USI-CSCS Summer School 2013: Foundation of the Third Pillar of Science

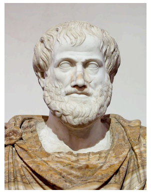
William Sawyer, Themis Athanassiadou

CSCS-USI Summer School on Computer Simulations in Science and Engineering, July 8-19 2013, Universita' Svizzera Italiana, Lugano, Switzerland

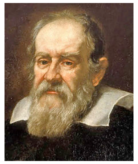
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

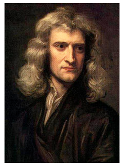
CSCS
Swiss National Supercomputing Centre

Premise: 3 pillars of 21. century scientific method

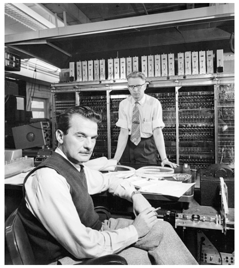


Theory (since antiquity)





combined with experiment (since Galilei & Newton)



and simulation
(since Metropolis, Teller,
von Neumann, Fermi, ... 1940s)

Excellence in Science requires excellence in all three areas: theory, experiment, and simulations

CSCS/USI Summer School

Goal: Introduce you to the *third pillar of science* through several key aspects of computer simulation:

- Basic concepts and tools
- "Best practices" for simulation software
- High performance computing (HPC)
  - parallel processing (message passing, multithreading)
  - analyzing performance
  - libraries: utilizing third-party software
- A scientific use case: the Tsunami model

Organizers: CSCS, USI/ICS and the Foundations in Mathematics and Informatics for Computer Simulations in Science and Engineering (FoMICS) program

Summer School 2013 AGENDA

- July 8-9: HPC Software Engineering Best Practices
- July 10: Parallel programming introduction
- July 10 evening: summer school dinner
- July 11-12: Parallel programming with the Tsunami model
- July 13 or 14: hike (weather dependent)
- July 15-16: Parallel programming with Tsunami model continued
- July 16-18: Performance analysis
- July 17 evening: beach party (weather dependent)
- July 19: Numerical libraries

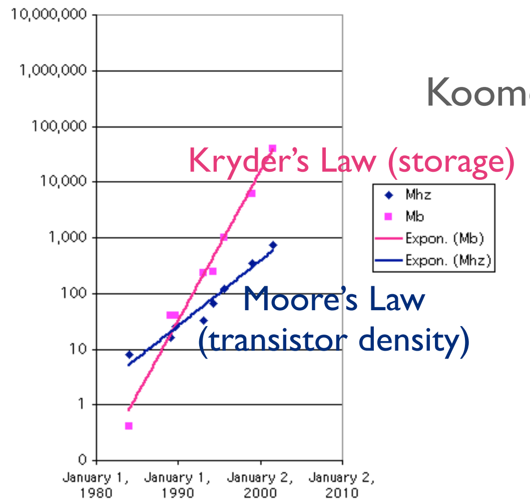
Beware: the summer school design is entirely new this year!

