Applied Mathematics & Computational Science 312: High Performance Computing II

Wolfgang Bangerth, Texas A&M University

David Keyes, KAUST

Top matter

About myself:

- Associate professor at Texas A&M University, one of the global research partners of KAUST
- I'm what they call a “Computational Scientist”
- Will be here till November 2, in Building 1, room 4409
- Will teach this and the next two weeks
- Will be glad to discuss with you any semester project you may want to do with our software (deal.II)

Will post class material at

`http://www.math.tamu.edu/~bangerth/teaching.html`

The four pillars of HPC

HPC has four pillars:

- Computer architecture
  - Processor characteristics, memory access, networks
- Algorithms
  - Scaling with input size, concurrency properties
- Software
  - Software design and best practices, testing, documentation
  - Managing complexity
  - Scalability of software, robustness, correctness, validation, verification
- Applications
  - All fields of the sciences (physics, chemistry, biology) and engineering (Therefore “Computational Sciences and Engineering (CS&E)”)
  - Extremely wide variety, often very discipline specific
  - Difficult to have an overview

The four pillars of HPC

HPC has four pillars:

- Computer architecture

  Processor characteristics, memory access, networks

- Algorithms

  Scaling with input size, concurrency properties

- Software

  Software design and best practices, testing, documentation
  Managing complexity
  Scalability of software, robustness, correctness, validation, verification

- Applications

  All fields of the sciences (physics, chemistry, biology) and engineering
  (Therefore "Computational Sciences and Engineering (CS&E)")

  Extremely wide variety, often very discipline specific

  Difficult to have an overview

My field and the focus of the next 3 weeks

Applications in HPC

“Most” HPC applications can be grouped into the following categories:

- Computational chemistry
  - Uses ~60% of CPU cycles on typical university supercomputers
  - About understanding catalysts, structure, arrangement; materials
  - Often difficult to make scale
- Partial differential equations & friends
  - Uses ~25% of CPU cycles
  - Posterchild of HPC because it has many flashy applications
- Computational biology
  - Uses maybe 5-10% of CPU cycles; often embarrassingly parallel
  - Gene sequencing, proteomics
- N-body problems
  - Used in astronomy, dilute gases, similar methods used in scattering
- The “rest”
  - Quantum field theory, data mining, financial mathematics, ...

Applications in HPC

"Most" HPC applications can be grouped into the following categories:

- Computational chemistry
  - Uses ~60% of CPU cycles on typical university supercomputers
  - About understanding catalysts, structure, arrangement; materials
  - Often difficult to make scale
- Partial differential equations & friends
  - Uses ~25% of CPU cycles
  - Posterchild of HPC because it has many flashy applications
- Computational biology
  - Uses maybe 5-10% of CPU cycles; often embarrassingly parallel
  - Gene sequencing, proteomics
- N-body problems
  - Used in astronomy, dilute gases, similar methods used in scattering
- The "rest"
  - Quantum field theory, ...

Looking forward

Plan for the next 3 weeks:

- Examples of PDEs in HPC
- A brief overview of the typical workflow in HPC for PDEs
- A very brief introduction to the finite element method
- Identifying mathematical concepts in the FEM that we need to represent in software

- Installing an open source FEM package, deal.II
- Going through and playing with some simple examples
- A more interesting example: the driven cavity
- Discussion of a homework project

- Beyond solving PDEs: optimization, inverse problems
- The role of software in HPC

Partial differential equations (PDEs) in HPC

Many well known applications in HPC are described by PDEs:

