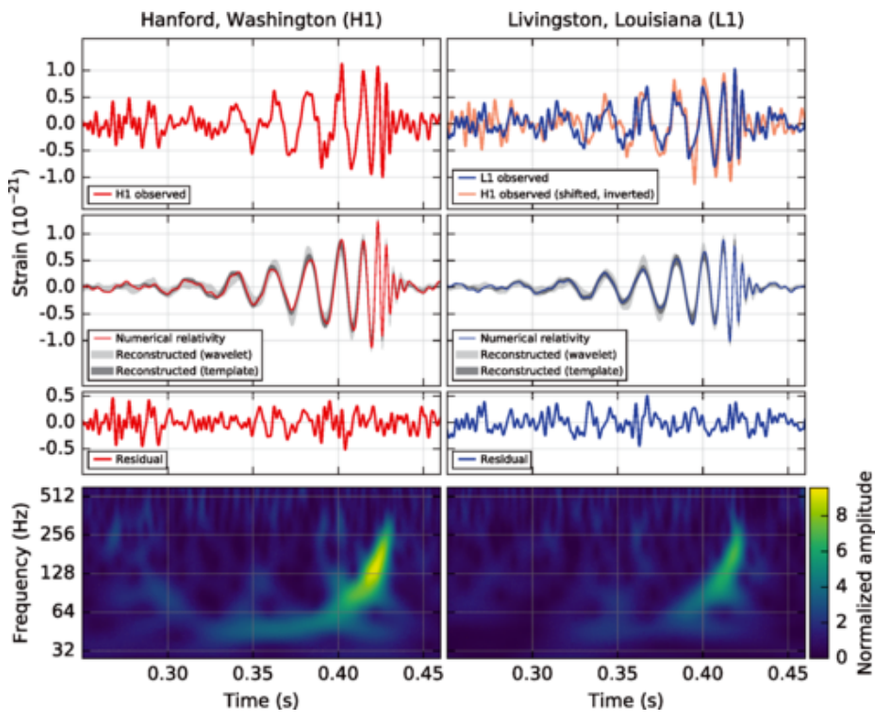
MS141: Introduction to Scientific Computing

for Scientists and Engineers



Instructor: Marco Bernardi

MS141: A Practice Oriented Course

Emphasis on practical knowledge and hands-on approach

- We will learn a bit about coding (and Python), a bit about numerical methods to solve problems arising in engineering and the physical sciences, and a bit about the specific project topics you'll choose

This is not a class on software engineering

- The goal is not to replace formal courses in computer science. In scientific computing, the goal is to know **just enough** about software design to use computers for research proficiently

This is not a class on numerical analysis or a rigorous math course

- We will emphasize intuitive understanding, and provide resources for learning more about numerical methods

This course introduces you to research in scientific computing. Key to research is **independence** (for students, with guidance from a mentor). The research projects will emphasize this goal

Scientific Computing

Solving domain-specific problems using computers

Experiment, theory… and **computation** (3rd paradigm)

Who

- Expert in an area of science or engineering with basic or intermediate computer science knowledge (esp. software design)

Why

- Solve difficult physical problems and equations
- Simulate the behavior of physical systems
- Formulate new approaches to solve a difficult problem (e.g., AI)
- Visualize and analyze large data sets

