# Astroinformatics: Day 1

Adam W. Lavely, Ph.D.

Morning Seminar A

June 4 2018

Slides/Files available: `https://goo.gl/Wz72vs`

PennState
Institute
for CyberScience

# Contents

# Contents

# Goals for Today

**Goal 1:** Learn about HPC and best practices

**Goal 2:** Learn how to apply different HPC technologies

**Goal 3:** Learn enough to get more information

**Notes:**

- Expect to be overwhelmed - questions!
- Experience is the best teacher
  - Learn terminology and basic concepts
  - Replicate our examples on your own
  - Replicate an analysis you have already done
  - Google is your best friend

# Who we are

**Adam Lavely** - *awl5173* - PhD in Aerospace Eng; Split between ICS staff (engagement) and AERSP Eng (research)

**Chuck Pavloski** - *cfp102* - PhD in Meteorology; ICS Advanced Technical Services team lead

**Justin Petucci** - *jmp579* - ABD in Materials; ICS iAsk team lead

PennState
Institute
for CyberScience

`ics.psu.edu`

# Contents

# Computing in science

**1990's:** ~~*Two*~~ *Three pillars of science*

Theory

Experiment

Simulation — Problems are too {big / small / fast / slow / expensive / dangerous} for experiments

**2010's** ~~*Three*~~ *Four paradigms of science*

Theory

Experiment

Simulation

Data analysis — Data is too {big / complex / fast / noisy / heterogeneous} for simulation

**Compute that is "large":**

- Large numbers of processors
  - Meteorology, nuclear physics, fluid dynamics
- Large numbers of runs
  - Genomics, astronomy, structural design
- Large memory requirements
  - Financial predictions, molecular analysis
- Large storage requirements
  - Big data analysis, machine learning
- Large runtimes
  - Optimization, Poor coding

HPC **does not** improve every single computational process!

**Your Goal:** Do research.

**Your Task:** Find tools for the best/fastest research

**Options:**

- Build/find a tool specific to your problem
- Find a multi-purpose (very capable) tool

# Finding the right tool

**Domain specific issues:**

- Details about the problem being solved
  - How non-linear? System size? Number of iterations?
- Domain best practices
  - Pull and push with standard tools

**Broader issues:**

- Coding technologies
  - Parallel? Compiled with optimizations? Coupling to outside libraries?
- Hardware
  - I/O speed? Inter-nodal communication speed? Many-core?

**Iterative process:** Get it working right, then running well

# Compute motifs

Couple **new science** with **existing methodologies**.

**Phillip Colella's** 7 HPC Dwarves (2004[1])
*Dense matrix algebra, Sparse matrix algebra, Spectral (FFT) methods , N-body methods, Mapreduce (Monte Carlo), Unstructured grids, Structured grid*

**UC Berkeley's** additional 6 Dwarves (2006[2])
*Finite state machines, Combinatorial logic, Graph traversal, Dynamic programming, Backtrack and branch-and-bound, Graphical models*

Dwarves were originally for comparing hardware beyond lapack but are now used for library development/optimization.

[1] `krellinst.org/doecsgf/conf/2013/pres/pcolella.pdf`
[2] `eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf`

# Contents

# Best practices

**Best practices** are a *systematic way of doing work* which makes it easy to:

- Find, *trust*, and use your code
- Modify your code for alternative purposes
- Prepare your code for publication
- Get feedback and help
- Be known as 'that guy/girl' (in a good way)

# Workflow

1. Define problem and size in compute terms
   - Flow-chart code with data types/sizes
2. Build in parts to get a minimal working example (serial)
   - An example case that requires all functionality at a small scale
   - It helps to know the answer
3. Play with flags/options for optimization
4. Get full-scale simulations going
   - Find the bottlenecks
   - Play with other appropriate compute resources
5. Parallelize/accelerate/optimize using libraries

# Names

Names should be **descriptive** and **consistent**.

**Example:**

```
def calcAmount(A,B,C,D):
    return A*(1+B/C)**(C*D)
```

*or*

```
def calcCompoundInterest(principal,rate,compInterval,time):
    return principal*(1+rate/compInterval)**(compInterval*time)
```

- Functions
- Variables
- Input/output files
- File and directory names

# Documentation

Documentation should be **descriptive** with both **what** is being done and **why**.

```
def calcCompoundInterest(principal,rate,compInterval,time):
    # Calculate the value of an investment with constant
    # interest rate compounded at regular intervals over time

    # principal - initial value
    # rate - interest rate (5% interest entered as 0.05)
    # compInterval - number of times per year interest is compounded
    # time - time the investment sits, in years
    return principal*(1+rate/compInterval)**(compInterval*time)
```

- Describe the contents of a file
- Describe variables when they are first used
- List references for atypical algorithms/methods
- A readme file with an overview of the code
- Usage and citation instructions

# Modularity

**Your analysis might be unique, but most of your code isn't...**

*Modularity:*

- Develop functions for individual tasks
- Write unit tests for each function
- Provide simple example for each/all functions
- Reuse in future endeavors

Your functions and documentation are an *Application Programming Interface (API)* for your code.

