

# Week 1: 02 – Introductions

# Acknowledgements

2

Similar courses that were helpful in preparing these lectures

- G63.2011.002/G22.2945.001: High Performance Scientific Computing
  - Marsha Berger & Andreas Klöckner – NYU
- CS 759: High Performance Computing for Engineering Applications
  - Dan Negrut – Univ. of Wisconsin
- CS267: Applications of Parallel Computers
  - Jim Demmel – Berkeley
- CS525: Parallel Computing
  - Ananth Grama – Purdue
- Seminar Series on High Performance Computing at The National Institute for Computational Science, University of Tennessee, Knoxville

Textbooks

- Introduction to Parallel Computing by Grama, Gupta, Kumar, Karypis
- Introduction to High-Performance Scientific Computing by Eijkhout

# Housekeeping

3

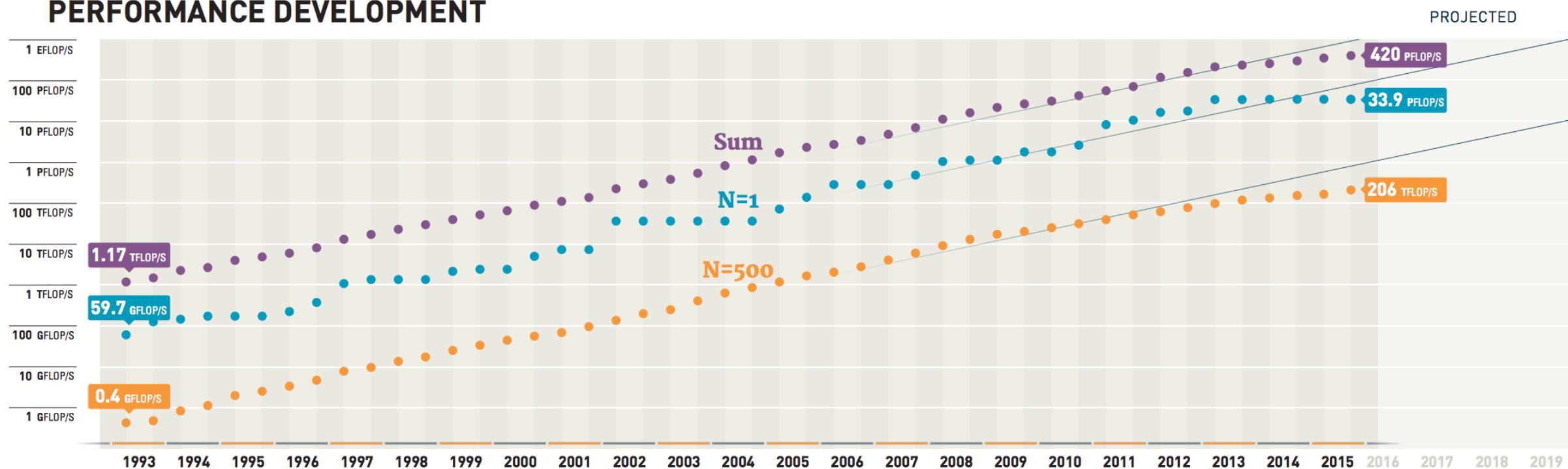
- Students who were unable to register
- Drop date: Monday, 01/18/2016
- OpenMP and MPI: C/C++ is preferred. Python could be tricky
- Homework clarifications
  - Github private repo
  - Besides your code, include a pdf with a brief writeup describing the hot spots in your code (if any), and things you could do to speed it up.