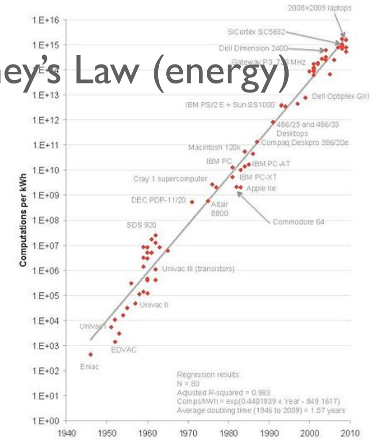
Some basic HPC concepts

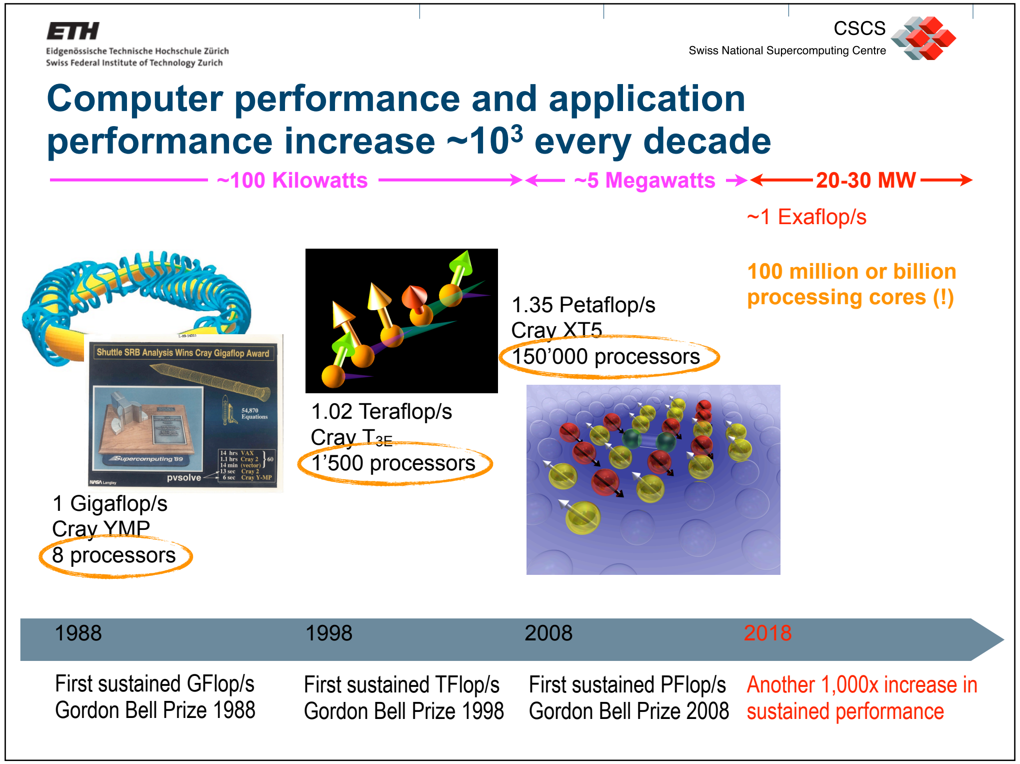
```
     934,569.299814557
x          52.827419489135904
----------------------------
= 49,370,884.442971624253823
+           4.20349729193958
----------------------------
= 49,370,888.64646892
```

- *Floating-point operation (Flop)*
  - ✳ Flop per second (Flop/s)
- *Byte:* 8 binary digits 0/1
  - ✳ Bytes per second (B/s, KB/s, MB/s, GB/s, ...)
- *Time to solution:* wall clock time (s.) to find answer
- *Energy to solution:* energy (Joules, KWh) to answer
- *Programming language:* language used describe calculation (generally at high level of abstraction)
- *Software engineering:* systematic approach for design/ development/operation/maintenance of software

Progress of computer power







Moore's gap: 10M processors is probably too optimistic, 100M processors with billions of threads more realistic
Opportunities for disruptive technologies, or maybe even a revolution?

Petaflop/s = 10^{15} 64-bit floating point operations / sec.

which takes more energy?

64-bit floating-point fused multiply add **or** moving three 64-bit operands 20 mm across the die

```
     934,569.299814557
x          52.827419489135904
----------------------------
= 49,370,884.442971624253823
+           4.20349729193958
----------------------------
= 49,370,888.64646892
```

this takes over 3x the energy!

loading the data from off chip takes > 10x more yet

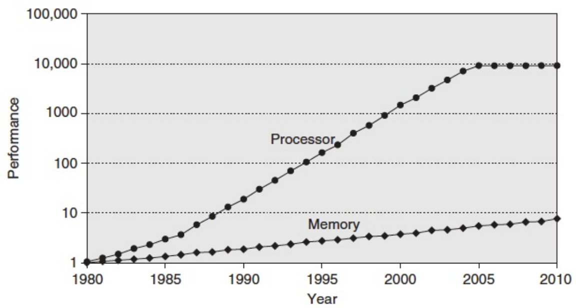
source: Steve Scott, Cray Inc.

moving data is expensive – exploiting data locality is critical to energy efficiency

If we care about energy consumption, we have to worry about these and other physical considerations of the computation – but where is the separation of concerns?