# Code Publication

**Publish your code to:**

- Share your tools with the world
- Analysis reproducibility
- Get a job

**Where to publish?**

| | |
|---|---|
| Journals | With paper[3] or without[4] |
| Websites | Personal or lab websites |
| Github | Or other repo |

[3] software.ac.uk/which-journals-should-i-publish-my-software

[4] joss.theoj.org/

# Publishing Code

**Self-contained package should have:**

| | |
|---|---|
| Code | All the code and a listing of external libraries required |
| Instructions | Compilation and execution |
| Examples | Sample input/output to verify compilation |
| License | Choose one (next slide) |

**and could have:**

| | |
|---|---|
| Feedback | Bug reporting and questions |
| Citation | List of papers for users to cite |

# Licensing Terms

| | |
|---|---|
| Free Software | Code that users can run/study/change and redistribute with or without changes[5] (social movement - freedom) |
| Open Source | Code that users can run/study/change and redistribute with or without changes (software development methodology - source code) |
| Freeware | Free to use but source is closed |
| F(L)OSS | Free(/Libre) and Open Source Software |
| Copyright | Creator of work has exclusive rights for use/distribution |
| Copyleft | Freely distribute copies and modified versions and same rights are preserved in derivative works |
| Commercial | Software made by a company (may be FLOSS) |
| Proprietary | Closed source software (may be free) |

[5] `gnu.org/philosophy/open-source-misses-the-point.html`

**How do I give my software a license?**

1. Chose one of many[6] (many, many,...) licenses available:

| | |
|---|---|
| Accessible | **GNU GPL**, Expat (MIT), Unlicense, *informal*[7] |
| Limited | APSL, OSL, CDDL, MS-PL, OpenSSL |
| Restrictive | *No license*, AT&T, Open Public License |

2. Put a line in your README file that says:

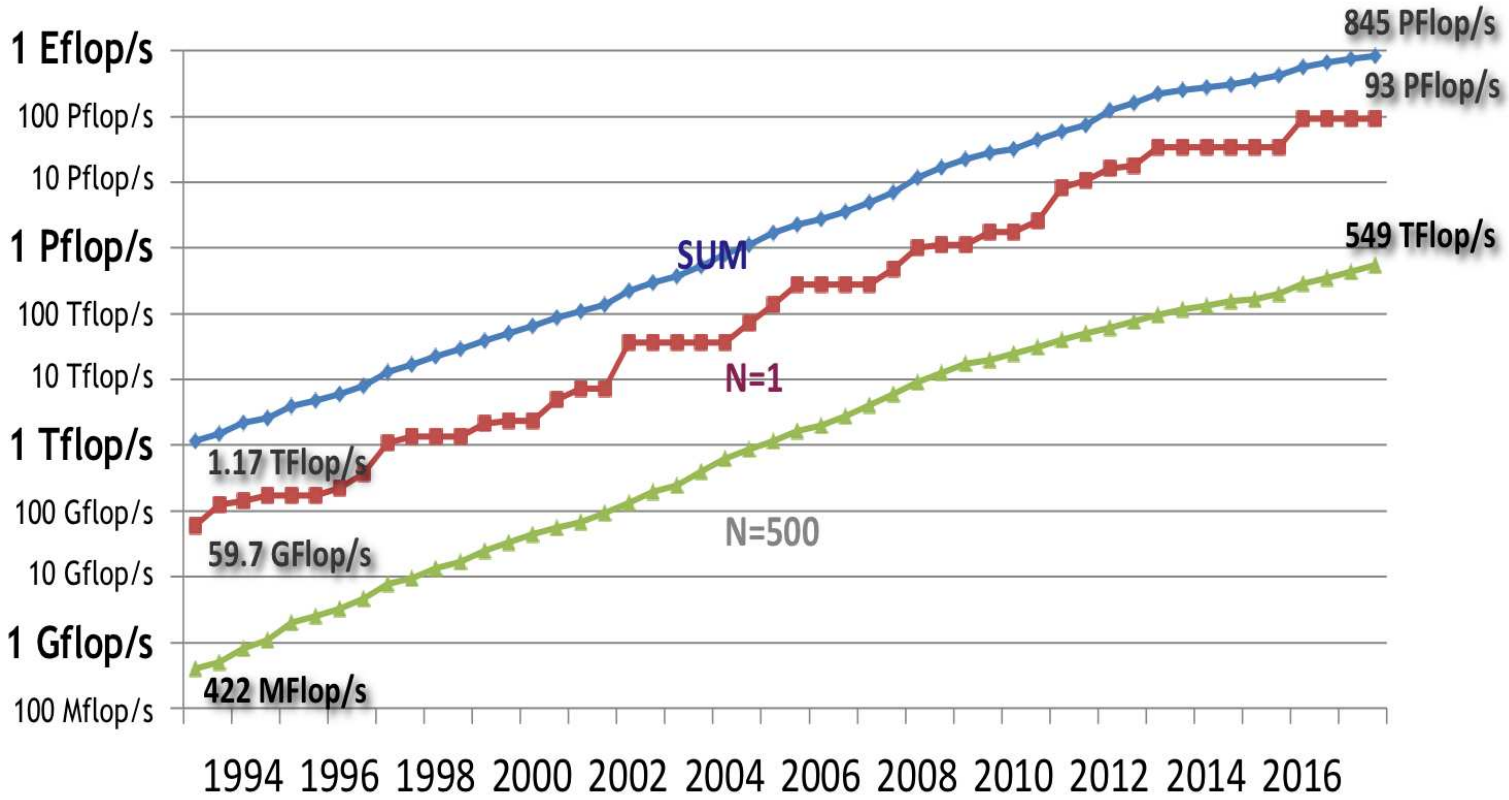
```
This project is licensed under the terms of the XYZ license
```