# Recap

4

- Moore's Law and the inevitability of parallel computing
- What is HPC

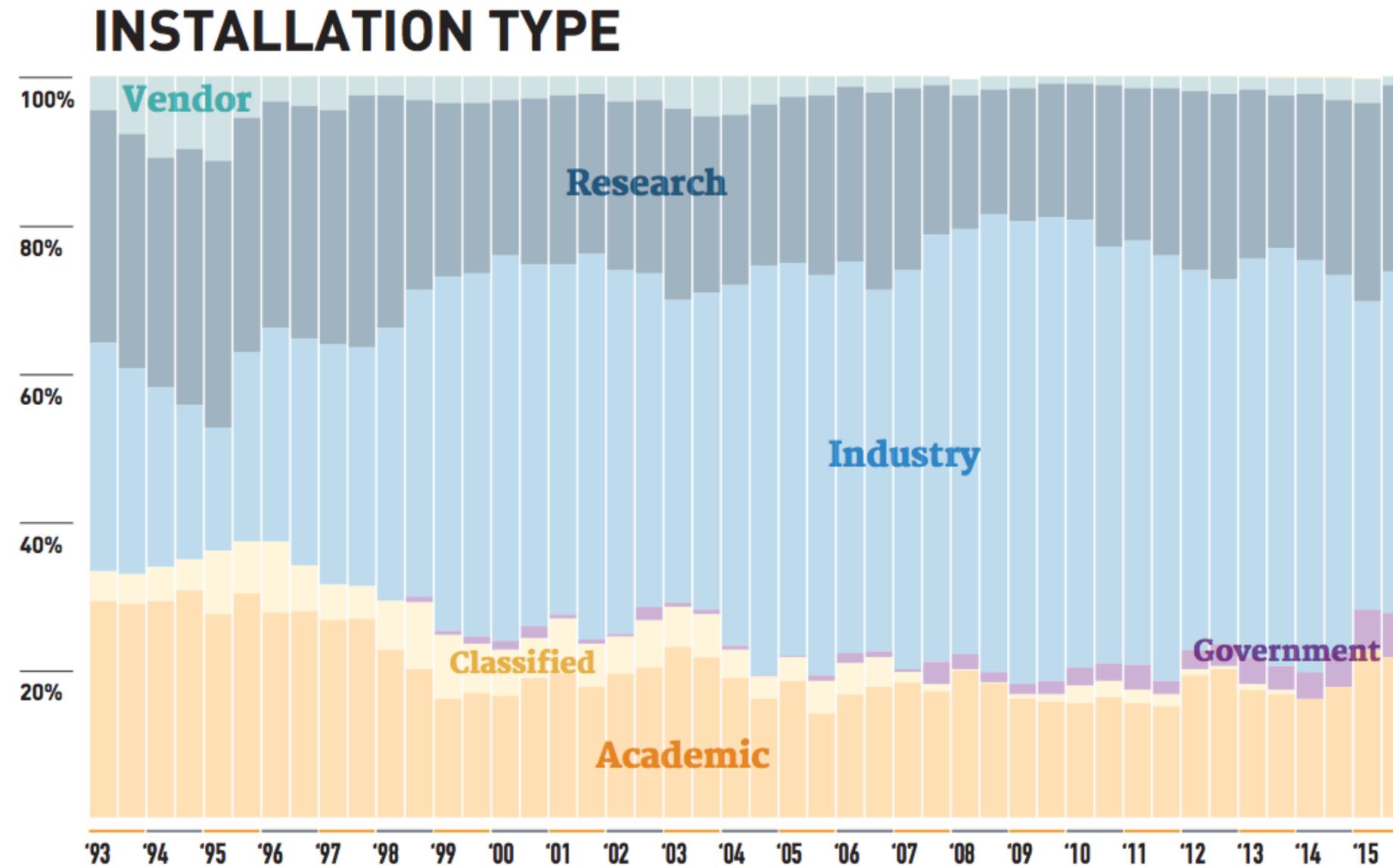
## PERFORMANCE DEVELOPMENT



	NAME	SPECS	SITE	COUNTRY	CORES	R <sub>MAX</sub> PFLOP/S	POWER MW
1	Tianhe-2 (Milkyway-2)	Intel Ivy Bridge (12C 2.2 GHz) & Xeon Phi (57C 1.1 GHz), Custom interconnect	NUDT	China	3,120,000	33.9	17.8
2	Titan	Cray XK7, Opteron 6274 (16C 2.2 GHz) + Nvidia Kepler GPU, Custom interconnect	DOE/SC/ORNL	USA	560,640	17.6	8.2
3	Sequoia	IBM BlueGene/Q, Power BQC (16C 1.60 GHz), Custom interconnect	DOE/NNSA/LLNL	USA	1,572,864	17.2	7.9
4	K computer	Fujitsu SPARC64 VIIIfx (8C 2.0 GHz), Custom interconnect	RIKEN AICS	Japan	705,024	10.5	12.7
5	Mira	IBM BlueGene/Q, Power BQC (16C 1.60 GHz), Custom interconnect	DOE/SC/ANL	USA	786,432	8.59	3.95