What for

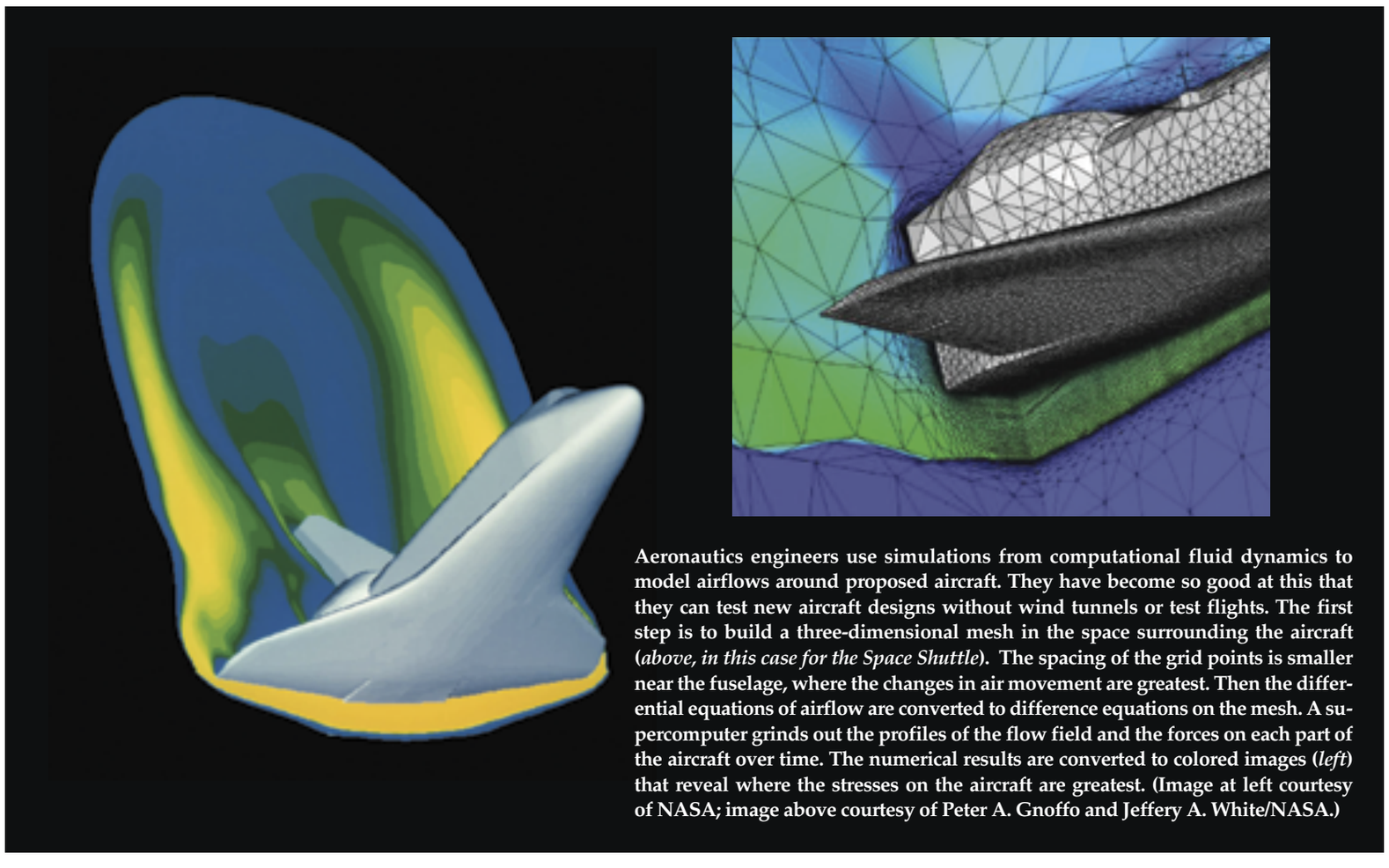
- Any field of science and engineering
- Research as undergraduate / graduate student or as a professional

Motivation & Course “Manifesto"

Computational Thinking in Science

The computer revolution has profoundly affected how we think about science, experimentation, and research.

Peter J. Denning



Aeronautics engineers use simulations from computational fluid dynamics to model airflows around proposed aircraft. They have become so good at this that they can test new aircraft designs without wind tunnels or test flights. The first step is to build a three-dimensional mesh in the space surrounding the aircraft (*above, in this case for the Space Shuttle*). The spacing of the grid points is smaller near the fuselage, where the changes in air movement are greatest. Then the differential equations of airflow are converted to difference equations on the mesh. A supercomputer grinds out the profiles of the flow field and the forces on each part of the aircraft over time. The numerical results are converted to colored images (*left*) that reveal where the stresses on the aircraft are greatest. (Image at left courtesy of NASA; image above courtesy of Peter A. Gnoffo and Jeffery A. White/NASA.)