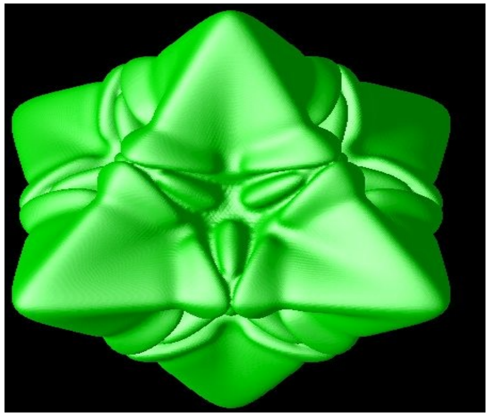
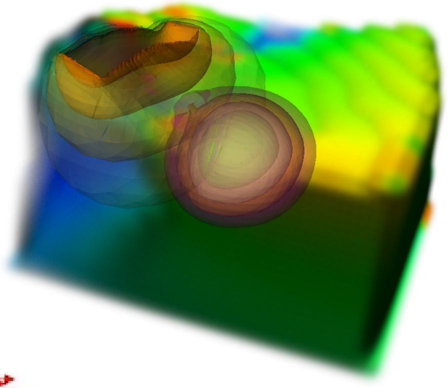
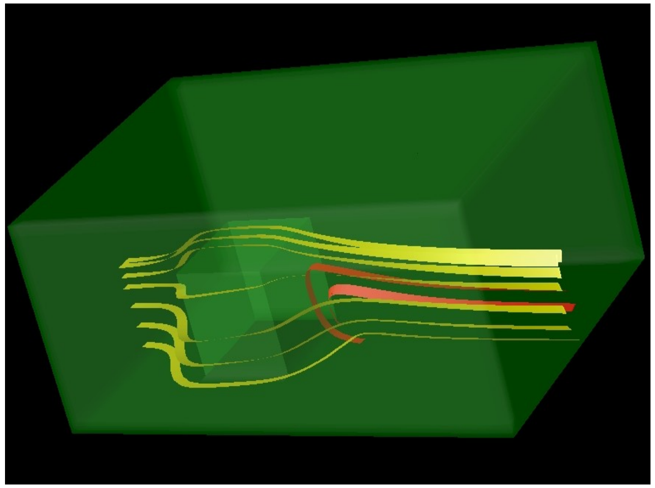
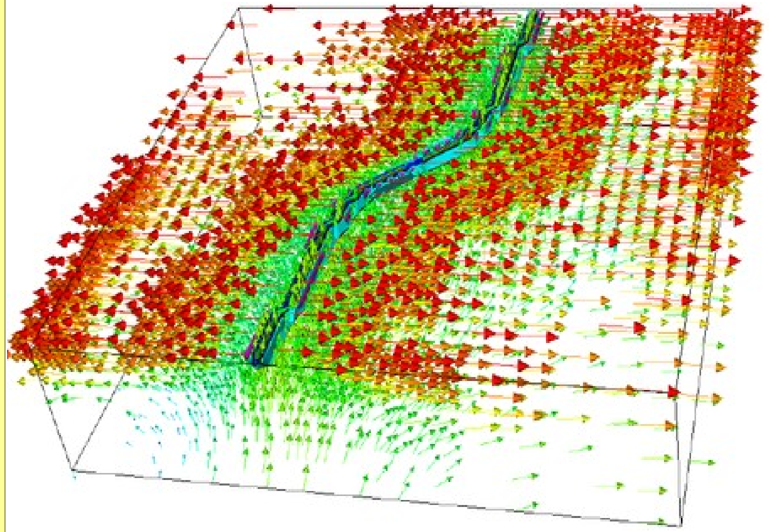
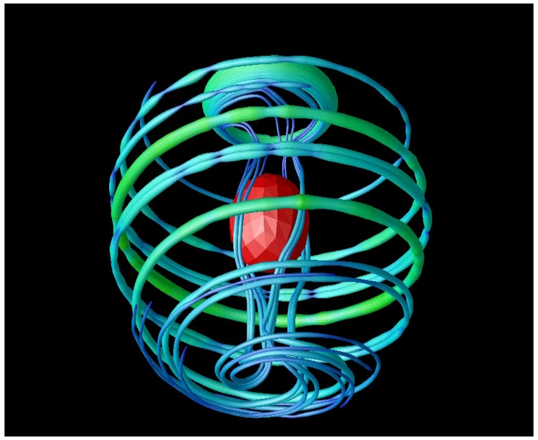
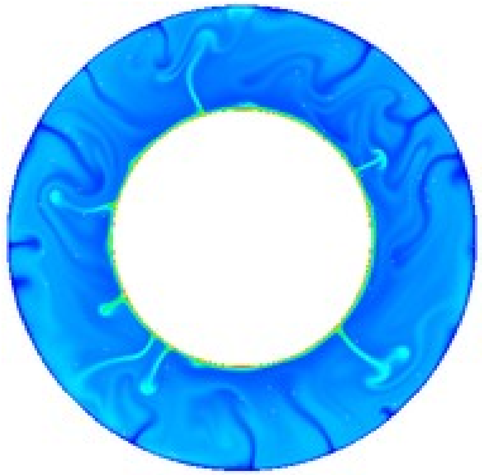
- Aerodynamics (cars, planes, windmills, ...)
- Weather and climate
- Statics and dynamics
- Earthquake modeling and seismic imaging
- Modeling of nuclear fission and fusion
- Biomedical imaging
- Combustion, explosions
- ...