3. Put a copy of the license file in the directory with you README called LICENSE or LICENSE.txt.

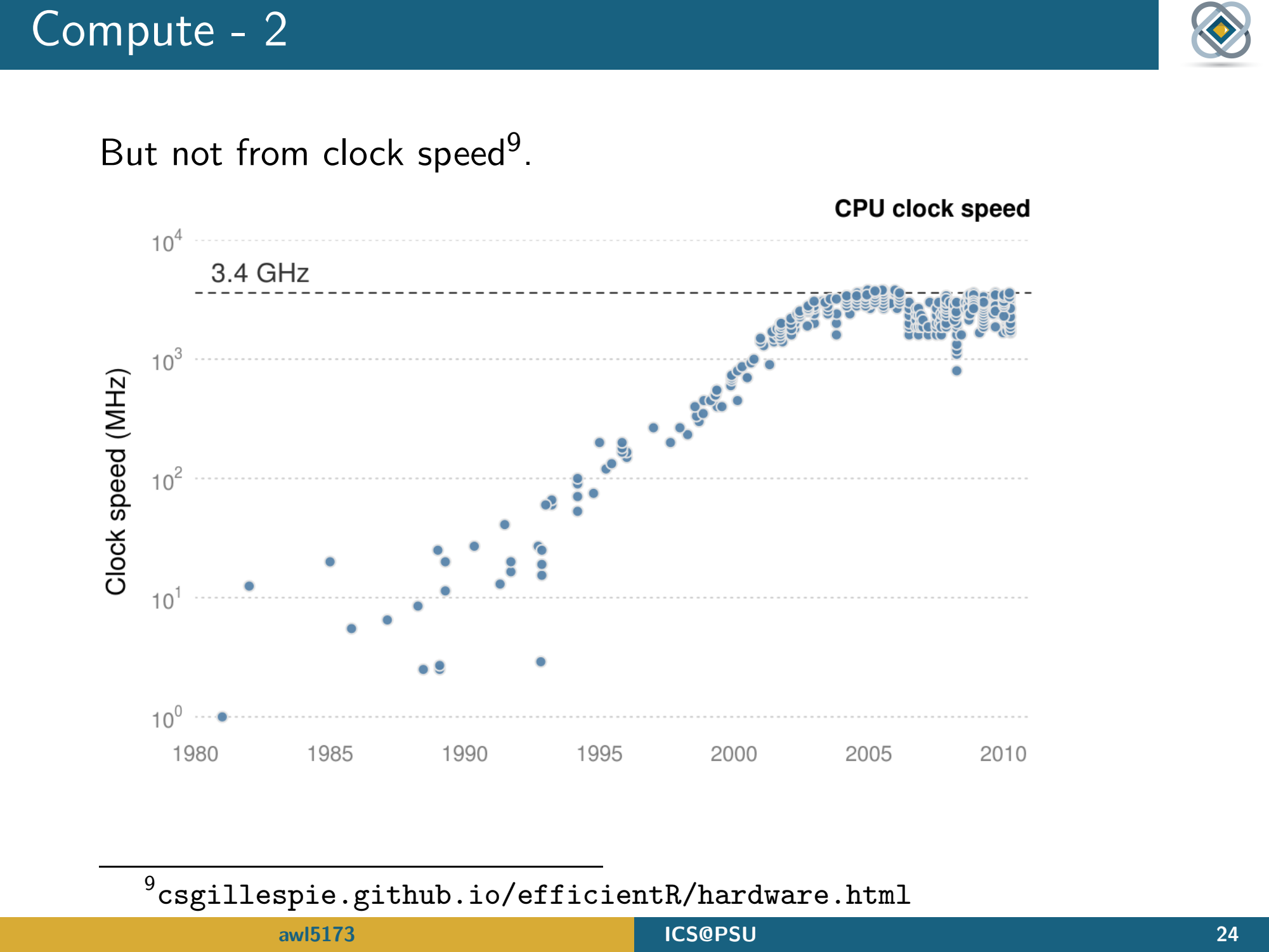
[6] gnu.org/licenses/license-list.html
[7] US only

# Contents

Computers are getting faster[8]...