Current challenges

- CPU frequency not increasing
  ➡ energy consumption
- Memory access times have lagged behind CPU performance

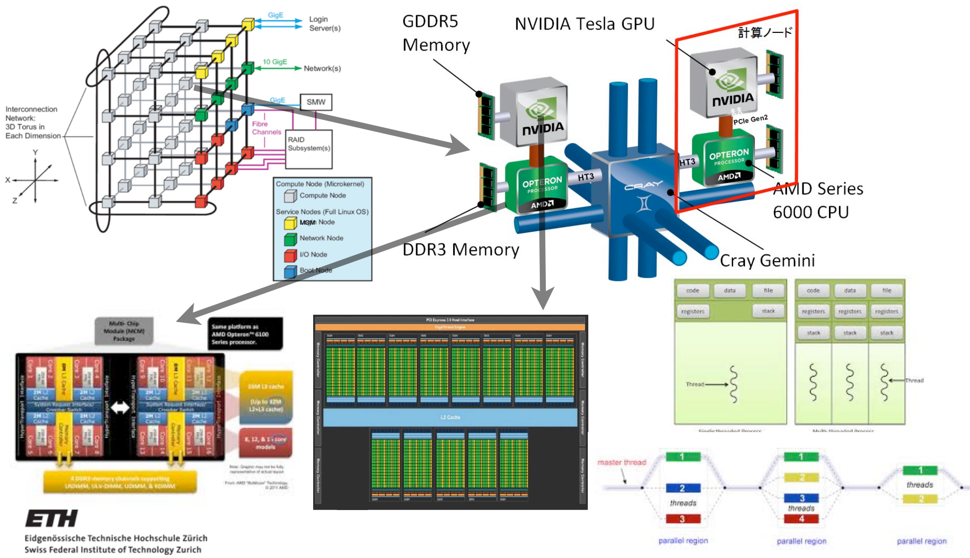


Solutions?

- Place multiple "cores" on same die within a "socket"
- Access data which is "nearby" ("caches", local memory, ...)

➡ *requires the latest HPC programming techniques (and possibly some not invented yet...)*

State-of the-art HPC platform



ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Important CSCS access information

- See: *Programming environment for Practicals*
- Your account: `courseXX` (XX & pwd supplied separately)
- Access HPC platforms through CSCS front-end `ela.cscs.ch`
  `ssh -Y First courseXX@ela.cscs.ch`
- Course platform: Cray XK7 (previous slide) `todi.cscs.ch`
  `ssh -Y todi`
- For most of course: switch to the GNU programming env.
  `module swap PrgEnv-cray PrgEnv-gnu`
- Compilers: C++: `CC` C: `cc` Fortran: `ftn`
- Allocating one node for interactive use:
  `salloc --res=sschool -N 1 --time=01:00:00`
- Run a parallel job: `aprun -n 8 ./myprogram`

Course information on-line

- Software management site: http://www.github.com
  - Repository: `SummerSchool2013`
  - GitHub User: `fomics`
  - Presentations: SummerSchool2013/PRESENTATIONS
  - Course information: SummerSchool2013/wiki
- CSCS on-line (http://www.cscs.ch)
  - User portal: http://user.cscs.ch/
- Course directory:
  - `ela:/project/csstaff/courses/CSCS_USI_School`

Introduction to Software Engineering for Computer Simulations

Dr. William Sawyer

CSCS-USI Summer School on Computer Simulations in Science and Engineering, July 8-19 2013, Universita' Svizzera Italiana, Lugano, Switzerland

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

CSCS
Swiss National Supercomputing Centre

Scientific simulation programming

In our experience:

- simulation software is written by scientists in the field
- these scientists are not trained in software engineering
- software often poorly designed/documented/implemented
- extensions often implemented by graduate students in the same field, again minimal training in software engineering. They adopt same style as the original code
- this development strategy is continued until (1) the software dies out, or (2) there is a major re-engineering effort

Poll: how many of you feel this applies to your project?