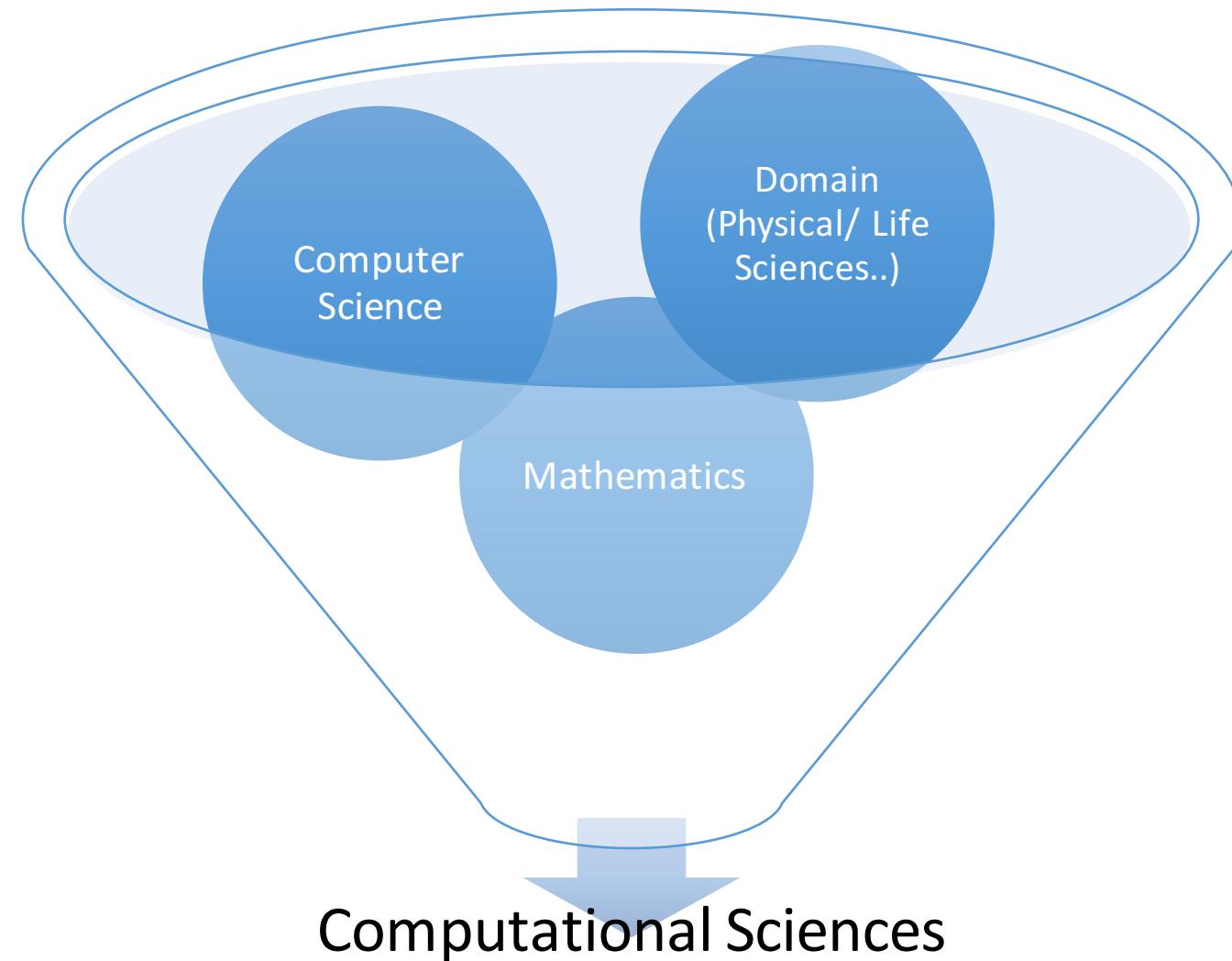
# Who uses HPC?

6



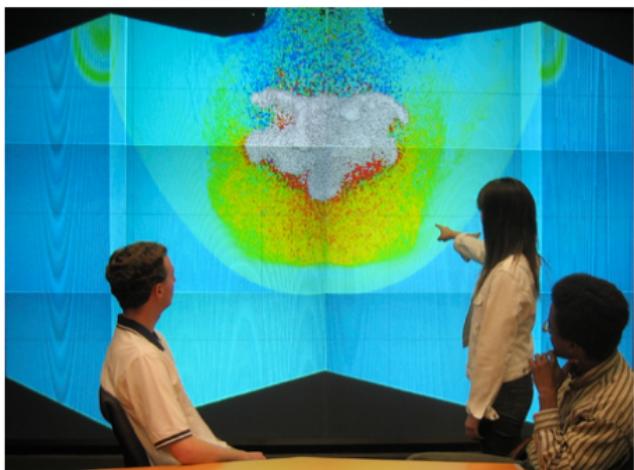