[8] Top 500, Fall 2017; Figure from Yelick

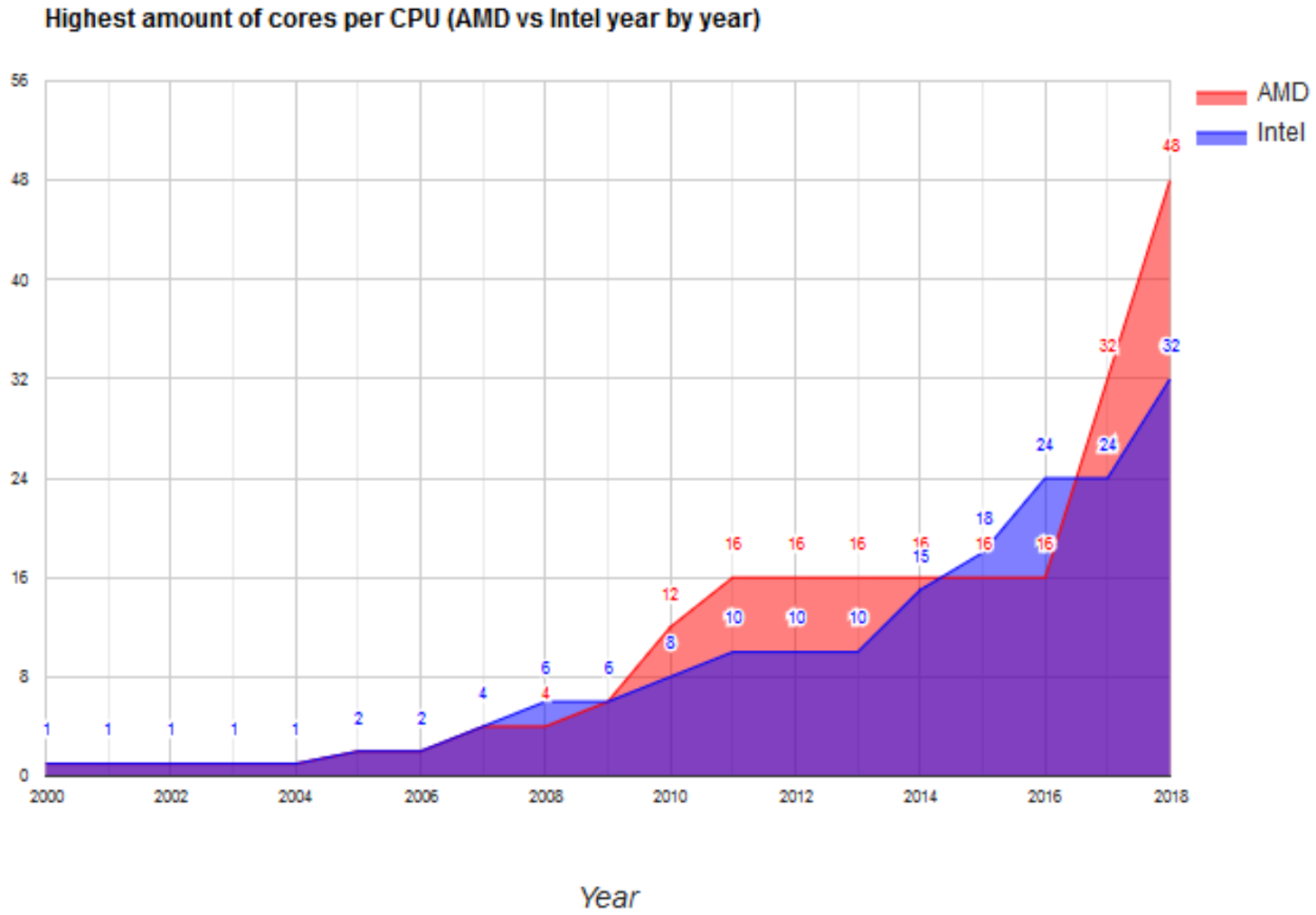


But not from clock speed[9].