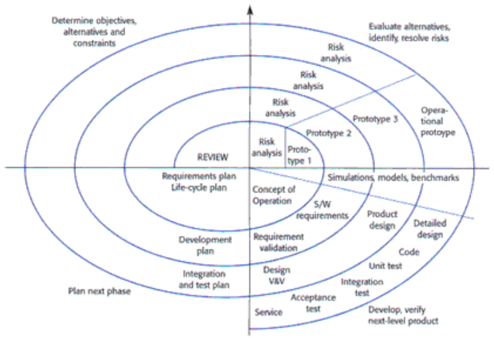
Software Engineering

[ACM Definition] *Software Engineering (SE) is concerned with developing and maintain software systems that:*

- *satisfy all the requirements that customers have defined*
- *behave reliably and efficiently*
- *are affordable*

SE introduces concepts like:

- *Requirements definition*
- *Development planning*
- *Prototyping*
- *Risk analysis*
- *Implementation / Validation*
- *Integration and acceptance*



The case for Scientific Software Engineering

Traditionally, SE processes and techniques have not employed widely for scientific simulation software. *Why not?*

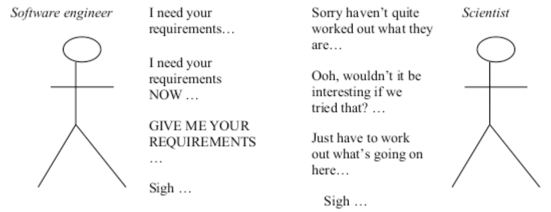
- Software is only a means to scientific results
- Perception: scientists and software engineers from 2 separate cultures
- Skilled scientists believe they can maintain overview of large software

Scientific Software Engineering can no longer be avoided:

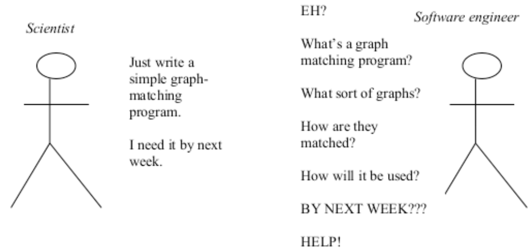
- scientific models can be very large (> 1M lines of code)
- large numbers of concurrent developers
- programming difficulty of emerging hardware is increasing: *co-design* of scientists with hardware and software specialists is needed
- scientific computing can require huge investment (development and operating costs)

Two cultures?

Software engineer's point of view:



Software engineer's point of view:



Source: Judith Segal

Useful links

Interestingly: little applicable web material available

- *Software Engineering for Scientific Computing* (Phil Colella, UC Berkeley)

  The Seven Dwarfs
- *Scientists and software engineers: A Tale of Two Cultures* (Judith Segal, Open Univ.)
- *Computational Science and Engineering Department* (Chris Greenough, STFC)
- *Repeating Our Mistakes? Software Engineering for HPC* (David Bernholdt, ORNL):

  "... long standing disconnect between computational science and software engineering"
- Custom state-of-the-art scientific software (Painter, Matthews, LANL):

  "Writing code isn't the problem. Understanding the problem is the problem."

Conclusions of David Bernholdt

- There is a long-standing disconnect between computational science and software engineering
- More software engineering research is needed to support HPC and computational science
  ➡Currently spotty, many gaps, lacks critical mass
- Are we repeating our mistakes?
  ➡Yes
- Do we have to continue repeating them?
  ➡No

David Bernholdt, Fall Creek Falls Conf., 25-27 Oct. 2010

Goals for this component

The concept of this course stems from discussions with David Ham (Imperial College) at Infrastructure for European Network for Earth Science workshop

If software engineers cannot be employed, then at least SE-knowledgable scientists!