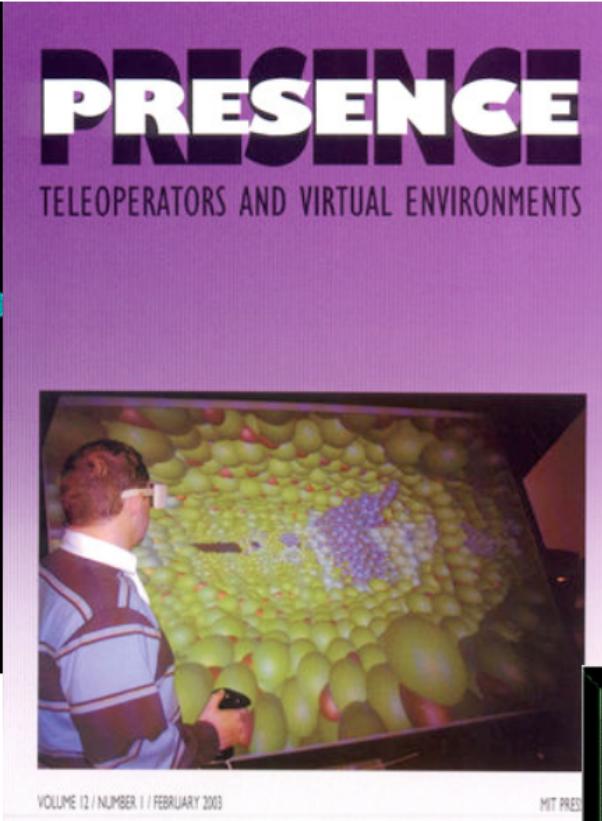
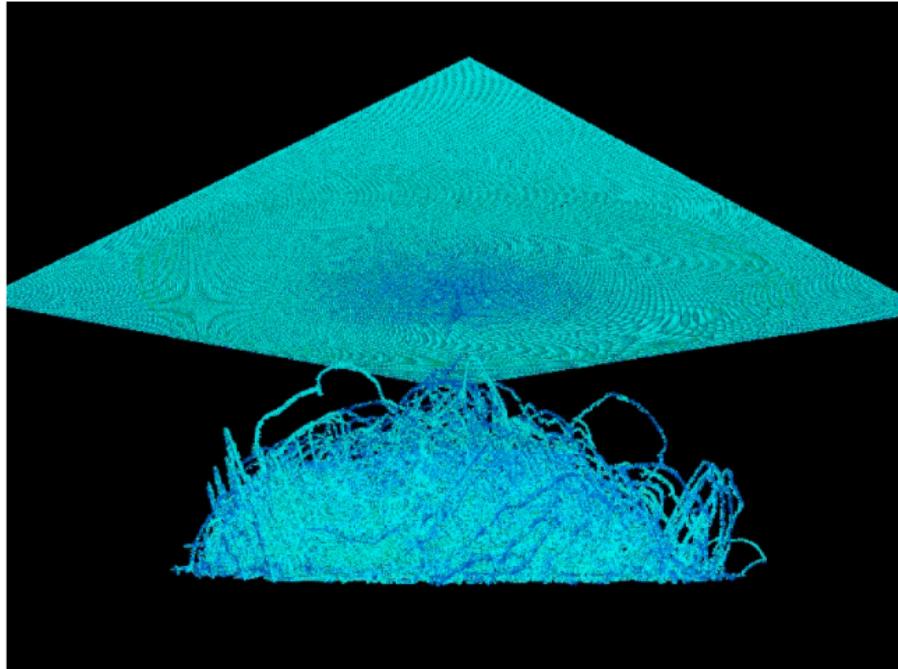
# Where is HPC used?