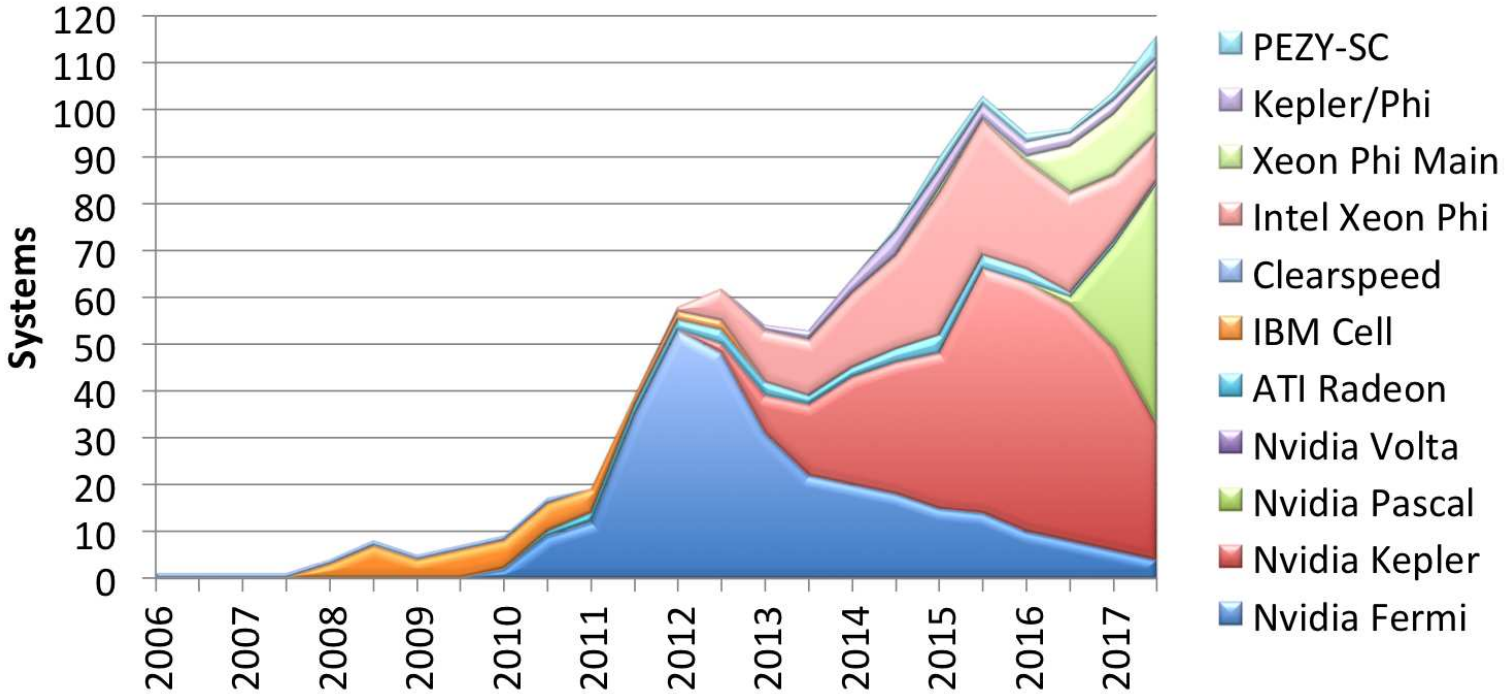

[9] csgillespie.github.io/efficientR/hardware.html

Increases are coming from more cores per chip[10]...




[10] i.imgur.com/Gad4cKk.png

and accelerators.[11]




[11] Top 500, Fall 2017; Figure from Yelick

# How do I improve my code?

A lot of *gains* can be made by taking advantage of *multicore and acceleration*, but *bad serial code makes terrible parallel code.*

- Understand what is performing poorly
- Attack this in serial first
- Parallelize or accelerate

# Bottlenecks

Find what parts of your code are slow:

- Know what to work on
- Know your requirements to find other resources

**Common Performance Improvements:**

| | |
|---|---|
| I/O | Reduce I/O whenever possible, use filesystem optimizations |
| Parallel I/O | Use existing libraries (hdf5, netCDF) |
| Algorithms | Use appropriate optimized libraries (boost, mkl, petsc, trilinos) |
| Coms | Reduce amount and frequency |
| Code/flags | Loop unrolling, **cache optimization** |

# Ways to Identify Bottlenecks

**Look at past processes:**

- logfiles
- tracejob output (batch)

**Watch running processes:**

- top or ps -aux
- showq -r (batch)

**Rerun with extras:**

- /usr/bin/time -v
- print statements
- strace

**Instrumented runs:**

- debuggers and profilers (vtune, tau)