Scientists who used computers found themselves routinely designing new ways to advance science. They became computational designers as well as experimenters and theoreticians.

Computational thinking emerged from within the scientific fields—it was not imported from computer science. Indeed, computer scientists were slow to join the movement.

theory and experiment. Some of them used the term *computational thinking* for the thought processes in doing computational science—designing, testing, and using computational models. They launched a political movement to secure funding for computational science

Computation has proved so productive for advancement of science and engineering that virtually every field of science and engineering has developed a computational branch. In many fields, the computational branch has grown to constitute the majority of the field. For

Learn by doing!

This Course – Learn the Basics

Become part of the scientific computing revolution

Basic tools of computation that will be useful in any field

Computational thinking / design

- Basics of hardware and of software design
- Learn at least one programming language (here, Python)
- Essential numerical methods – differentiate / integrate, linear algebra, solve simple equations, ordinary / partial differential equations

Build "computational thinking" skills with a small project

- Understand the problem and algorithms
- Design, implement, **test / debug** the code
- Use / extend the code to obtain results
- Visualize the results, include them in a report / notebook / presentation

Identify resources for further learning

- **Main goal**: prepare students for research involving scientific computing

Hardware

Computing and Computer Programs

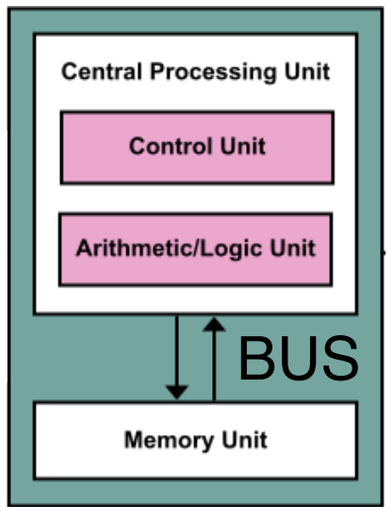
Fixed-program electronic digital computer (ENIAC) $\xrightarrow{1940s}$ stored-program machine (EDVAC)

Modern computers are “stored program computers”
Conceived by Alan Turing in 1936, implemented in 1949 (EDVAC)

Processor (CPU)

Control unit fetches instructions

Arithmetic unit executes them



Also, registers, cache memory

Array of bytes & addresses

Von Neumann architecture

Computer program = series of **instructions**
Instruction = arithmetic calculation or logic operation

Executed program is in “binary” / machine readable format

Compiled languages: Fortran, C++,…: **compiler** translates **source code** into binary

Interpreted languages: Python, PERL,…: don’t compile the code, interpreter translates

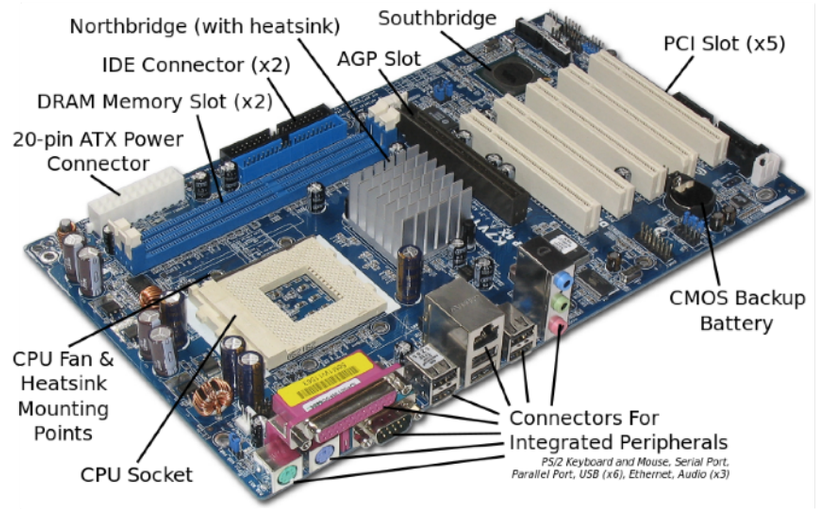
Compiler: a software that converts source into machine-readable code, which can be executed by a computer. Provided by the chip manufacturer (e.g., Intel) or open (e.g., GNU)

