HIGH PERFORMANCE COMPUTING FOR SCIENCE AND ENGINEERING

LECTURE 1

Fabian Wermelinger

Harvard University

CS205

Tuesday, January 25th 2022

TODAY

- CS205 introduction
  - Teaching staff
  - Important information
- Introduction to parallel computing
  - Why do we need it?
  - Computer characterization
  - Basic Terminology
  - Classification of parallel computers
- Brief summary of reading for next class

TEACHING STAFF

- ***Head instructor:*** Fabian Wermelinger (PhD, ETH Zürich)
- ***Research interests:*** Fluid Mechanics, compressible multiphase flows, high performance computing, data compression, software development
- ***Office:*** SEC 1.312-02 (fabianw@seas.harvard.edu)

Teaching Fellows:

- Erick Ruiz
- eruiz@g.harvard.edu

- Jiahui Tang
- jiahuitang@g.harvard.edu

- Javiera Astudillo
- jastudillo@g.harvard.edu

- Yuxin (Iris) Ye
- yye@g.harvard.edu

SYLLABUS

- ***Syllabus:*** https://harvard-iacs.github.io/2022-CS205/pages/syllabus.html The class provides an introduction to parallel programming techniques and programming models for applications in science, engineering as well as data science. We focus on shared memory and distributed memory programming paradigms which are the most useful paradigms when developing software targeted for high performance computing (HPC) platforms.
- ***Schedule and deadlines:*** The class schedule can be found here. *All deadlines are listed in the schedule, it is your responsibility to meet them.*
- ***Prerequisites:*** You should be comfortable with the `C` and/or `C++` programming languages. No thorough knowledge is required, examples in class, homework and labs will be presented in `C++`. You can find lecture slides for a `C/C++` primer class at this link: https://github.com/Harvard-IACS/c_cpp_primer.

SYLLABUS

- ***Class format:*** https://harvard-iacs.github.io/2022-CS205/pages/syllabus.html#course-format

Homework (5 total)	40%
Lab (6 total)	10%
Quiz (4 total)	10%
Project	35%
Mailing list contributions	5%
Bonus	5%

- ***Homework:*** Homework solutions (and lab solutions) are submitted as a single `zip` or `tar` archive on Canvas. Written homework (non-code) is expected to be typeset using LaTeX. See the `README.md` files distributed with the handouts.
- ***Homework late days and regrading policy:*** You have ***3 late days*** for the 5 homeworks. In case of grading errors, homeworks can be regraded once.

  Homework will be regarded in full which may result in a higher or lower grade.

SYLLABUS

- ***Textbooks:*** The class does not follow a specific textbook. The topics discussed in class are covered in the following texts:
  - *"Introduction to High Performance Scientific Computing"*,
    V. Eijkhout, free pdf 3rd edition 2020
  - *"Parallel Programming for Science and Engineering"*,
    V. Eijkhout, free pdf 2nd edition 2020
  - *"An Introduction to Parallel Programming"*,
    P. Pacheco, Morgan Kaufmann 2011
  - *"Introduction to High Performance Computing for Scientists and Engineers"*,
    G. Hager and G. Wellein, CRC Press 2011
  - *"Computer Organization and Design"*,
    D. Patterson and J. Hennessy, Morgan Kaufmann 2018 (RISC-V edition)
  - *"Computer Architecture"*,
    J. Hennessy and D. Patterson, Morgan Kaufmann 2019
  - *"Programming Massively Parallel Processors"*,
    D. Kirk and W. Hwu, Morgan Kaufmann 2017

SYLLABUS

- ***Project:*** https://harvard-iacs.github.io/2022-CS205/pages/project.html
  The CS205 project objective is to apply the techniques learned in class on a concrete real application to gain hands-on experience with writing and optimizing parallel code.
  The final report is written in the form of a proposal (in $\LaTeX$) that could be submitted to a high performance computing center to request compute hours on a supercomputer.
- The project should be oriented on a compute/data intensive problem, either from your research area or build something from scratch that interests you (given that it fits into the time frame of the class)
- Project groups are teams of *3-4 students*, you free to choose your team mates. Ideally you have similar interests. Use the class mailing list to exchange ideas and look for team mates.
- ***Milestones:*** There are 5 milestones further described on the project homepage. The first is dedicated to the team formation and is due on February 8th.

IMPORTANT INFORMATION

You can find the following information on the class website as well:
https://harvard-iacs.github.io/2022-CS205/#important

- ***Canvas:*** https://canvas.harvard.edu/courses/100601
  Administrative matters for the class are carried out through Canvas (e.g. grades). Solutions for homework and labs are submitted on Canvas in a `zip` or `tar` archive. *See the class schedule for all deadlines.*

- ***Class material:*** https://code.harvard.edu/CS205/main
  All handouts are distributed to the class through the main class `git` repository, hosted at the link above.