- Get you acquainted with common SE tools used in scientific computing
- Promote “best practices” and recipes for good scientific software
  - ✳Clean programming style including documentation
  - ✳Software configuration management
  - ✳Validation and verification (automated)
- Explain legal issues relevant for scientific programmers (particularly grad students)
- Avoid boring business and project management aspects

Your input is needed to refine the course for future generations

Under development

Scenario 1: simultaneous development

A large scientific code being developed by geographically distributed team

- individuals usual work on separate parts, but there is often overlap
- different components need to be "welded" together
- other developers not concerned about "details" of your implementation

Questions:

- How to make sure that changes to code base do not conflict?
- How does an individual develop/test without affecting others?
- How is the software merged into one?
- How are the individual components validated? the entire code?

Scenario 2: extensions to existing software

Your professor gives you the code from his dissertation and tells you to make extensions to it

- It's poorly written and documented
- It is monolithic, all low-level details visible
- It performs poorly, or and runs "sequentially" on one core

Questions:

- How do you re-engineer the code to "clean it up"?
  - ➡Code needs: modularity, data encapsulation, proper documentation
  - ➡Software configuration management
  - ➡Validation
- How to track the record of all the improvements to the code?
- How to make it run more efficiently on the latest architectures?

Software Best Practices

The objectives of software development should be to make

- code simple to use by those who use it, but don't need to know the details
- your component easy to manage by those who come after you
- it as easy as possible to locate problems / bugs in the code
- it as easy as possible to integrate/merge development into a single system

Best practices aid toward these objectives

- Code broken up into modules with class hierarchies and clear interfaces; generally a single developer maintains one module
- Clear programming style: adherence to guidelines, expressive documentation, use of documentation system
- ...

Software Configuration Management

SCM is the *task of tracking and controlling changes in the software*

- Configuration Control: *implementing a controlled change process*
- Process Management: *adherence to organization's development process*
- Build Management: *managing the process and tools used to build software*
- Defect Tracking: *making sure every defect is traceable back to the source*
- **Data Provenance:** sometimes called “lineage” or “pedigree,” is the description of the origins of a piece of data and the process by which it arrived in a database
- **Revision Control**
  - ➡ *management of changes to documents, computer programs, and other collections of information*
  - ➡ *embedded into numerous document systems, e.g., Excel, Word, etc.*
  - ➡ *Version Control Systems (VCS) for large software projects: generally stand-alone applications*
    - *can be centralized: client-server, developers check into a central server*
    - *or distributed: each developer has a local repository, can share changes globally*

A few words about *data provenance*

Data provenance is really mentioned in SCM. Why not?

- In many ways it goes beyond SCM
- In most software fields, the software is the product, but
- In science, the resulting *data are the product,* i.e., the software is the means
- Scientific professionalism required data be *reproducible for years,* but
  - data depend on the computer architecture, program, algorithms, operating system, compiler brand and version
  - some fields (e.g. climatology) even require bit-for-bit reproducibility

Data provenance is a kind of metadata: is specifies the derivation history of a data product from its original sources. Its precise description depends on the field. In science, it ensures that data are reliable and reproducible, and therefore usable in other scientific work

Agenda Software Engineering

Today:

- Introduction to Software Configuration Management (David)
- C++ and its Best Practices in the Nutshell (Jean)
- Introduction to the Shallow Water Model "Tsunami" (Will)
- Using GIT for version control + LAB (John)

Tomorrow:

- Building code with Cmake + LAB (John)
- Verification techniques and manufactured solutions (David)
- Legal Issues with scientific code development (David)
- LAB: team development of Tsunami

Brief Introduction to the Doxygen Documentation System

Dr. William Sawyer, staff TU Munich

CSCS-USI Summer School on Computer Simulations in Science and Engineering, July 8-19 2013, Universita' Svizzera Italiana, Lugano, Switzerland

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

CSCS
Swiss National Supercomputing Centre

What is doxygen?