# Memory

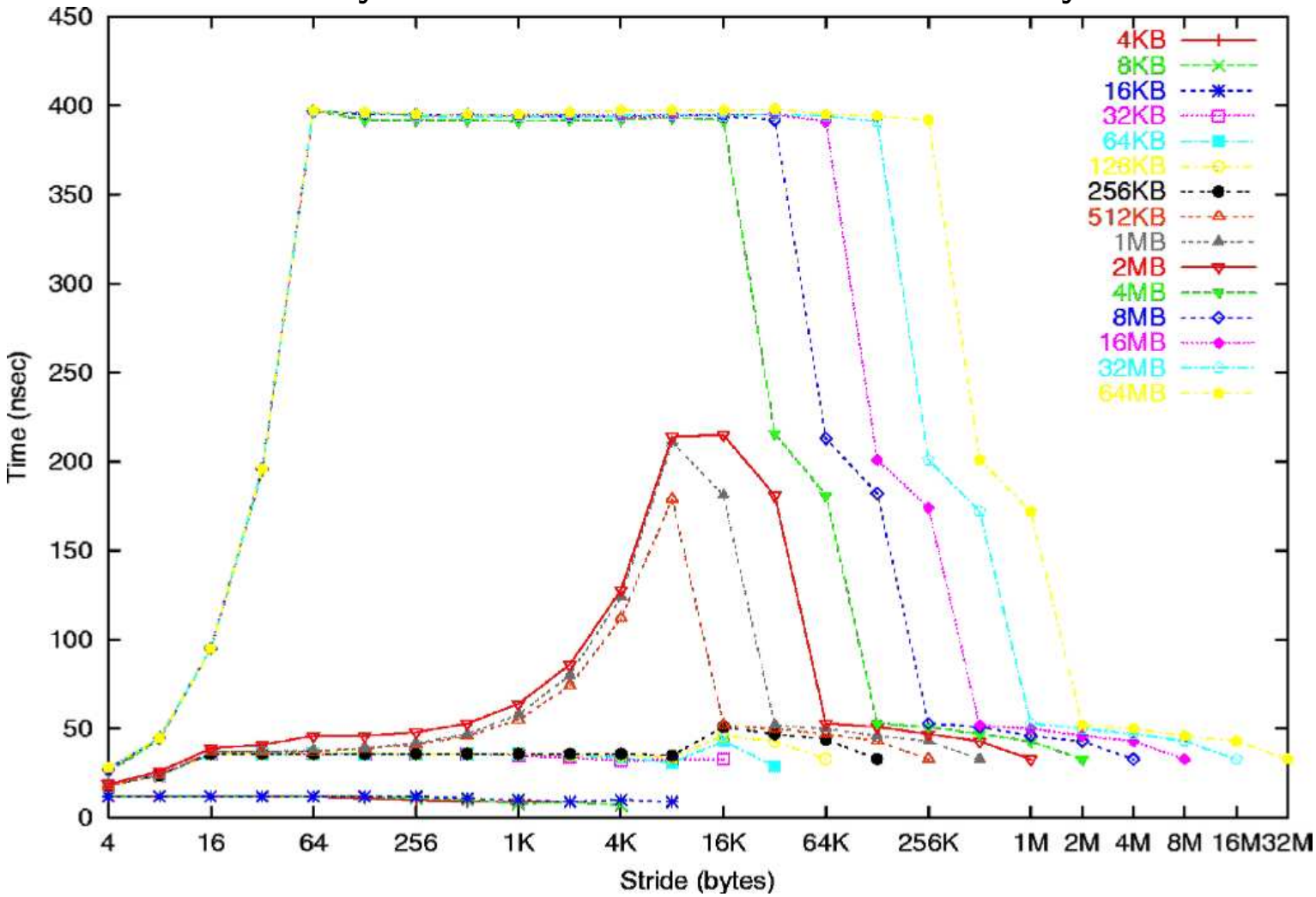
| | |
|---|---|
| Register | On each core, memory required for the cycle at hand |
| Cache | On each core or processor, the block of data that includes what is required for the register |
| RAM | On each processor, things that are read from disk |
| Disk | Shared among all processors, your local filesystem |
| Non-local | Offsite data storage |

| **Type** | L1 cache | L2 cache | RAM | Disk | Non-local |
|---|---|---|---|---|---|
| **Speed** | 1 ns | 10 ns | 100 ns | 10 ms | 10 sec |
| **Size** | KB | MB | GB | TB | PB |