IMPORTANT INFORMATION

You can find the following information on the class website as well: https://harvard-iacs.github.io/2022-CS205/#important

- ***Class material:*** https://code.harvard.edu/CS205/main

  All handouts are distributed to the class through the main class `git` repository, hosted at the link above.

  You must be a member of the CS205 organization on https://code.harvard.edu to clone and pull from this repository. You can request membership by sending an email to cs205-staff@lists.fas.harvard.edu (include your Harvard NetID in the body of the email. ***Note:*** your NetID is also your username on https://code.harvard.edu). *Please use your `.harvard.edu` email address when sending email to the address above. Other email addresses will be rejected*. Please contact me directly if you are attending from another school.

IMPORTANT INFORMATION

- ***Class mailing list:*** https://harvard-iacs.github.io/2022-CS205/#class-mailinglist
  CS205 will use a mailing list for all questions and knowledge exchange regarding the class. *We will not use Piazza, why:*
  - Piazza is no longer entirely free to use and ads suck!
  - Email is a powerful communication tool (often used poorly, we practice in CS205)
  - You can easily archive all class communication in your email client and keep it forever
  - You can reply to the list and the original poster *or* establish a private communication by replying to the original poster only
  - Main communication in open-source software projects (Linux kernel, `git`, `python`, etc.)
  - The class mailing list is cs205@lists.fas.harvard.edu

  Before you can send emails to the list, you must sign-up by sending a (blank) email to cs205-join@lists.fas.harvard.edu. *You will receive an email where you are asked to confirm your registration by simply replying to email.* ***Use the email address associated with your HarvardID.***
- ***Mailing list etiquette:*** Please see this link for information on how to post on the list.
- ***Teaching staff only:*** cs205-staff@lists.fas.harvard.edu (use your `.harvard.edu` email)

MAILING LIST EXAMPLE

- Assume you have a question related to homework 1 and you want to post it to the mailing list. Your post might look like this:

  John Doe has a question for homework 1. He prefixes `[HW1]` to his subject and writes:

```
From: John Doe <john.doe@harvard.edu>
To: cs205@lists.fas.harvard.edu
Subject: [HW1] Implicit barrier at end of omp parallel
         for loop?

Dear All,

I am writing a #pragma omp parallel for loop and am
not sure whether there is an implicit barrier at the
end of the loop.  I could not find the answer in the
documentation.

Thanks,
JD
```

  Jane Smith knows the answer and clicks the "Reply" button in her email client:

```
From: Jane Smith <jane.smith@harvard.edu>
To: CS205 class mailing list <cs205@lists.fas.harvard.edu>
Subject: Re: [cs205] [HW1] Implicit barrier at end of omp
             parallel for loop?

On Sun, 09 Jan 2022 17:15:32 +0000, John Doe wrote:
>I am writing a #pragma omp parallel for loop and am not
>sure whether there is an implicit barrier at the end of
>the loop.

There is an implicit barrier if you haven't specified the
nowait clause.

Best,
JS
```

  Jane received the message from the list, which always prepends `[cs205]` to the subject line of messages. Her email client adds the standard `Re:` for a reply.

- You can view the mailing list archive online at https://web.lists.fas.harvard.edu
- Your posting frequency on the mailing list is worth 5% of the final grade.

CLASS POLICIES

- ***Attendance:*** It is expected that when you decide to take the class that you also attend the lectures as well as the labs. These are core parts of the class and therefore mandatory to attend. The 5% bonus outlined in the grading section can only be exploited via lecture attendance.
  https://harvard-iacs.github.io/2022-CS205/pages/syllabus.html#attendance-policy
- ***Collaboration:*** You are welcome to discuss the course material and homework with others in order to better understand it, but the work you turn in must be your own (with exception of the project where collaborative work is permitted). Any work that is not your own, without properly citing the original author(s), is considered plagiarism. Failure to follow the academic integrity and dishonesty guidelines outlined in the Harvard Student Handbook will have an adverse effect on your final grade.
  https://harvard-iacs.github.io/2022-CS205/pages/syllabus.html#collaboration-policy