- Bioinformatics and Computational Biology
- Biomedical Informatics
- Cancer Research
- Computer Vision
- Drug Discovery
- Game Shows (IBM Watson)
- Material Sciences
- Movies
- Natural Language Processing
- Nuclear Fusion
- Precision Medicine
- Retail
- Weather modelling and Prediction

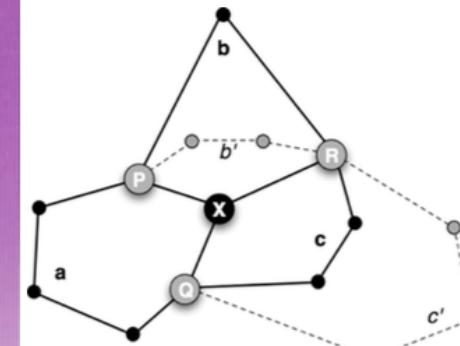


# Material Science

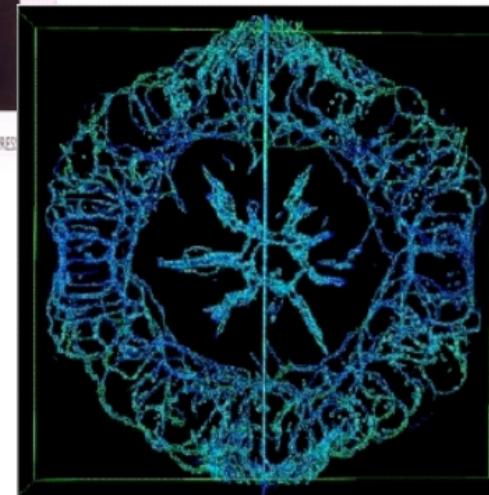
8



**Immersive & interactive  
visualization of  
billion atoms**



**Graph-based  
data mining of  
chemical bond  
network topology**



# Precision Medicine

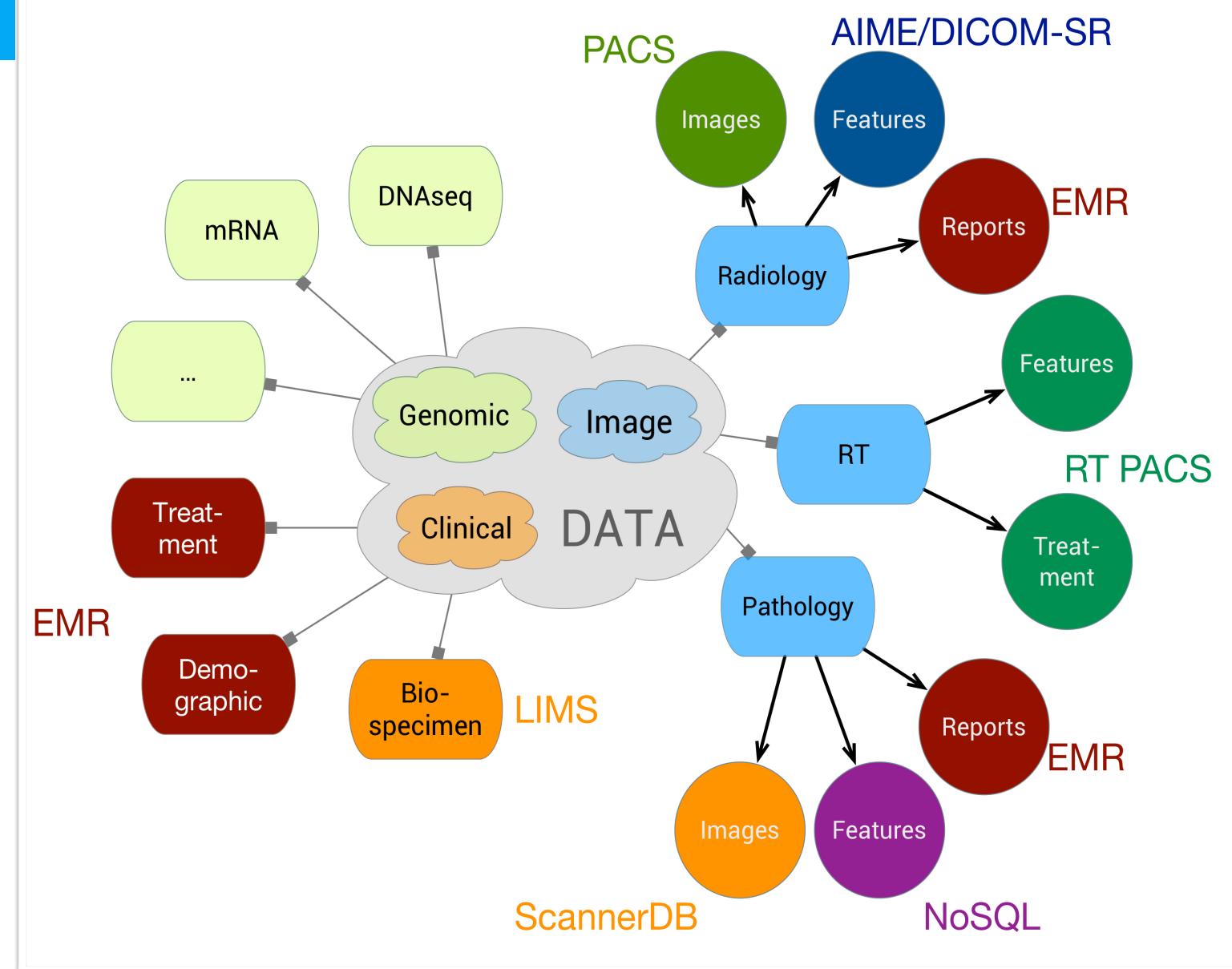
9

## DATA CHARACTERISTICS

Large number of small datasets  
Structured...Semi-structured  
...Unstructured...Ill formed  
Noisy and Fuzzy/Uncertain  
Spatial, Temporal relationships

## DATA MANAGEMENT

Variety in storage and messaging protocols  
No shared interface



# Hollywood



- DreamWorks has a "render farm" of servers made up of about 20,000 processors (HP BladeSystem c-Class server blades).
- The image rendering jobs are broken up into small pieces, distributed out to the server farm, and are later recompiled to create the final images for a film.
- Required a whopping 80 million compute hours to render, 15 million more hours than DreamWorks' last record holder, "The Rise of the Guardians."
- Between 300 and 400 animators worked on "The Croods" over three years.
- After completing a film, about 70TB worth of data (things like background art or plants) is stored for future usage in future productions.

With Moore's law, why aren't  
we computing @2x  
data/speed every 18months?

# Ideal vs. Real Processors

12

## IDEAL

Processors can only compute on what is in memory

Memory Addresses → Bytes/Int/Floats/...

Compute → Rd/Wr to Registers  
→ Math/Logical ops on data  
→ All Data in registers

Ordered execution

Cost of computing =  $\Sigma(Rd, Wr, Math ops)$  and Rd, Wr, Math = 1 cycle

## REAL

1. Registers and Caches  
Cost of mem ops =  $f(type, size, locality)$
2. Implicit Parallelism in processor  
FunctionalOps can run in parallel
3. Pipelining  
Another form of parallelism

Compilers need help so they can best optimize your code

# MEMORY

## IMPLICIT PARALLELISM

## PIPELINING

# Latency vs. Bandwidth

14

- Consider the example of a fire-hose. If the water comes out of the hose **two seconds after the hydrant is turned on**, the **latency** of the system is two seconds.
- Once the water starts flowing, if the hydrant delivers water at the **rate of 5 gallons/second**, the **bandwidth** of the system is 5 gallons/second.
  
- If you want faster response from the hydrant → reduce latency.
- If you want to fight big fires, → increase bandwidth.

# A better example: (**Calculate dot product of 2 vectors**)

15

- 1Ghz processor → 2 Mul-Add → 4 FP ops/sec
  - Peak Performance = 4GFlops ( $4 \times 1 * 10^9$ )
- Processor has to fetch data from memory (no cache)
  - Latency = 100ns
  - Block size = 1 word
- What is the Peak Performance
  1. Issue command
  2. Wait (latency) for 100ns
  3. Compute
  4. Peak Performance = 1 mul-add / 100ns → 10MFlops