How to Build a (Small) Computer

Watch 2-part video **“How to build a computer” Newegg TV on youtube**

Part 1 – Obtain the hardware

1. **Computer case**: protects parts, provides ventilation (heat worst enemy of a computer)
2. **Power supply**: converts AC to a useful voltage to operate
3. **Motherboard**: ties all components together



4. **CPU:** most expensive part. Delicate

Intel and AMD main manufacturers

Fan & heatsink on top of CPU

5. **Memory (RAM): volatile,** flushed when computer off

Crucial to get maximum amount; not expensive nowadays

6. **Storage: non-volatile,** typically >1 Tb these days

HDD (hard drive, spinning disk), SSD (solid-state ICs, faster and smaller than HDD)

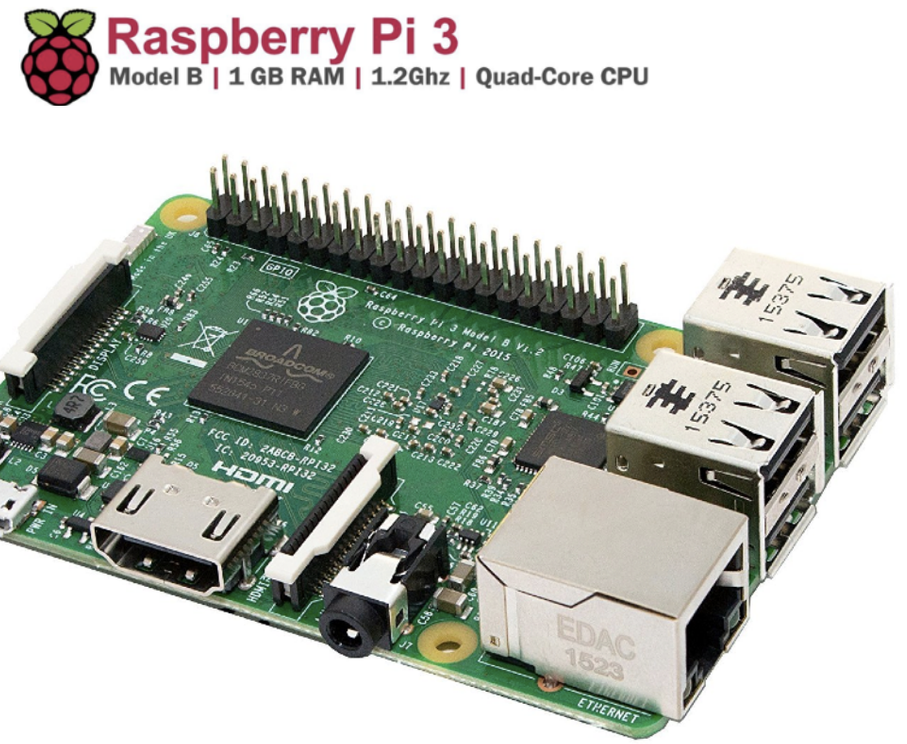
7. **Video card,** embedded in motherboard or separate (plugs into PCI slots).
8. **Peripheral devices:** monitor, keyboard, mouse, etc.

Part 2 – Assemble parts (a few hours)

A Fun Example – Raspberry Pi

Raspberry Pi (RPi) are small computers (credit-card sized) with small compute power
RPi0 costs $5, RPi3 roughly $35 (kit with case, heat sink, 32 Gb SD card, power is $65)
RPi0: Single-core CPU (ARM v6), **RPi3:** quad-core (ARM v8 in RPi3)
RPi3: 1 Gb RAM, Video Core IV GPU, MicroSD (4+ GB) for boot media and storage

Flash UNIX OS to MicroSD card through website
Install python, gfortran, mpich, etc
Can run density functional theory on it!



Can even make a cluster of several RPi's:

Pico 5: 5 RPis with interconnects!

Large Computers – High-Performance Computing (HPC)

For performance, **computing in parallel** is essential.