ROADMAP

Wk	Tuesday	Thursday	Labs	Events
1(4)	***Lecture 1:*** *2022-01-25* • Class introduction/organization • Moore's Law • Transistor density and power limit • Parallel computing • Flynn's taxonomy • Overview of parallelism treated in class: DLP, ILP, TLP, shared memory and distributed memory	***Lecture 2:*** *2022-01-27* • Computer architecture • von Neumann architecture • Memory pyramid • Linux process anatomy • Introduction to compute cluster: access, job submission • ***Reading*:** Leiserson paper	***Sign-up:*** Select one of the offered lab session days according to your schedule	***Note:*** The "***Reading***" assignments are relevant for the lecture and due ***on the day of the lecture!*** Questions may be asked to individual students. 1. Doodle for lab day selection due (2022-01-28)
2(5)	***Lecture 3:*** *2022-02-01* • Cache memories: why are they there, how they work • Cache lines and the 3 C's • What is temporal and spatial locality • Cache associativity: fully, n-way, direct mapped • Memory access patterns (differences row-major / column-major)	***Lecture 4:*** *2022-02-03* • Shared memory introduction • Examples of concurrency and concurrent memory access • Why is shared memory programming hard: what is a race condition and why/how does it happen • ***Quiz 1***	***Lab 1:*** Accessing cluster, SLURM, Linux, compiler and C++ tutorials.	1. HW1 release (2022-02-01)
3(6)	***Lecture 5:*** *2022-02-08* • Memory model for shared memory programming and its implications on compilers • Sequential consistency • Mutual exclusion / critical sections / locks • Overview of thread libraries	***Lecture 6:*** *2022-02-10* • Introduction to OpenMP: why OpenMP and how to use it in new or existing codes • OpenMP: fork/join parallel regions • OpenMP: work sharing constructs • OpenMP: data environment • ***Reading*:** OpenMP specification 5.2 Chap. 1 (until 1.4 inclusive)		1. Lab 1 due (2022-02-11) 2. Project team formation due (2022-02-08)

ROADMAP

Wk	Tuesday	Thursday	Labs	Events
4(7)	***Lecture 7:*** *2022-02-15* • OpenMP: synchronization constructs • OpenMP: library routines • OpenMP: environment variables • OpenMP: processor binding	***Lecture 8:*** *2022-02-17* • UMA/NUMA memory architectures • What is cache coherency and why is it required in shared memory programming • Cache coherency protocols (focus on MESI) • False sharing • ***Quiz 2***	***Lab 2:*** OpenMP locks, critical sections and atomic clauses.	1. HW1 due (2022-02-15) 2. HW2 release (2022-02-15)
5(8)	***Lecture 9:*** *2022-02-22* • Performance analysis (single node) • Relationship of compute performance (flop) to memory bandwidth • Roofline model • ***Reading***: Williams paper	***Lecture 10:*** *2022-02-24* • Introduction to distributed programming (recap Flynn's taxonomy) • What is the Message Passing Interface (MPI) • Simple parallel MPI program example	***Lab 3:*** False sharing and cache thrashing.	1. Lab 2 due (2022-02-25) 2. Project high-level description due (2022-02-22)
6(9)	***Lecture 11:*** *2022-03-01* • MPI: blocking point-to-point • MPI: blocking collective • ***Reading***: MPI 4.0 Standard 3.1, 3.2, 3.4, 3.5	***Lecture 12:*** *2022-03-03* • MPI: non-blocking point-to-point • MPI: non-blocking collective • ***Reading***: MPI 4.0 Standard 3.7		1. Lab 3 due (2022-03-04)
7(10)	***Lecture 13:*** *2022-03-08* • MPI: I/O file management • MPI: I/O read and write routines • Parallel I/O for data compression example	***Lecture 14:*** *2022-03-10* • Hybrid MPI and OpenMP • Overhead associated with sending messages • Message packing • Working with scientific libraries (BLAS/LAPACK/Eigen) • ***Quiz 3***	***Lab 4:*** MPI reductions and scans.	1. HW2 due (2022-03-08) 2. HW3 release (2022-03-08)

ROADMAP

Wk	Tuesday	Thursday	Labs	Events
8(11)	*Spring break:* 2022-03-15	*Spring break:* 2022-03-17		
9(12)	*Presentations for project proposals:* 2022-03-22	*Presentations for project proposals:* 2022-03-24		1. Lab 4 due (2022-03-25) 2. Project proposals due
10(13)	***Lecture 15:*** *2022-03-29* • Parallel scaling analysis • Strong scaling / Amdahl's law • Weak scaling	***Lecture 16:*** *2022-03-31* • Instruction set architecture (ISA) / RISC / CISC • Assembly language (`x86_64`) • Processor pipelining (ILP) • ***Reading:*** Hennessy and Patterson Turing lecture	***Lab 5:*** Linking your code with third party libraries. Examples for BLAS and LAPACK.	1. HW3 due (2022-03-29) 2. HW4 release (2022-03-29)
11(14)	***Lecture 17:*** *2022-04-05* • Recap Flynn's taxonomy: SIMD • Instruction set architecture extensions • What is vectorization and why is it important • Memory alignment and relation to cache lines	***Lecture 18:*** *2022-04-07* • Manual vectorization • Intel intrinsics • Bit masking/shuffling • Examples for manual vectorization and performance impact (DLP in roofline)		1. Lab 5 due (2022-04-08)
12(15)	*Presentations for project designs:* 2022-04-12	*Presentations for project designs:* 2022-04-14		1. Project designs due