Characteristics:

- Can be grouped into a number of model categories
- Can often be made to scale to 1000s or more processors
- Many excellent and efficient algorithms known
- Often complex software: 100,000s of line of code

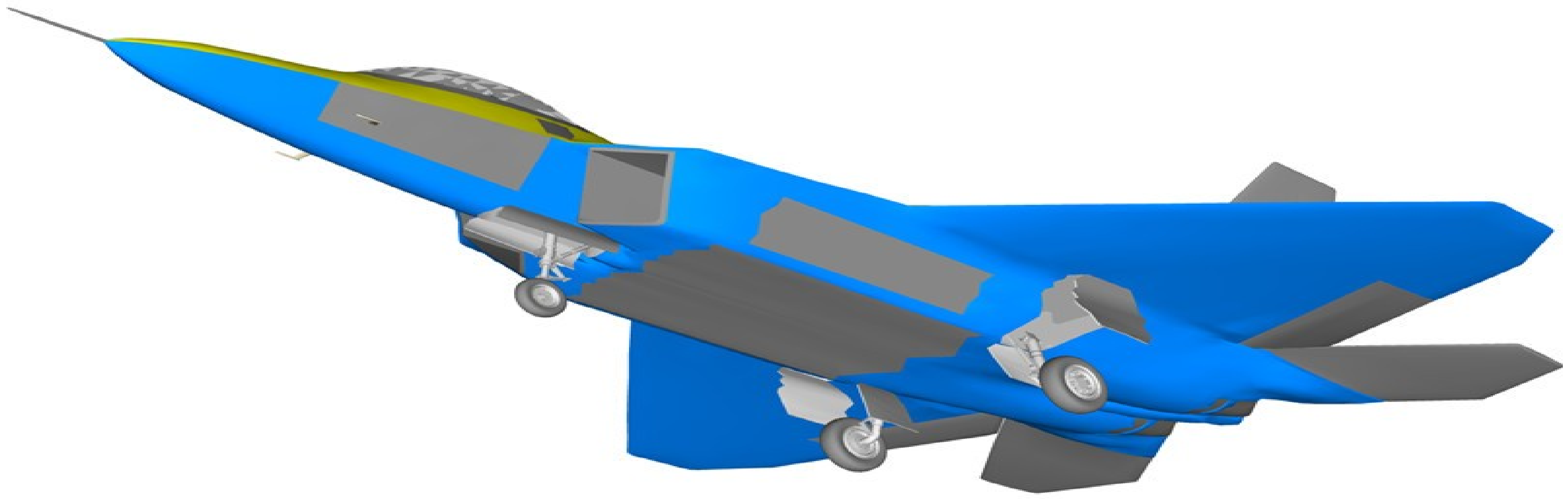
Partial differential equations (PDEs) in HPC

Examples from a wide variety of fields in my own work:

Workflow for HPC in PDEs

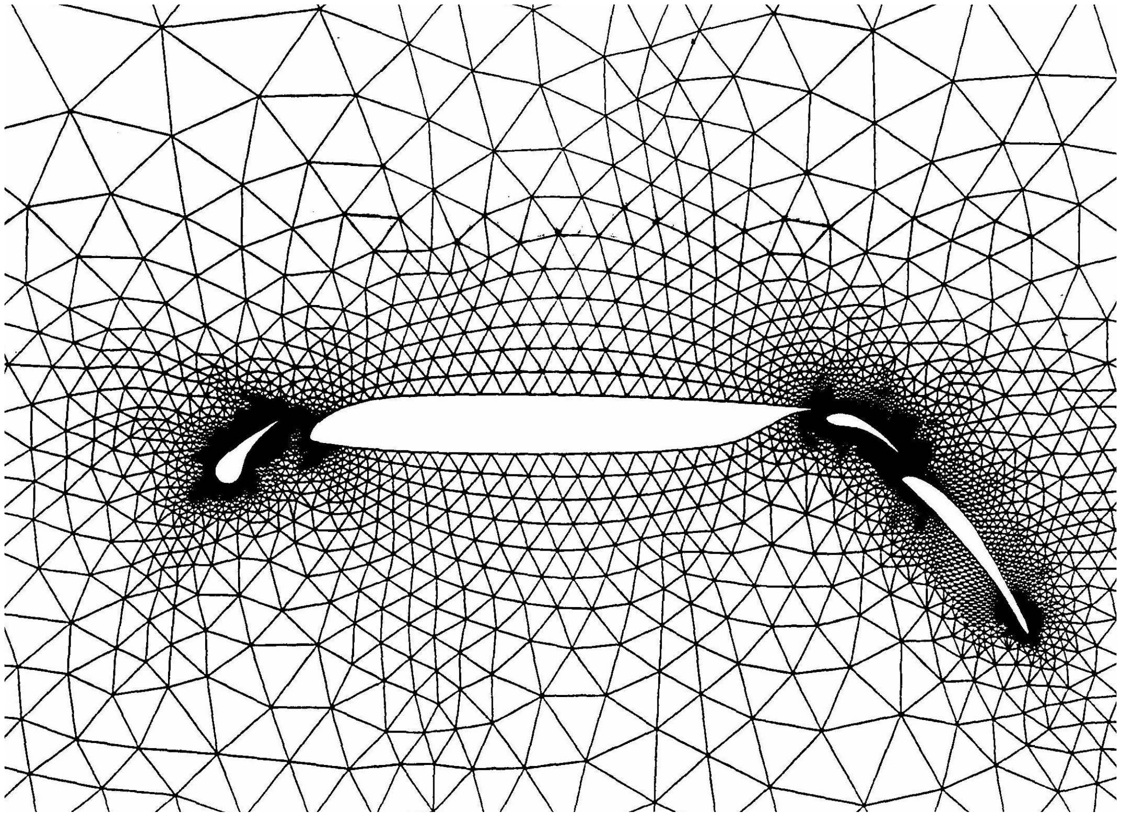
Step 1: Identify geometry and details of the model



May involve tens of thousands of pieces, very labor intensive, interface to designers and to manufacturing

Workflow for HPC in PDEs

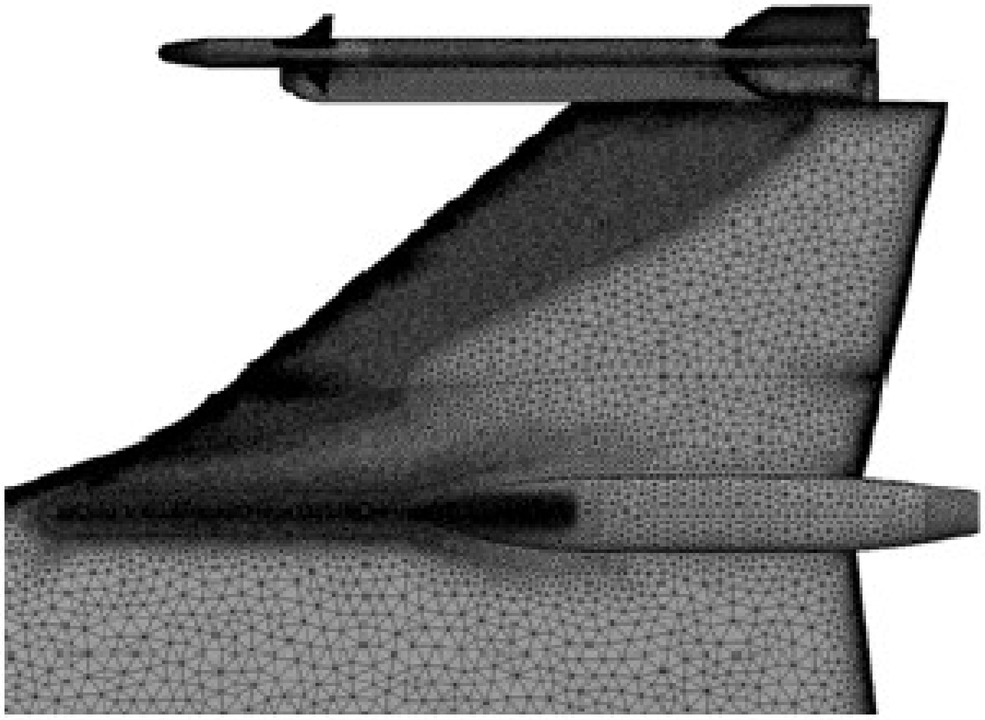
Step 2: Mesh generation and maybe partitioning (preprocessing)