Doxygen is a documentation system for C++, C, Java, Objective-C, Fortran and other languages. It helps in

- Generating on-line documentation or offline reference manual from documented source files
- Extracting the code structure and visualising the relations between various elements by means of include dependency graphs, inheritance diagrams, and collaboration diagrams, which are all generated automatically

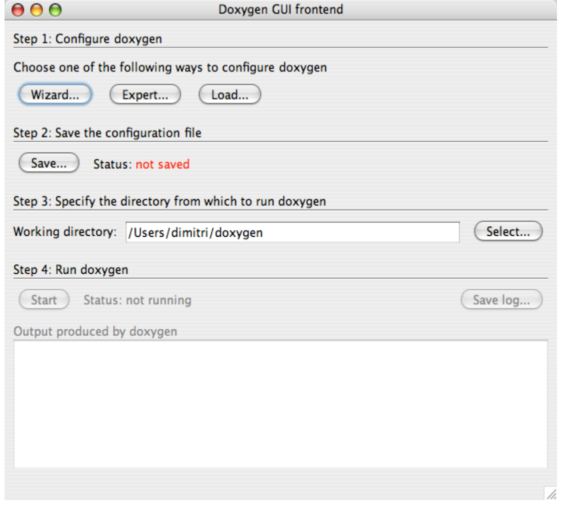
Doxygen steps

- Create a configuration file
- Document the code
- Run doxygen

Creating a configuration file

- `Doxygen` determines settings from configuration file
- `Doxywizard` is a GUI front-end for configuring and running doxygen

Doxywizard



Ways to configure and run doxygen

- Wizard
- Expert
- Load

Ways of documenting code

- Document blocks or lines
- Document members
- Structural commands
- Create lists

Documenting a block

Specify comment blocks:

- Special documentation blocks: comment blocks with additional markings
- Brief and detailed descriptions (see later)

C/C++ syntax

- Extra * or ! For Comment Blocks

e.g.

```
/**
 * ... text ...
 */
```

or

```
/*!
 * ... text ...
 */
```

Documenting lines

Specify comment lines:

- For comment lines: addition slash or an exclamation point

C/C++ syntax

```
///
/// ... text ...
///
```

or

```
//!
//!... text ...
//!
```

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

CSCS
Swiss National Supercomputing Centre

Brief description

Specify a brief description of block

- `\brief` command

C/C++ syntax

```
/*! \brief Brief description.
 *         Brief description continued.
 *
 *  Detailed description starts here.
 */
```

Document members

- Documentation after member specification in source file
- Use ‘<‘ to indicate that member is located in front of descriptor

C/C++ syntax

```
int var ; /*!< Description   */
```

or Brief Description

```
int var ; //!< Brief Description
```

Creating lists

- Bulleted Lists: column-aligned minus sign '–'
- Numbered Lists: column-aligned minus sign followed by hash '–#'
- Nesting level is maintained according to column alignment
- HTML commands: can be used inside comment

C/C++ syntax

```
/*!
 * List of events
 *    -Event1
 *       -#Subevent1
 *       -#Subevent2\n
 *         subevent2 cont.
 *       -#Subevent3
 *    -Event2
 *    -Event3
 */
```

Structural commands

- Starts with '\' or '@'
- Main commands

  `\author` {list of authors}

  `\brief` {brief description}

  `\date` {date description}

  `\file` [<name>]

  `\fn` (function name) used if comment block is not placed

  `\param` <parameter-name> {parameter description}

  `\return` {return value description}

Visual enhancement commands

\a - Special font

\b - bold

\c - Typewriter Font

\arg - Simple bulleted list, not nested

\e - Italics

\em - Emphasize word and in Italics