ROADMAP

Wk	Tuesday	Thursday	Labs	Events
13(16)	***Lecture 19:*** *2022-04-19* • Compiler auto vectorization • SPMD programming model • Intel ISPC compiler • ***Reading***: Pharr paper • ***Quiz 4***	***Lecture 20:*** *2022-04-21* • GPU computing I: ▪ Streaming processors ▪ Main difference between CPU and GPU architectures ▪ SIMD and SIMT ▪ Introduction to CUDA	***Lab 6:*** Understanding machine instructions by learning how to debug code.	1. HW4 due (2022-04-19) 2. HW5 release (2022-04-19)
14(17)	***Lecture 21:*** *2022-04-26* • GPU computing II (CUDA): ▪ CUDA warps and threads ▪ Streaming multiprocessor and Little's Law ▪ Example CUDA kernel for vector addition • Class summary	***Reading period:*** *2022-04-28*		1. HW5 due (2022-05-01) 2. Lab 6 due (2022-04-29)
15(18)	***Reading period:*** *2022-05-03*	***Exam period:*** *2022-05-05*		
16(19)	***Exam period:*** *2022-05-10* • Project final presentations (date TBD)	***Exam period:*** *2022-05-12* • Project final presentations (date TBD)		1. Project deliverables due 2. Project final presentations due

WHAT I WISH THAT YOU GET OUT OF CS205

1. ***Think*** parallel and know parallel computing
2. Understand fundamental computer hardware and its link to performance
3. Foundations of shared memory (OpenMP) and distributed memory (MPI) paradigms
4. Performance analysis and optimization options
5. Enjoy writing code and appreciate large computing systems

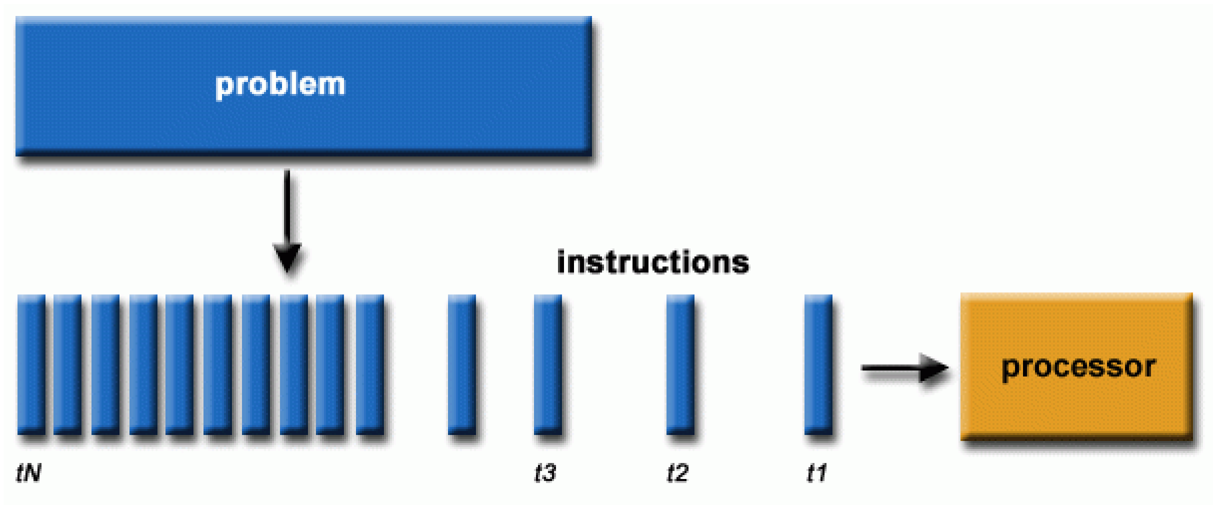
WHAT IS COMPUTING?

> ***Computing*** *is a domain of knowledge dealing with the study of information processing, both* ***what*** *can be computed and* ***how*** *to compute it.*
>
> *Joseph Sifakis*

- ***information*** is data
- ***processing*** is transformation

Traditionally, the information is transformed by a single processor which we call ***serial*** computing.