**Note:** These are all hardware dependent (use optimized tools).

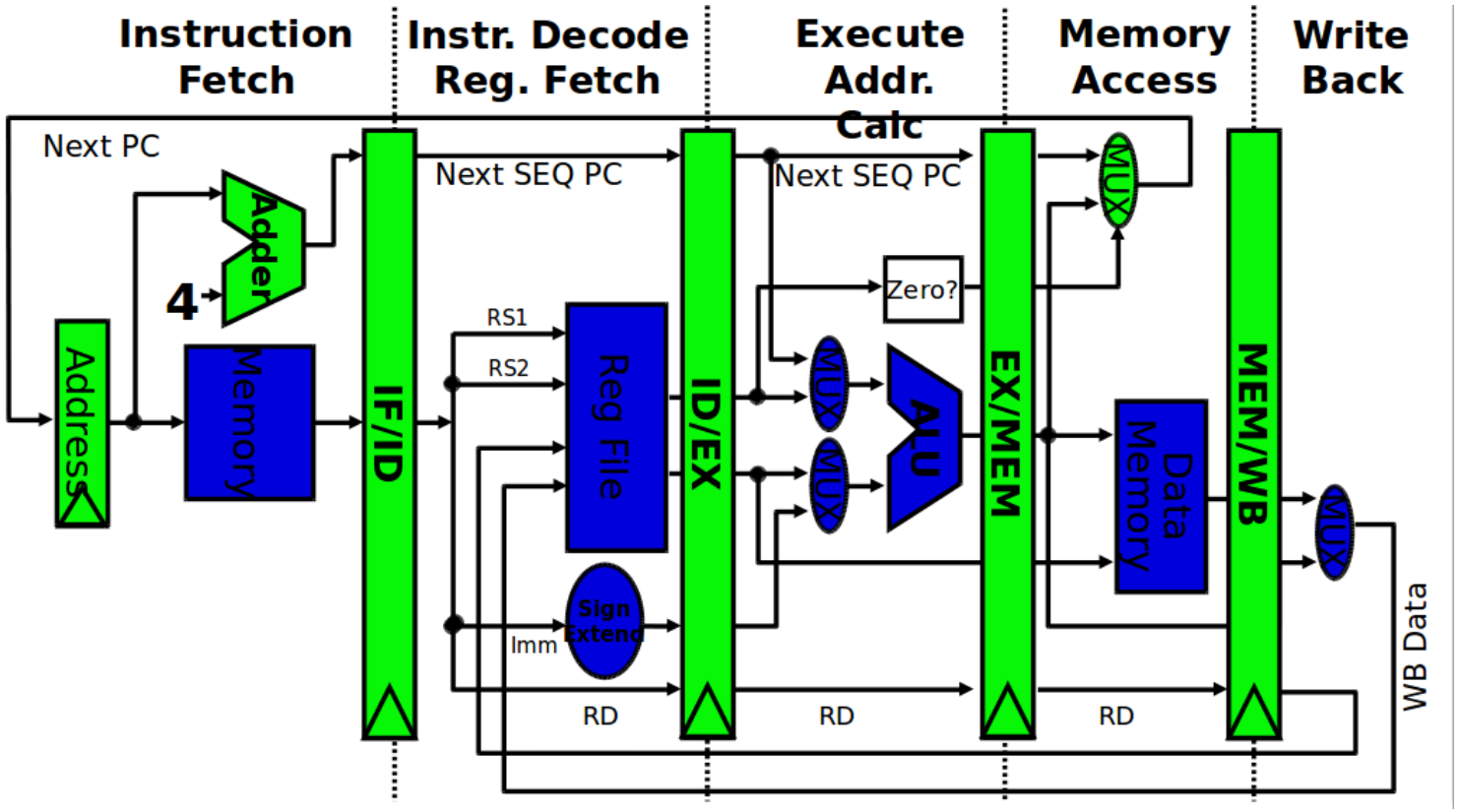
# Memory Speed Example

Time for memory access for different size memory blocks[12]




[12] cs.berkeley.edu/~yelick/arvindk/t3d-isca95.ps

Microprocessor without Interlocked Pipeline Stages (MIPS)[13]




Processors have stages - multiple things can be 'in process' at once.

[13] Figure 3.4, Page 134 , CA:AQA 2e by Patterson and Hennessy

# Parallelism within processors - 2

**Pipelining:**
Increase *bandwidth* without changing *latency* by taking advantage of processor stages. *Bandwidth:* Number of outputs per unit time
*Latency:* Time for individual instruction

**Single Instruction, Multiple Data (SIMD):**
Modern chips can do multiple things at once.
*Sandy Bridge* - AVX; *Haswell* - AVX2; *Skylake* - AVX-512

**Fused-Add Multiply:**
Multiple steps can be done at once
$x = y + c \cdot z$

**What ideal speed up is possible?**

$t$ = time
$P$ = percentage of code that can be parallelized
$N$ = number of cores being used
$S$ = speed up

$$t_N = \underbrace{(1-P)}_{\text{serial part}} + \underbrace{P/N}_{\text{parallel part}} \quad (1)$$

$$S = \frac{t_1}{t_N} \quad (2)$$

**Amdahl's Law:**

$$S = \frac{1}{(1-P) + \frac{P}{N}} \quad (3)$$

**Example A:** 50% of the code can be parallelized, running on 4 processors:

- Amdahl: S=1.6

**Example B:** 75% of the code can be parallelized, running on 16 processors:

- Amdahl: S=3.4

**Example C:** 98% of the code can be parallelized, running on 200 processors:

- Amdahl: S=40.2

Non-parallelized portion of the code causes processors to sit idle.