May involve 10s of millions or more of cells; requires lots of memory; very difficult to parallelize

Workflow for HPC in PDEs

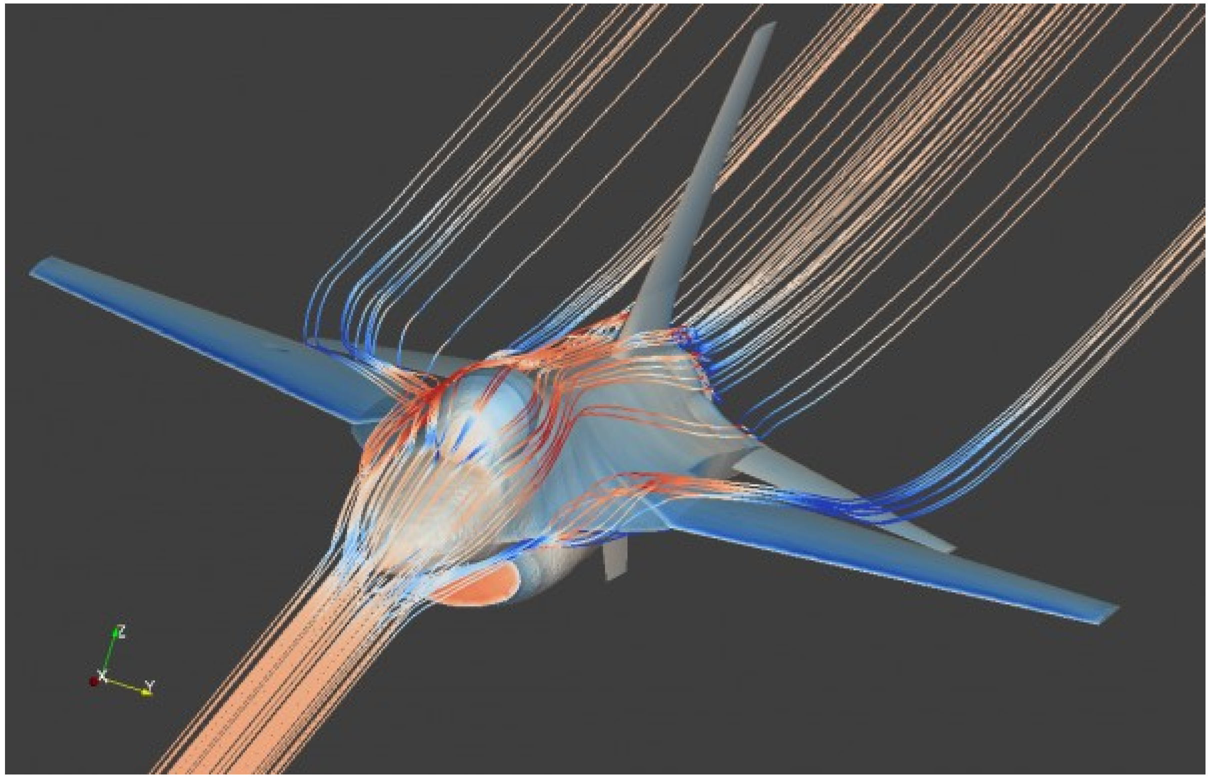
Step 2: Mesh generation and maybe partitioning (preprocessing)