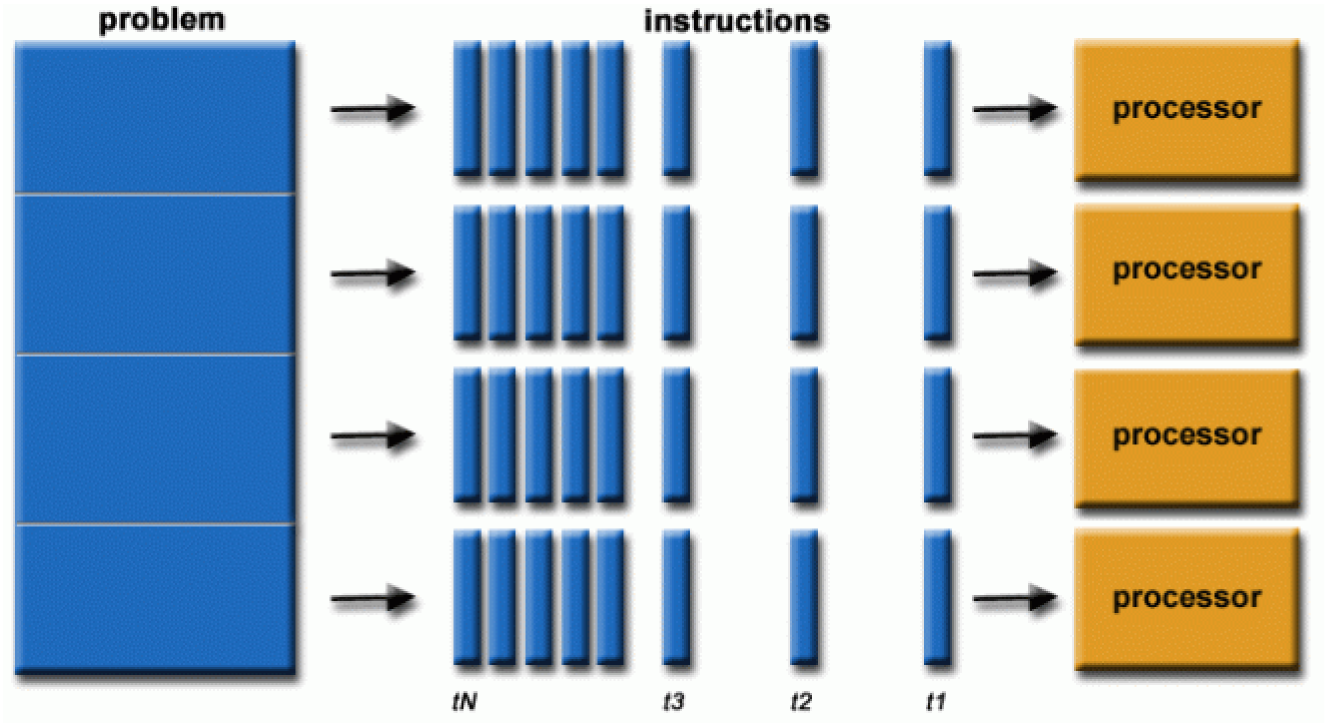
SERIAL COMPUTING



https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial

- A serial program consists of a series of *instructions*
- Instructions are processed in *sequential* order (one after the other)
- They are executed on a single *processor* (entity that can perform work)
- Only *one* instruction can be executed at any time

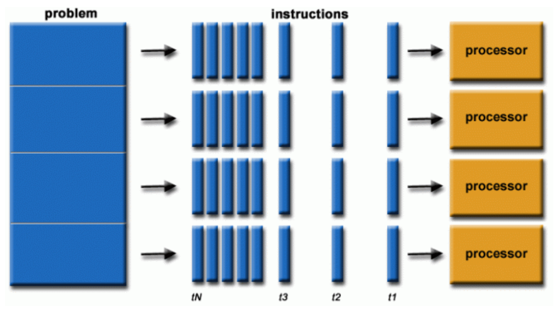
PARALLEL COMPUTING



https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial

- A problem is divided into discrete chunks that can be solved concurrently
- Each chunk is processed in sequentially, but many chunks run simultaneously on different processors
- Requires a control/communication/coordination mechanism

PARALLEL COMPUTING



- A problem is divided into discrete chunks that can be solved concurrently

  Can any problem be divided into such discrete chunks (trivially)?

  - Programs = Algorithms + Data Structures
  - Both, ***algorithms*** and ***data structures*** need be suitable for dividing a problem into discrete chunks
  - An algorithm may be inherently sequential, offering little or no parallelism
  - There might be *data dependencies* that limit parallel execution

