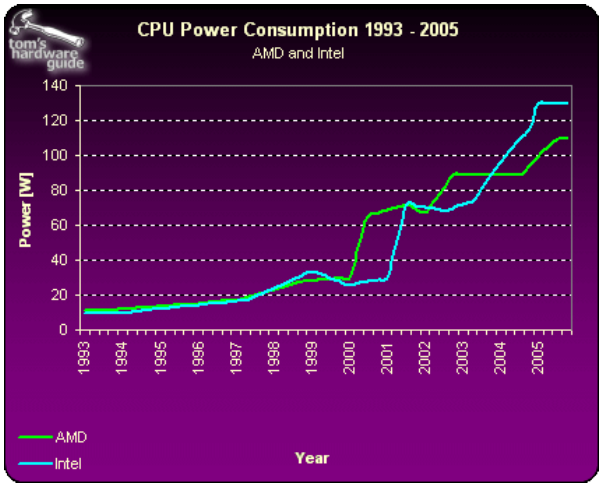
- Idea, experimental implementation.
- Prototype, useful for a single paper.
- Production code, more features added, most bugs removed, useful for a series of papers.
- Mature code, very useful, few changes.
- Outdated, used mostly for historic investigations but still somewhat useful.

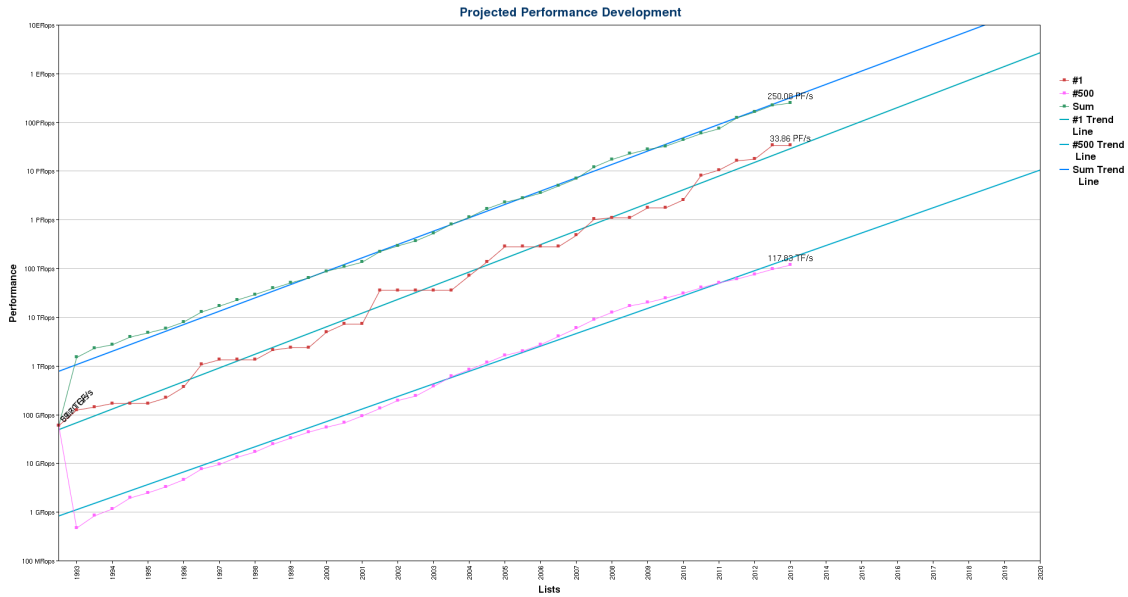
Portability

- Machines become old, outdated and unreliable after a few years, while new machines become available.
- HPC systems frequently (sometimes once a week!) require maintenance or a unavailable for longer periods of time for an upgrade (maybe once a year!).
- Installed software (compilers) may have bugs that makes a machine unusable until fixed.
- Therefore, scientific codes need to be portable so that one can then quickly use other machines.

New Hardware Architectures







New Hardware Architectures

- Software stays around much longer than hardware.
  - Software: $>$ 15 years (Cactus).
  - Hardware: 3 years on average (5 years at most).
- Software design must not only be portable but also architecture independent.
- Software has to be adaptable when architecture changes dramatically.