May involve 10s of millions or more of cells; requires lots of memory; very difficult to parallelize

Workflow for HPC in PDEs

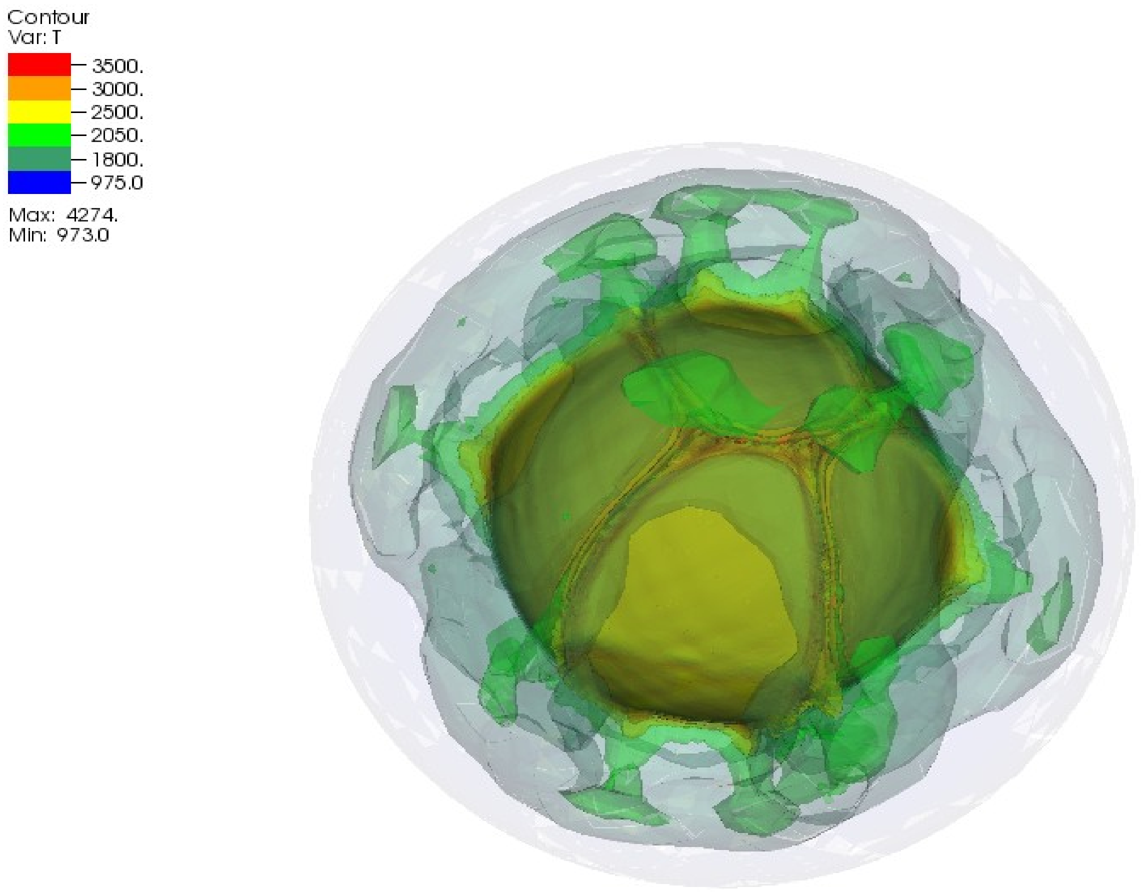
Step 3: Solve model on this mesh using finite elements, finite volumes, finite differences, ...