Use **multiple CPUs** (hardware), **distribute workload** (software) efficiently

What are the 5 main parts of a high-performance computer (HPC)?

1. CPUs (each with multiple cores)
2. Memory (RAM, volatile)
3. Nodes
4. Inter-node network for fast communication
5. Non-volatile storage (disks, tape…)

Core: Hardware unit that performs arithmetic operations. CPU typically has > 1 core

CPU: Central Processing Unit. Arithmetic + control units, cache memory. Ambiguous

Node: Physical collection of CPUs + shared memory. Single memory address space

Memory access on-node much faster than off-node

Interconnect: A high-performance data network that connects nodes to each other

Performance measured in **FLOPs/sec (floating point oper. / s). ~ 10 Gflop/sec per core**

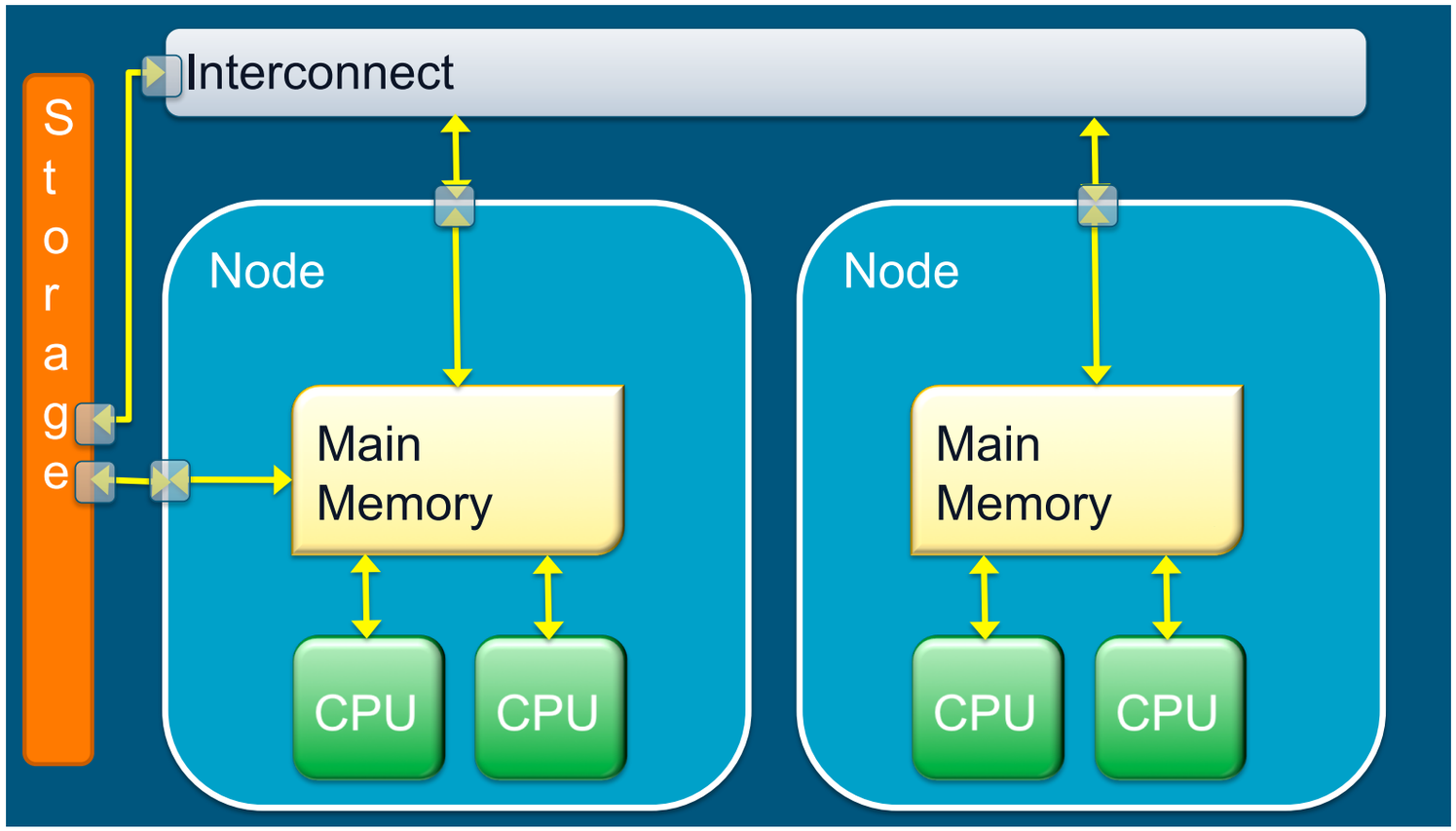
“Petascale”, “exascale” mean large machines with peta / exa flops/sec capability