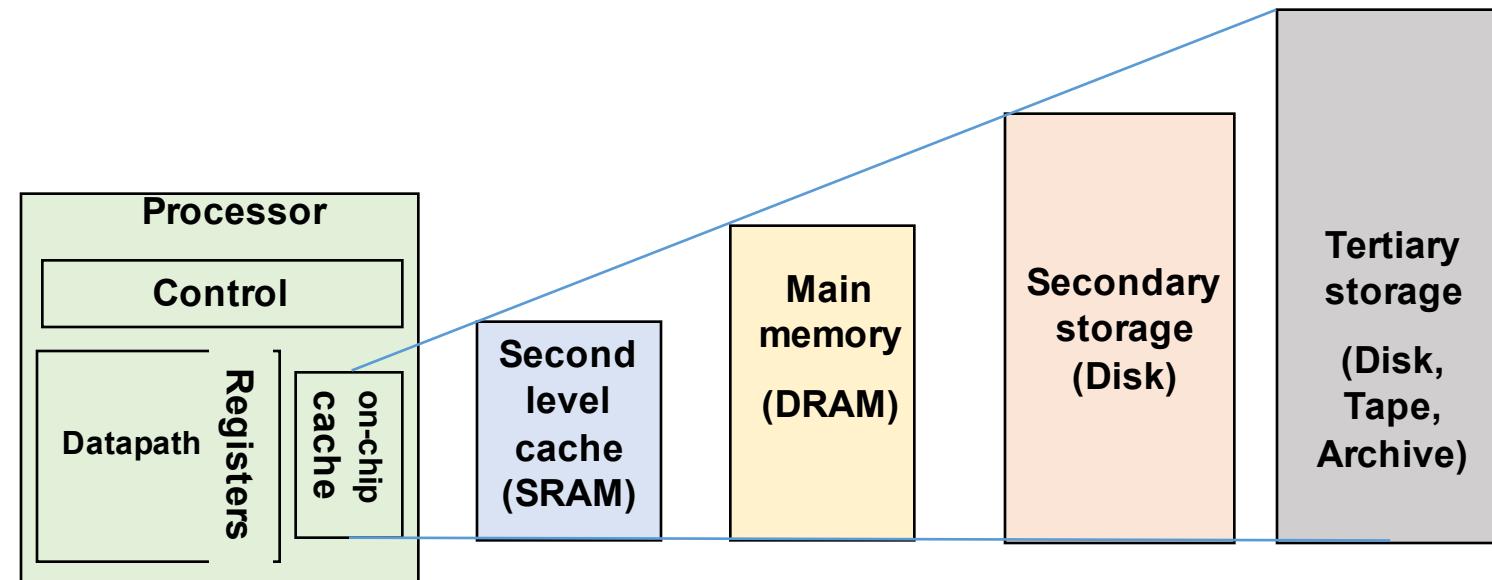
4GFlops vs 10MFlops!!!

# Memory Locality and Hierarchy

16

Temporal Locality: Reuse data that was used (and accessed) in prior ops

Spatial Locality: Data that is spatially (in memory) close to what you are accessing (was just accessed)



Speed	1ns	10ns	100ns	10ms	10sec
Size	KBs	MBs	GBs	TBs	PBs

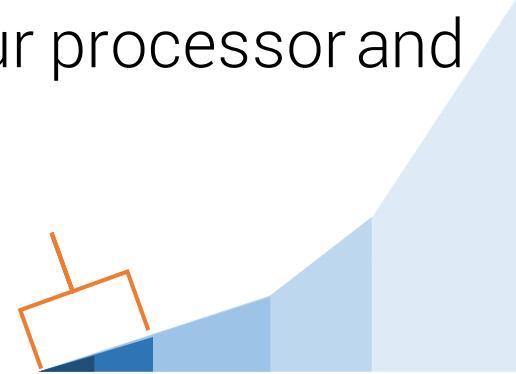
## Reduce Latency + Increase Bandwidth

1. Reduce/Eliminate mem ops by keeping data in fast (**small**) memory and reuse it
  - **Reduce Latency via TEMPORAL LOCALITY**
2. Fetch a chunk of memory and saving it in fast (**small**) memory and use the chunk
  - **Maximize Bandwidth via SPATIAL LOCALITY**
  - Bandwidth improving @ ~23%/yr vs latency 7%/yr
3. Maximize bandwidth usage by issuing multiple reads
  - Load an entire array vs one element/read
4. Overlap computation & memory operations
  - Pipelining and Prefetching

# Reducing Memory Latency w/ Cache

18

- Cache are small memory elements that live between your processor and the DRAM
- Cache hit ☺ vs Cache-miss ☹
- Cache acts as a low-latency high-bandwidth storage.
- If a piece of data is repeatedly used, the **effective latency** of this memory system **can be reduced** by the cache.
- The **fraction of data references satisfied** by the cache is called the **cache hit ratio** of the computation on the system.
- Cache hit ratio achieved by a code on a memory system often determines its performance.



# In practice:

19

1Ghz processor → 100ns Latency → block size – 1word → 4ops/cycle  
32KB Cache + 1ns Latency

Task: Multiply two 32x32 element matrices

1. Fetch Matrix A, B
2. Size(A) = 1024 words. Time →  $1024 * 100\text{ns} = \sim 100\mu\text{s}$   
 $200\mu\text{s} \rightarrow$  A, B in cache
3.  $n \times n$  matrix multiplication  $O(n^3)$  i.e. 64K ops  
 $64/4 = 16$ K cycles → 16us
4. Total Time =  $200 + 16 = 216\mu\text{s}$
5. Peak Computation =  $64K/216\mu\text{s} = \sim 300\text{MFlops}$

**More computation – 1 vs 64K → 10MFlops vs 300MFlops**

Repeated references to the same data item corresponds to what type of locality??