Involves some of the biggest computations ever done, 10,000s of processors, millions of CPU hours, wide variety of algorithms

Workflow for HPC in PDEs

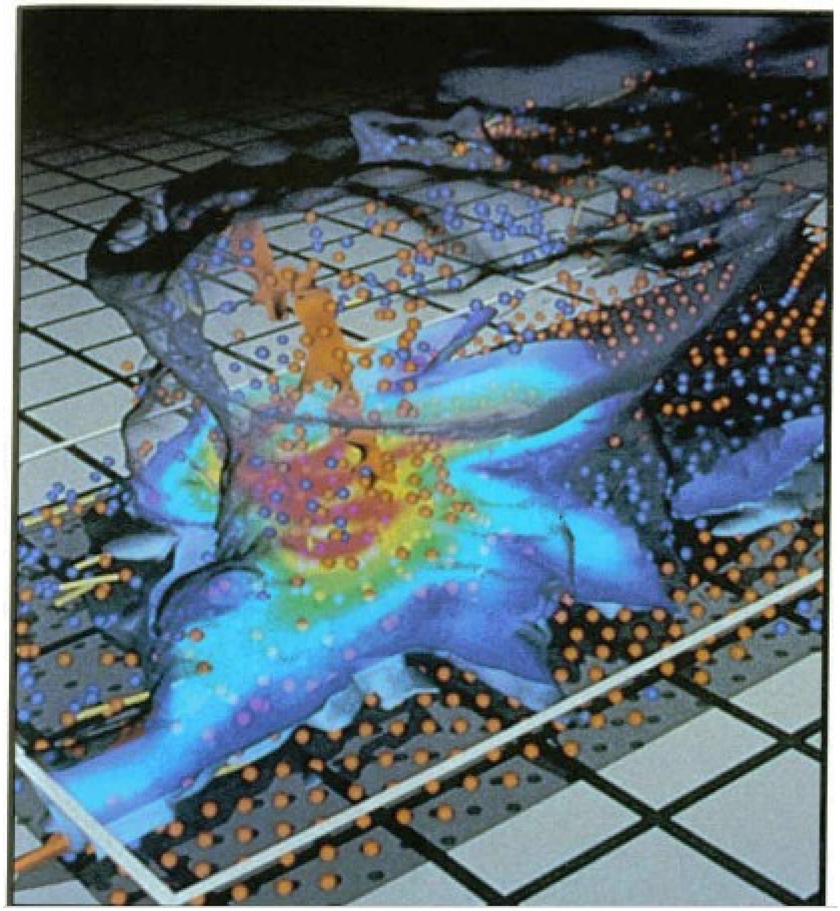
Step 4: Visualization to learn from the numerical results



Can be done in parallel, difficulty is amounts of data

Workflow for HPC in PDEs

Step 4: Visualization to learn from the numerical results



Can be done in parallel, difficulty is often amount of data

Workflow for HPC in PDEs

Step 5: Repeat

- To improve on the design
- To investigate different conditions (speed, altitude, angle of attack, ...)
- To vary physical parameters that may not be known exactly
- To vary parameters of the numerical model (e.g. mesh size)
- To improve match with experiments

Workflow for HPC in PDEs

Each of these steps...

- Identify geometry and details of the model
- Preprocess: Mesh generation
- Solve problem with FEM/FVM/FDM
- Postprocess: Visualize
- Repeat

...needs software that requires:

- domain knowledge
- knowledge of the math. description of the problem
- knowledge of algorithm design
- knowledge of software design and management

We will come back to some of these issues later.

A prototypical example: FEM simulations

Next steps:

- Derive the finite element method (on the whiteboard)
- Identify things in the math that we need to represent in FEM codes
- Identify where in these concepts HPC is involved
- Download and install the software from

  `http://www.dealii.org/`

  onto your Mac or linux laptops, university workstation or supercomputer account
- Go through some of the tutorial programs

Components of a finite element program

In our derivation of the finite element method we have identified the following components that we need to represent:

- FE shape functions defined on a reference cell
- The mesh
- The DoFHandler enumeration of degrees of freedom
- The mapping from reference cell to each mesh cell
- Quadrature rules to approximate integrals
- Data structures to store linear systems
- Solvers for linear systems
- Postprocessing algorithms

Components of a finite element program

Memory requirements: (Blue: O(1). Red: O(N))

- FE shape functions
- Mesh
- DoFHandler
- Mapping
- Quadrature rules
- Linear systems
- Solvers for linear systems
- Postprocessing algorithms

Note: Because of their memory requirements, no processor in a parallel program can store every piece of the red components (i.e., they must be *distributed*). On the other hand, blue components can be replicated.