Typical HPC: Distributed Memory Systems

Many nodes, each with its local memory

Nodes communicate through high-speed network

Designed to run tasks in parallel



Several ways of **parallel programming** to take advantage of hardware

Typical model is SPMD (single program, multiple data), aka "message passing"

Multiple copies (tasks) of the program execute on different processors

Each task computes different data, which is then combined

MPI programming to move / combine data among different tasks

Example 1: My Caltech Computer Cluster

Name	Qty	Unit Price
Storage Node: Pinnacle 3X3601H16	1	10,877.21
Head Node: Pinnacle 2X3601H8	2	4,198.14
Compute Nodes: Pinnacle 1FX3601	30	4,944.87
Compute Node Enclosure	1	1,372.34
Infiniband Switching	1	6,646.41
Network System - Gigabit	1	1,067.06
Management System	1	1,014.07
Rack and Power Distribution	1	10,690.10
Software	1	0.00
Installation & On-site Training	1	3,600.00

Intel Xeon E5-2650 v3
(aka, Intel Haswell)

Total of 30 nodes. 20 cores (2 CPUs) / node

Each node 20 cores, 64 Gb RAM, 1 Tb storage

Roughly 1–10 Tera FLOPs/sec at peak

Total cost ~$200,000.

rack / tower

25 nodes / rack

size of a drawer

Example 2: CORI Supercomputer at NERSC

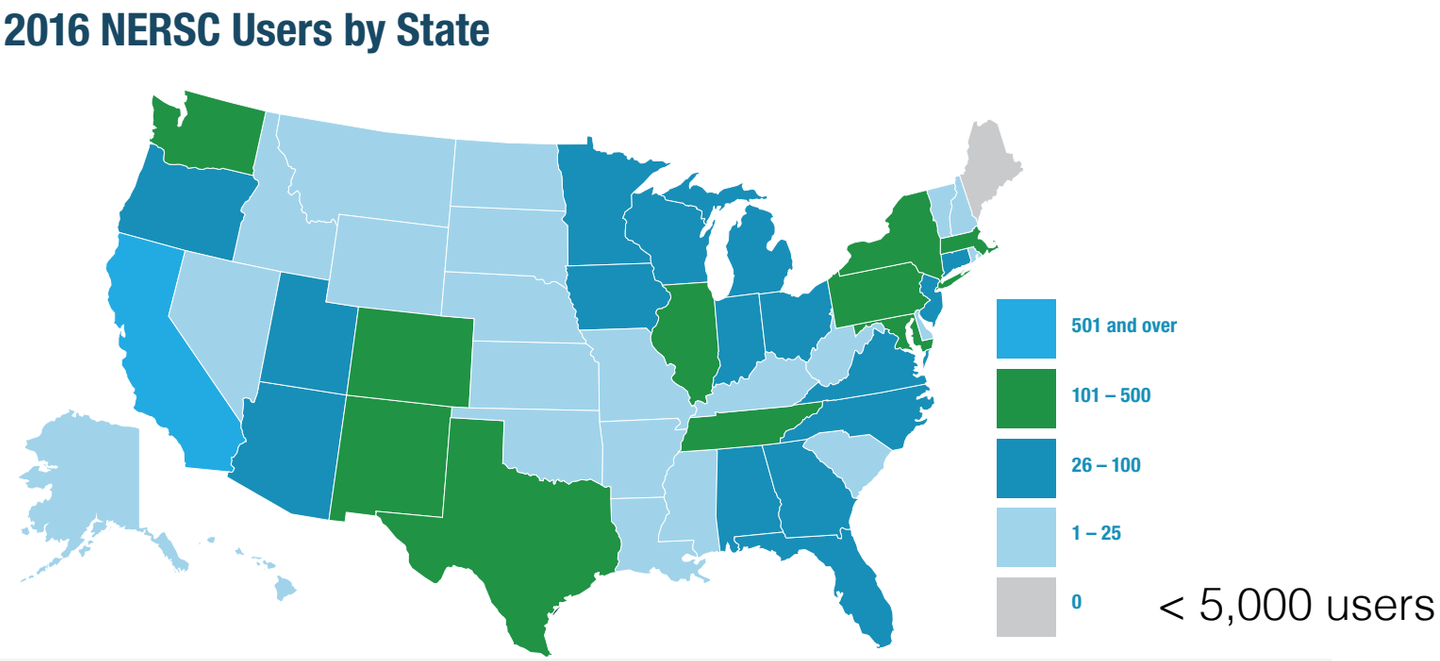
CORI is a petaflop machine at NERSC (see www.nersc.gov)