Multi Core CPUs

- The clock speed has not increased since 2005 whereas the transistor density has continued increasing.
- End result: more and more cores on a chip resulting in nodes with multiple cores able to access the same shared memory.
- Could in principle continue to use pure MPI parallelization.
- However, remember memory overhead due to ghost zones and additional computational overhead from domain decomposition.
- Scaling can suffer at very large core counts.

OpenMP

- Better scaling may be obtained by using OpenMP (Open Multi-Processing) parallelization within a shared memory node and MPI parallelization between nodes.
- OpenMP is a shared memory parallelization paradigm based on lightweight threads and workload distribution between threads.
- Parallisation is implemented using compiler directives to tell the compiler how to distribute the work in a loop among threads.
- Parallelization can be done incrementally (i.e. loop by loop).
- OpenMP directives ignored when compiling with a sequential compiler.
- Works with F77, F90, C and C++.

OpenMP examples

C:

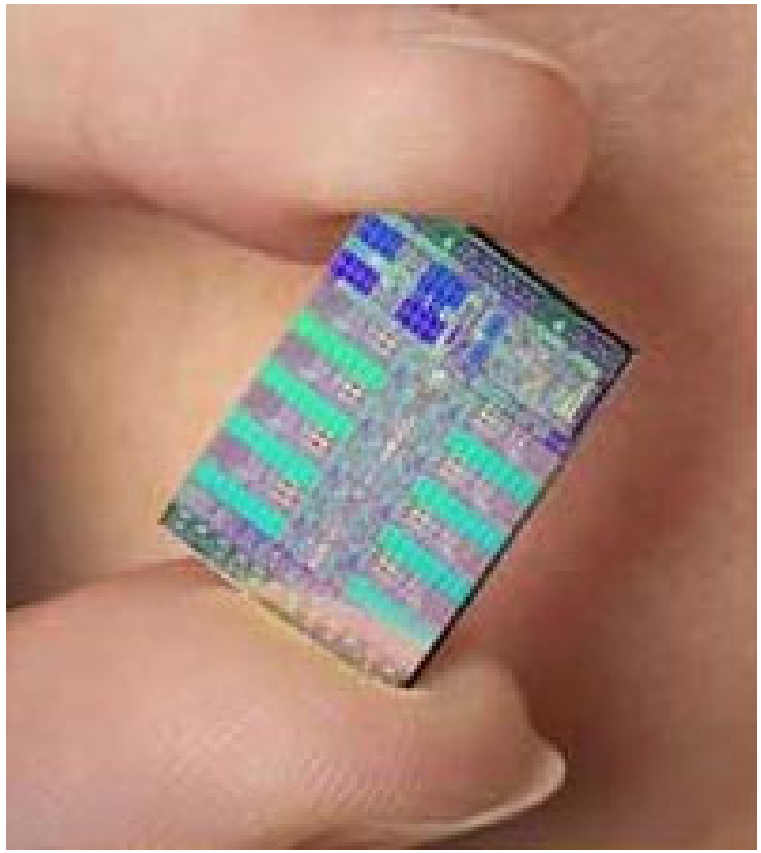
```
#pragma omp parallel for private(i,w) shared(N,a,b) reduction(+:sum)
for(i = 0; i < N; i++)
{
  b[i] = 2*a[i]+3;
  w = i*i;
  sum = sum + w*a[i];
}
```

Fortran:

```
!$omp parallel do private(i,w) shared(N,a,b) reduction(+:sum)
do i = 1, N
  b(i) = 2*a(i)+3
  w = (i-1)*(i-1)
  sum = sum + w*a(i)
end do
!$omp end parallel do
```