Software issues in HPC

Ultimately, HPC is about *applications*, not just algorithms and their analysis.

Thus, we need to consider the issue of *software that implements* these applications:

- How complex is the software?
- How do we write software? Are there tools?
- How do we verify the correctness of the software?
- How do we validate the correctness of the model?

- Testing
- Documentation
- Social issues

Complexity of software

Many HPC applications are *several orders of magnitude* larger than everything you have probably ever seen!

For example, a crude measure of complexity is the number of lines of code in a package:

- Deal.II has 550k
- PETSc has 500k
- Trilinos has 3.1M

At this scale, software development does not work the same as for small projects:

- No single person has a global overview
- There are many years of work in such packages
- No person can remember even the code they wrote

Complexity of software

The only way to deal with the complexity of such software is to:

- *Modularize:* Different people are responsible for different parts of the project.
- *Define interfaces:* Only a small fraction of functions in a module is available to other modules
- *Document:* For users, for developers, for authors, and at different levels

- *Test, test, test*

How do we write software

Successful software must follow *the prime directive of software*:

- Developer time is the single most scarce resource!

As a consequence (part 1):

- Do not reinvent the wheel: use what others have already implemented (even if it's slower)
- Use the best tools (IDEs, graphical debuggers, graphical profilers, version control systems...)
- Do not make yourself the bottleneck (e.g. by not writing documentation)

- Delegate. You can't do it all.

How do we write software

Successful software must follow *the prime directive of software*:

- Developer time is the single most scarce resource!

As a consequence (part 2):

- Re-use code, don't duplicate
- Use strategies to *avoid* introducing bugs

- Test, test, test:
  - The earlier a bug is detected the easier it is to find
  - Even good programmers spend more time debugging code than writing it in the first place

Verification & validation (V&V): Verification

Verification refers to the process of ensuring that the software solves the problem it is supposed to solve:

“The program solves the problem correctly”

A common strategy to achieve this is to...

Verification & validation (V&V): Verification

Verification refers to the process of ensuring that the software solves the problem it is supposed to solve:

"The program solves the problem correctly"

A common strategy to achieve this is to *test test test:*

- *Unit tests* verify that a function/class does what it is supposed to do (assuming that correct result is known)
- *Integration tests* verify a whole algorithm (e.g. using what is known as the Method of Manufactured Solutions)
- Write *regression tests* that verify that the output of a program does not change over time

Software that is not tested does not produce the correct results!

(Note that I say "does not", and not "may not"!)

Verification & validation (V&V): Validation

Validation refers to the process of ensuring that the software solves a formulation that accurately represents the application:

“The program solves the correct problem”

The details of this go beyond this class.

Testing

Let me repeat the fundamental truth about software with more than a few 100 lines of code:

Software that is not tested does not produce the correct results!

No software that does not run lots of automatic tests can be good/usable.

As just one example:

- Deal.II runs ~2300 tests after every single change
- This takes ~10 CPU hours every time
- The test suite has another 250,000 lines of code.

Documentation

Documentation serves different purposes:

- It spells out to the developer what the *implementation* of a function/class is supposed to do (it's a *contract*)
- It tells a user what a function does
- It must come at different levels (e.g. functions, classes, modules, tutorial programs)

Also:

- Even in small projects, it reminds the author what she had in mind with a function after some time
- It avoids that everyone has to ask the developer for information (bottleneck!)
- Document the history of a code by using a version control system

Social issues

Most HPC software is a collaborative effort. Some of the most difficult aspects in HPC are of social nature:

- Can I modify this code?
- X just modified the code but didn't update the documentation and didn't write a test!
- Y1 has written a great piece of code but it doesn't conform to our coding style and he's unwilling to adjust it.
- Y2 seems clever but still has to learn. How do I interest her to collaborate without accepting subpar code?
- Z agreed to fix this bug 3 weeks ago but nothing has happened.
- M never replies to emails with questions about his code.