PARALLEL COMPUTING

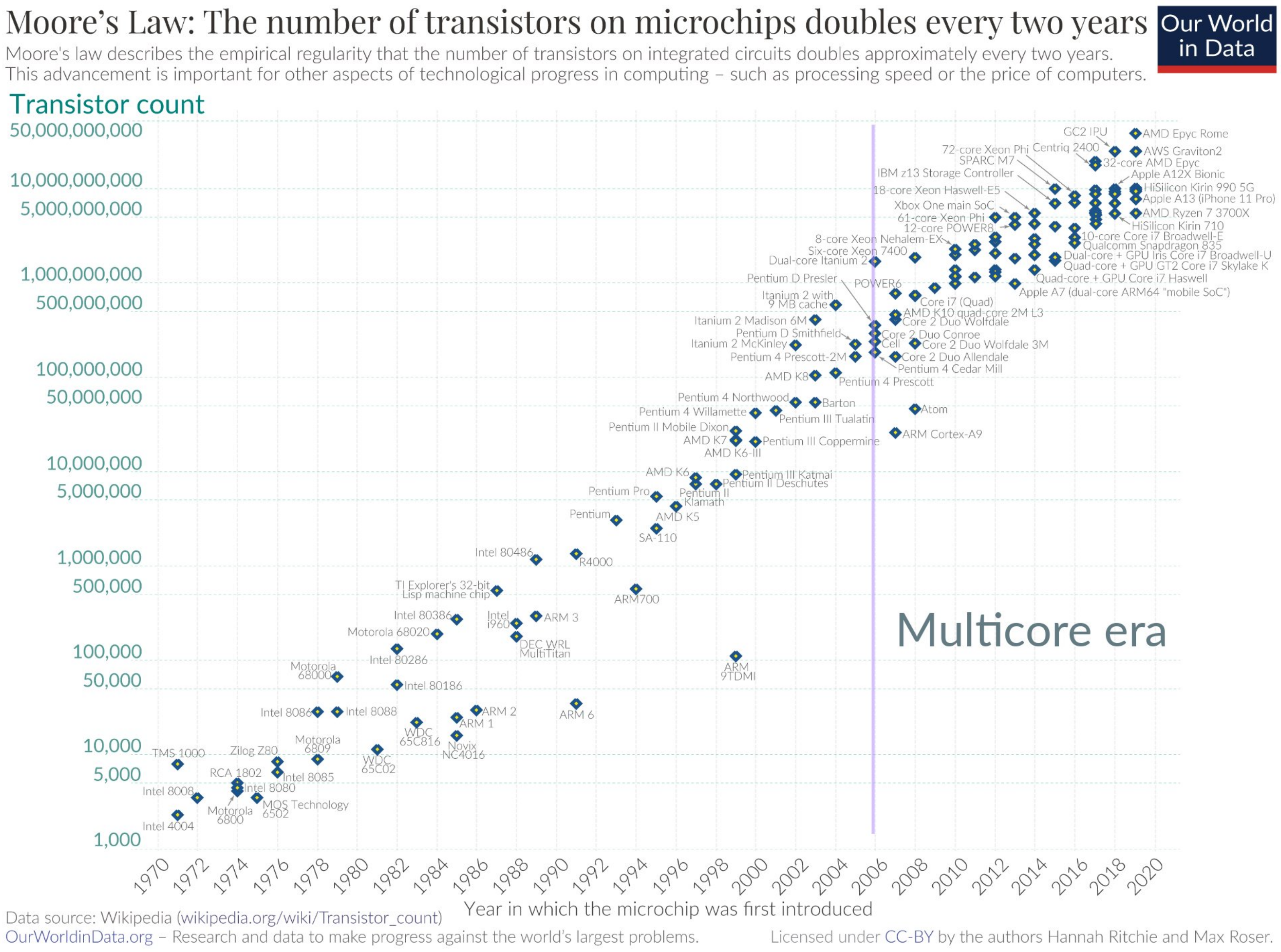
We should be able to break down a computational problem into:

- discrete pieces of work that can be solved simultaneously (the size of these pieces determine the *parallel granularity* of the problem)
- the discrete pieces should allow to execute multiple instructions at any moment in time
- the problem should be solved faster than with a single processor

The most typical compute resources are:

- A single entity with multiple processors such as CPUs, GPUs, TPUs, etc. We call such an entity a ***node***. *Examples:* laptop, desktop, cell phone, Raspberry Pi
- An arbitrary number of nodes connected by a local network. We call such an entity a ***cluster***. *Examples:* supercomputers, cloud (with local network)

TECHNOLOGY TRENDS: MICROPROCESSOR CAPACITY


Moore's Law: The number of transistors on microchips doubles every two years

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.

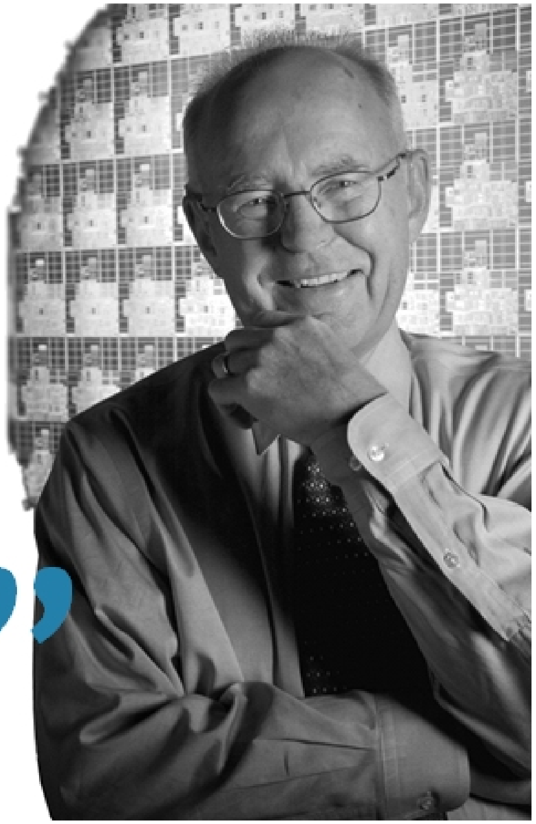
Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)

OurWorldinData.org – Research and data to make progress against the world's largest problems.

MOORE'S LAW

“Frankly, I didn't expect to be so precise.”

Gordon Moore
Intel co-founder and author of Moore's law



Moore's Law states that integrated circuit resources double every 18-24 months. It resulted from a 1965 prediction of such growth in IC capacity made be Gordon Moore, one of the founders of Intel.