TEMPORAL

In our example,  $O(n^2)$  data accesses and  $O(n^3)$  computation. Such asymptotic difference makes the above example particularly desirable for caches.

Data Reuse Is Critical For Cache Performance.

# Bandwidth

21

- What is it?
  - Bandwidth → Throughput → How much data can move on the wire
  - Objective: Maximize bandwidth
- Dot-Product Example:
  - Bus is 4x wider: 4 words (128bits) instead of 1 word (32bits)
  - Latency → 100ns
  - Assuming that the vectors are laid out linearly in memory, eight FLOPs (four multiply-adds) can be performed in 200 cycles.
  - 8 FLOPS / 200 cycles = **40MFLOPS** (vs 10MFLOPS for a 1 word bus)
- In practice, wide buses are expensive to construct.
- In a more practical system, consecutive words are sent on the memory bus on subsequent bus cycles after the first word is retrieved.

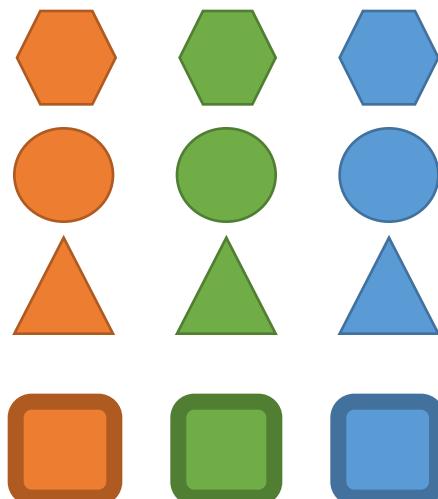
# Bandwidths and Data Layout

22

Consider the following code fragment:

```
for (i = 0; i < 1000; i++)  
    column_sum[i] = 0.0;  
    for (j = 0; j < 1000; j++)  
        column_sum[i] += b[j][i];
```

The code fragment sums columns of the matrix *b* into a vector *column\_sum*.



How can we improve this?

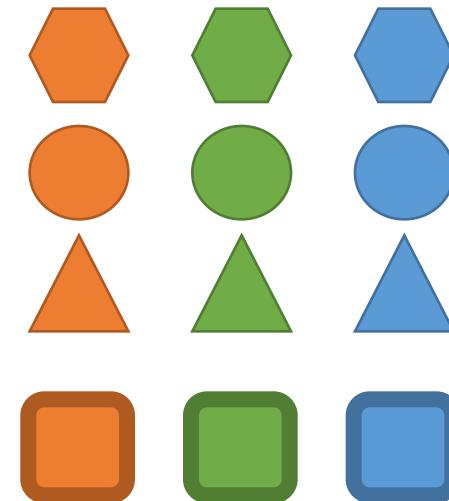


SPATIAL LOCALITY

# Variant:

We can fix the above code as follows:

```
for (i = 0; i < 1000; i++)  
    column_sum[i] = 0.0;  
  
for (j = 0; j < 1000; j++)  
    for (i = 0; i < 1000; i++)  
        column_sum[i] += b[j][i];
```



In this case, the matrix is traversed in a row-order and performance can be expected to be significantly better.

**DEMO**

- The series of examples presented in this section illustrate the following concepts:
  - Exploiting spatial and temporal locality in applications is critical for amortizing memory latency and increasing effective memory bandwidth.
  - The ratio of the number of operations to number of memory accesses is a good indicator of anticipated tolerance to memory bandwidth.
  - Memory layouts and organizing computation appropriately can make a significant impact on the spatial and temporal locality.