# Comparing to ideal parallelization

**Reasons to beat ideal (super-linear scaling):**

- Variables are 'too large' for a single core
  - Memory going from disk to RAM or RAM to cache
  - I/O buffer sizes

**Reasons to you won't beat ideal (sub-linear scaling):**

- Overhead associated with starting/initiating parallelization
- Communication between cores
- Memory synchronization
- Redundant computation

# Parallelization Methods

| | |
|---|---|
| Granularity | How large each task should be |
| Locality | Moving data is costly |
| Load | 999 cores not waiting on 1 |
| Sharing | Coordination/synchronization of data safely |

Parallelization is a {costly / beneficial} tool.

- Do the results from one task influence the other?
- How often does information need to be shared
- What resources (RAM, procs) are required?
- What's already available to build upon?

# Parallelization Methods

**Dinner table analogy:** Who has access to what food?

- Family style
- Fancy restaurant

**Shared Memory:** All cores have access to all memory

**Distributed Memory:** Cores only have access to their memory only

# Shared Memory Parallelization

**Benefits:** Typically easier to code and understand

**Difficulties:** Data safety, debugging

Single node — Local Address Space

- *Loops:* OpenMP
- *Tasks:* Pthreads

Multi-node — Partitioned Global Address Space (PGAS)

- *Extensions:* UPC, UPC++, Fortran 2008
- *Novel:* X10 (java based), Chapel

**More info in afternoon session 1.**

# Distributed Memory Parallelization

**Benefits:** Powerful

**Difficulties:** Lots of complicated code

Chatty — Message passing interface (MPI)

- *Implementations:* OpenMPI, iMPI, MPICH

Quiet — Map-reduce framework

- *Embarrassingly Parallel:* Monte-Carlo, job arrays
- *Big-Data:* Spark, Hadoop

**More info in afternoon session 2.**

# Languages

**Long ago:**

Programming Languages Fast to execute! Compiled into machine code executables, executed in separate step

Scripting Languages Fast to write! Interpreted interpreted/executed in real-time

**Now:**

- Compilers/interpreters available for almost every language
- Possible to combine compiled and interpreted portions
- Just-in-time (JIT) compilation

# Language Choice

**Law of the instrument:**
*If all you have is a hammer, everything looks like a nail*

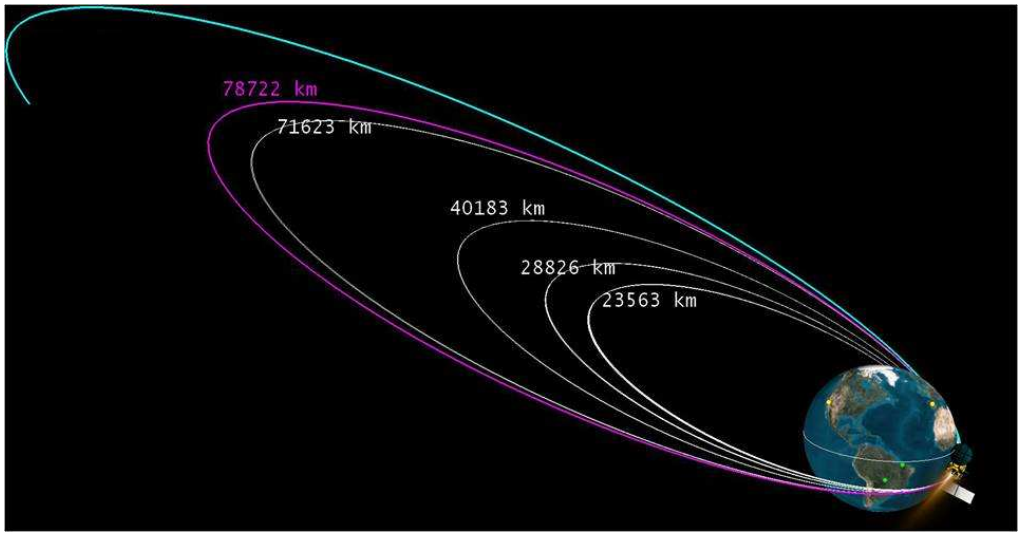
**Languages exist for at least one reason:**
*Speed of writing:* Python, javascript, bash, ....
*Speed of execution:* C, fortran,...
*Make something easier:* Chapel, LabVIEW,...

**Use a language that fits your needs:**

- Functionality (included and available libraries)
- Speed (writing and executing)
- Ability to profile and optimize
- Portability

**There are no wrong choices, but lots of not-good choices.**

*Genetic Algorithm for Satellite Raising*[14]




**Full problem**
Matlab: 3-4 hours
C++: 4 mins

**Time-step**
GPU: 0.015s
CPU: 0 0001s

[14] CSE 597: Vidullan Surendran and Will Paik

# Coffee Break

**Restart at 11 AM**

**Please stay after if you need help with:**

- Accessing a terminal
- Starting Jupyter