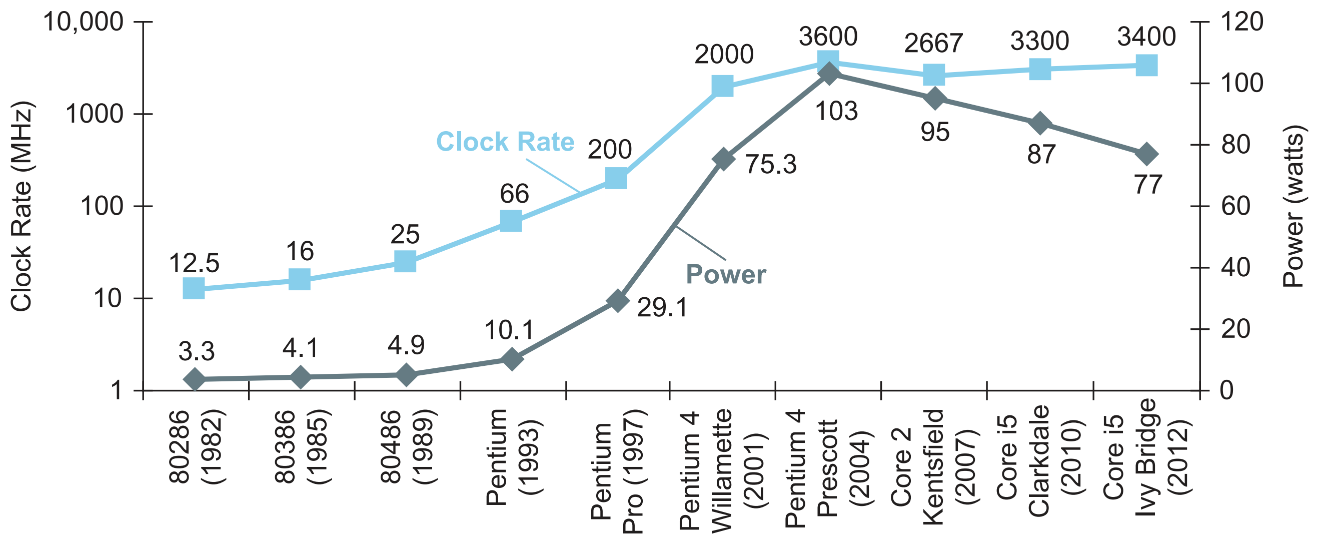
- Every two years you get a ~2x performance boost without doing anything in your code
- Achieved by reducing the transistor size and therefore place more transistors on the dye (increase transistor density)

Can this go on forever?

THE POWER WALL

- Moore's Law in principle can grow like that
- The problem when transistors get smaller is a different one:
  - Dominant technology for integrated circuits is CMOS (*complementary metal oxide semiconductor*)
  - For CMOS the primary source of energy consumption is when the transistor switches states:

$$\text{Power} \propto \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency switches}$$