MEMORY

IMPLICIT PARALLELISM

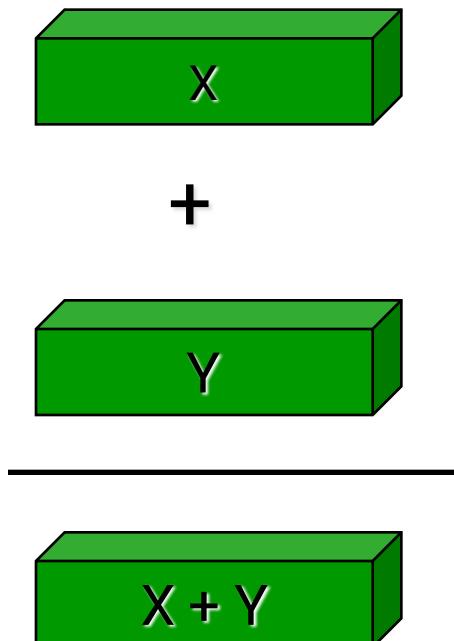
PIPELINING

# Scalar vs Vector Operations

26

## Scalar processing

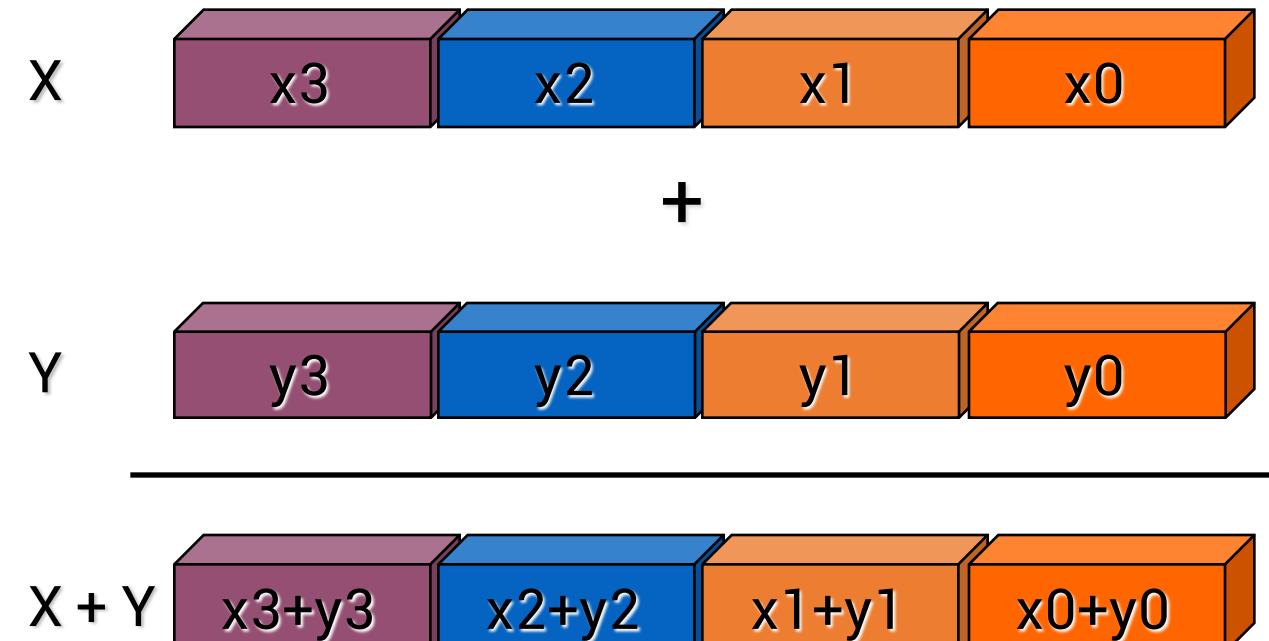
- traditional mode
- one operation produces one result



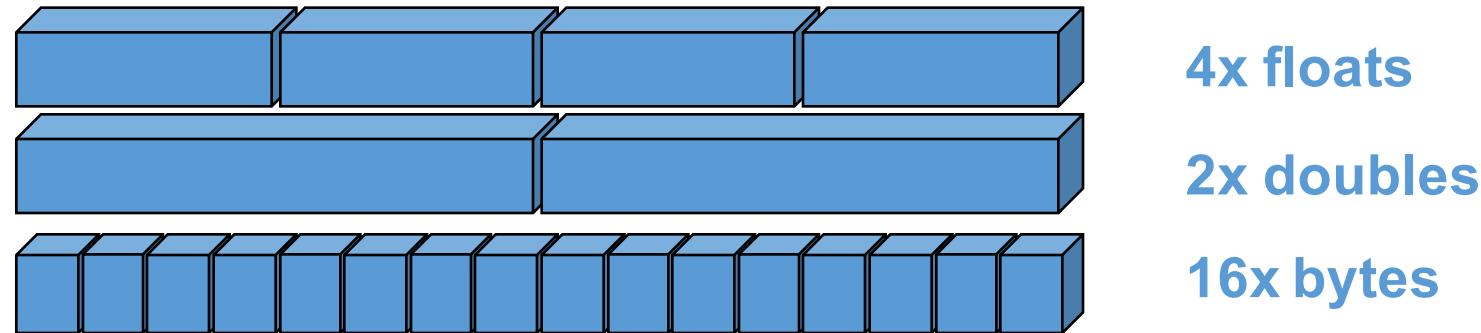
## SIMD → Single Instruction Multiple Data

### SIMD processing

- with SSE2/SSE3/SSE4/AVX
- SSE = streaming SIMD extensions
- one operation produces multiple results



- SSE2 data types: anything that fits into 16 bytes, e.g.,



- Instructions perform add, multiply etc. on all the data in this 16-byte register in parallel
- Challenges:
  - Need to be contiguous in memory and aligned
  - Some instructions to move data around from one part of register to another
- Similar on GPUs, vector processors (but many more simultaneous operations)

In addition to SIMD extensions, the processor may have other special instructions

- Fused Multiply-Add (FMA) instructions:

$$x = y + c * z$$

is so common some processor execute the multiply/add as a single instruction, at the same rate (bandwidth) as + or \* alone

In theory, the compiler understands all of this

- When compiling, it will rearrange instructions to get a good “schedule” that maximizes pipelining, uses FMAs and SIMD
- It works with the mix of instructions inside an inner loop or other block of code

But in practice the compiler may need your help

- Choose a different compiler, optimization flags, etc.
- Rearrange your code to make things more obvious
- Using special functions (“intrinsics”) or write in assembly 😞

# MEMORY

# IMPLICIT PARALLELISM

# PIPELINING

# What is pipelining

30