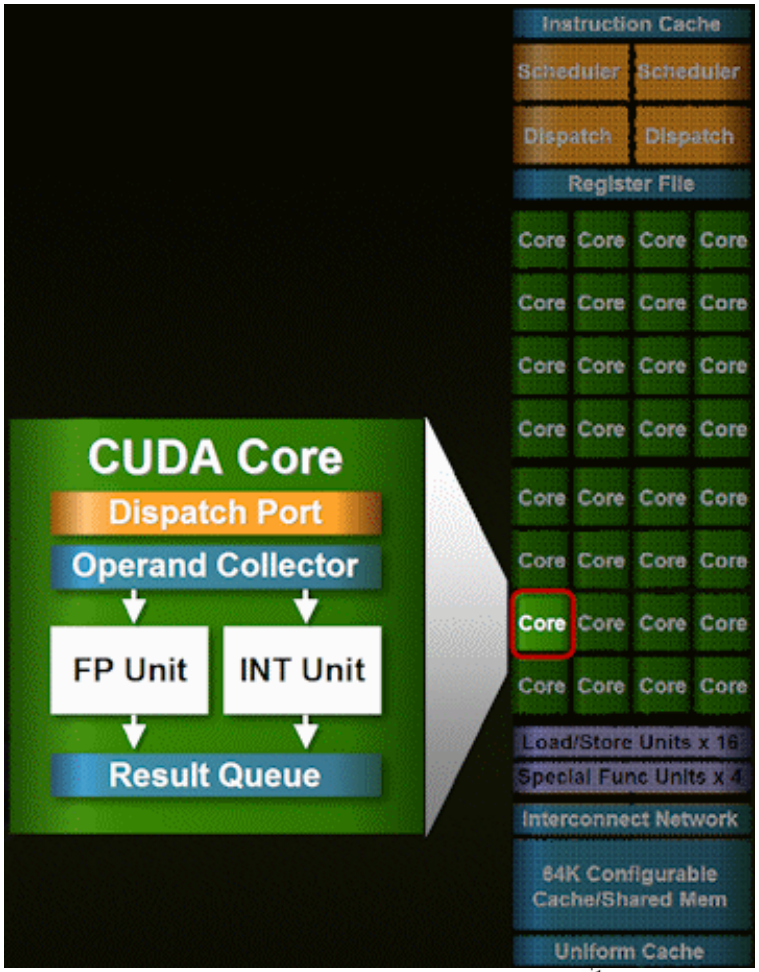
Accelerator Computing

- Started in 2001 with a Sony/Toshiba/IBM collaboration for the *Cell* processor.
- It has one "real" core (Power Processor Element) and 8 "additional" cores (Synergistic Processing Element) on a single chip that are linked together by a high speed bus (Element Interconnect Bus).
- It is used in Sony's Playstation, Roadrunner at Los Alamos National Laboratory (#19 on the top 500 in 2012) and other places.
- Newest player: Intel Xeon Phi used in TACC's Stampede.



GPU Computing

- Graphics cards with a Graphic Processing Unit (GPU) have a similar architecture with many cores.
- In a GPU each core has to perform the same operation (Single Instruction Multiple Data) i.e. the cores together work like a vector unit.
- The GPU cannot access the main memory of the CPU.
- Three out of the top 10 machines on the current top 500 contains a significant portion of GPUs.



GPU Computing Challenges

- Each core is small and slow. A GPU needs to use many cores to be as fast as a regular CPU.
  - Slower clock speed.
  - Less memory per core.
- Need to use even more cores to get good performance.
- But, if this is possible, then total speed is much larger than that of regular CPUs (of same total cost) and uses a lot less power.
- Memory management is a lot more complicated.
  - Data has to be copied from CPU main memory to GPU memory and back (slow).
  - Algorithms have to be modified to do as many calculations as possible on data on GPU before copying back and forth.

Framework Architecture Challenges

- Don't want to redesign framework and rewrite code for new architectures.
- The framework should isolate the programmer from architectural changes as much as possible.
- MPI clusters have been around for about 20 years.
- Now something new is coming (has already arrived).
  - Multicore CPUs (requires use of OpenMP, pthreads or similar).
  - Cell and GPU accelerators (requires use of CUDA, OpenCL or OpenACC).
  - Intel Xeon Phi accelerators currently only works with the Intel compilers.

Cactus Approach to Architecture Independence

- Separate physics code from computer science code.
- Each thorn "sees" only the small part of the overall problem that is relevant for its function (information hiding).
- Ideally each physics thorn would act on a single grid point at a time. However, that would have too much overhead.
- This externalizes parallelism, load balancing, data distribution and data sharing (*largest Cactus success*).
- This also significantly complicates programming (*largest Cactus problem*).
- Multicore and accelerator programming poses a distinct challenge for Cactus.
- Currently being addressed with macros, templates and automatic code generation (LoopControl, CaKernel and Kranc).

Scientific Programming Practices Summary

- Use a (or several) source code management system for code development.
- Keep track of software versions for each simulation (Formaline).
- Have test cases for each component to ensure correctness.
- Need portability and architecture independence or at least enough flexibility to make programs future-proof.