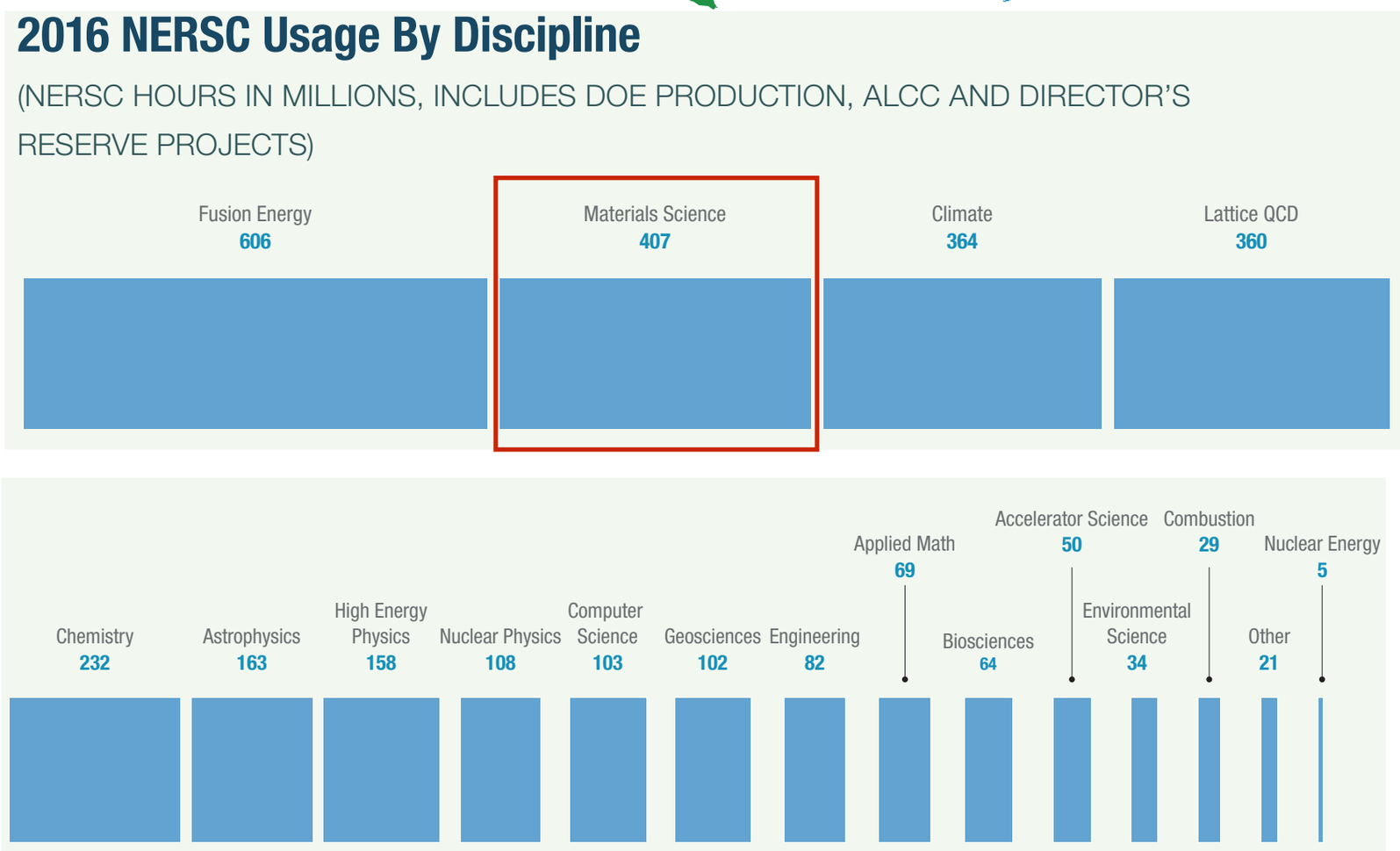
NERSC's Cori supercomputer consists of two partitions, one with Intel Xeon "Haswell" processors (Phase I) and another with Intel Xeon Phi "Knights Landing" (KNL) processors (Phase II), all on the same Cray "Aries" high speed inter-node network. The system also has a large Lustre scratch file system and a first-of-its kind NVRAM "burst buffer" storage device.

Haswell **Cabinets**	14	Each cabinet has 3 chassis; each chassis has 16 compute blades, each compute blade has 4 dual socket nodes
Haswell Compute nodes	2,388	Each node has two sockets, each socket is populated with a 16-core **Intel® Xeon™ Processor E5-2698 v3 ("Haswell") at 2.3 GHz**
		32 cores per node
		Each core supports 2 hyper-threads, and has 2 256-bit-wide vector units
		36.8 Gflops/core; 1.2 TFlops/node 2.81 PFlops total (theoretical peak)
		Each node has 128 GB DDR4 2133 MHz memory (four 16 GB DIMMs per socket); 298.5 TB total aggregate memory.
		Each core has its own L1 and L2 caches, with 64 KB (32 KB instruction cache, 32 KB data) and 256 KB, respectively; there is also a 40-MB shared L3 cache per socket

NERSC Supercomputers (US DOE)

Source: NERSC 2016 annual report





A 10 people group can get up to ~10 million core-hours / year

Software

Survival Skills for Computational Scientists

Excellent resources on edX, software carpentry, etc

1. Basics of UNIX: https://swcarpentry.github.io/shell-novice/
2. Text editors: **VI** , nano, emacs, **Sublime text**
3. Scripting (short codes, processing raw data): **Python**, Perl
4. Plotting raw data: **Python,** gnuplot
5. Writing (serial) code:
Python (interpreted), Fortran, C++ (compiled)
6. Parallel programming: MPI, OpenMP (most challenging task)
7. Building code – libraries, linking and compiling
8. Managing projects – version control: **Git**, SVN
Basics of Git: https://swcarpentry.github.io/git-novice/

See the cheat-sheets for UNIX, VI, Python

Projects and practice are essential to acquire these skills

Basics of Programming: Python

Several courses on edX and Coursera, tutorials on python.org, books

Python is a widely used high-level, general-purpose, **interpreted**, dynamic programming language with great code readability.

It is suitable for both large and small codes / projects
Interpreted language, so **performance is a concern for large projects**

Ideal to create **workflows** and keep track of work; ideal for reproducibility
Example 1: post-process AND plot data in one file
Example 2: Jupyter notebook integrating text and code

Learn basic Python programming
Learn to use **python "packages" (libraries): matplotlib, numpy, scipy**

Numpy & scipy: numerical libraries with many built-in functions, also efficiently create and handle large arrays
Matplotlib: great-looking plots, has become a new standard for plotting

Bundle Python distribution with all these packages: **Anaconda (install it!)**

Our Jupyter notebook will cover the very basics