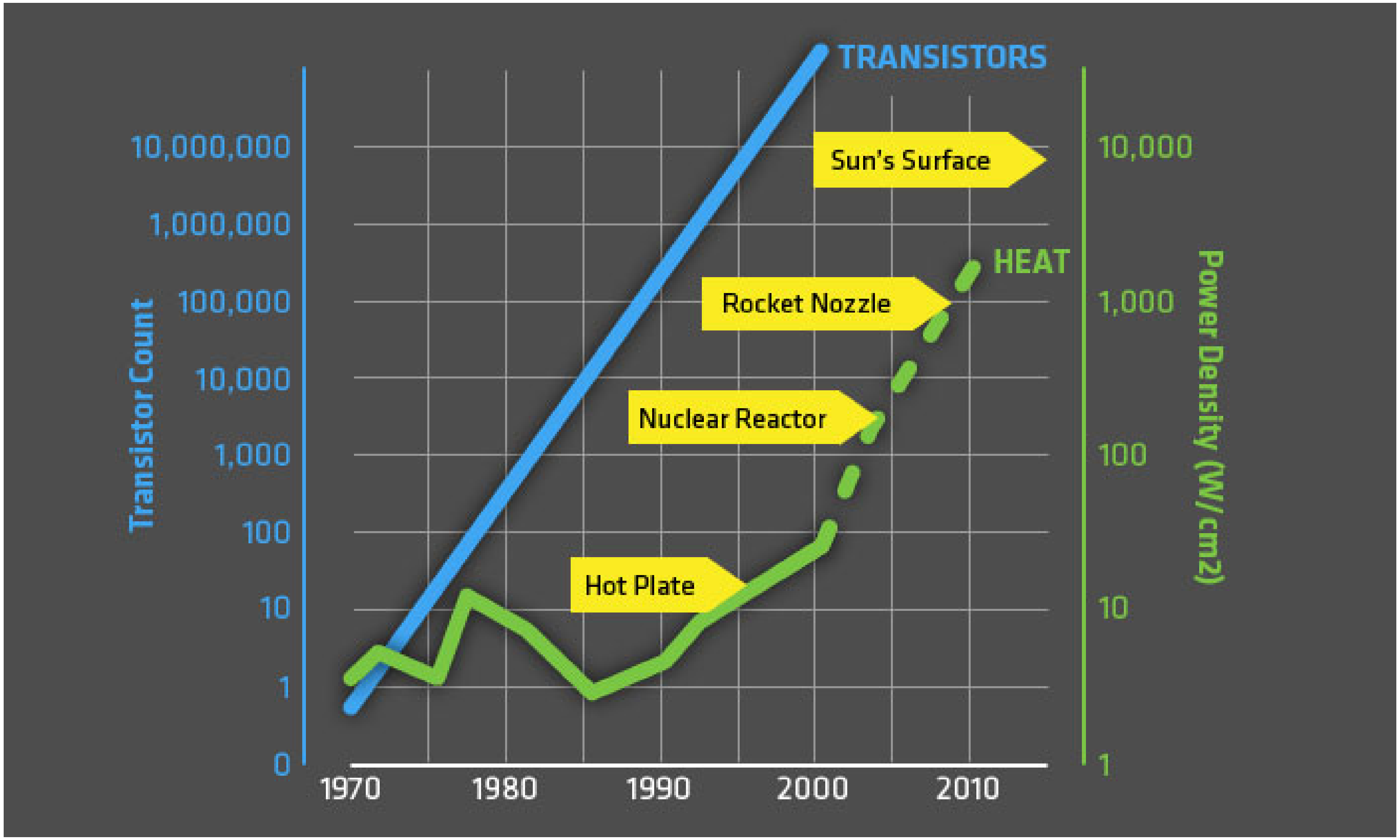


Patterson and Hennessy, 2017

THE POWER WALL

- The *capacitive load* is related to the number of transistors on the chip
- The *switch frequency* is related to the clock rate
- How can the power consumption only grow by 30x while the clock rate increased by 1000x?
- *Voltage is reduced by about 15% per generation (from 5V to 1V)*
- Lowering the voltage makes transistors more *leaky* (dissipate energy, even today about 40% of power consumption is due to leakage)
- Two alternatives to satisfy Moore's Law in the next generations (as transistor density reaches limit):
  1. Increase clock rate but must invest in sophisticated cooling solutions ***(can run current code without modification)***
  2. Operate at the power wall but increase number of processor cores ***(can not run current code without modification)***

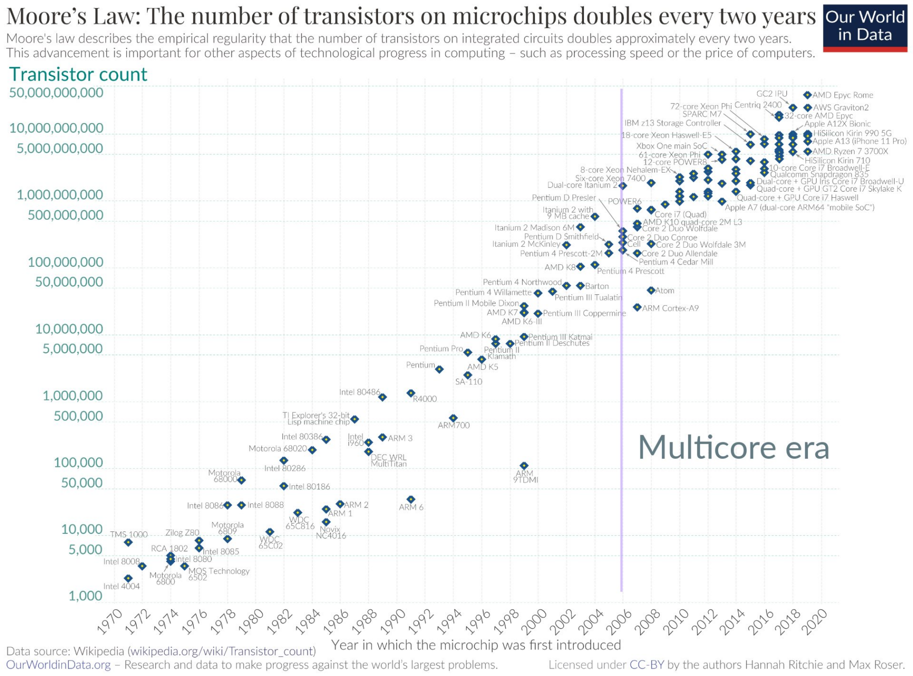
TRANSISTORS AND HEAT



Power density is a function of the number of transistors on the chip. ***Cooling large computer systems is associated with huge cost!*** Source article: https://cns.utexas.edu/news/researchers-tackle-the-dark-side-of-moore-s-law

MULTICORE ERA

- Since 2006 all microprocessor manufacturers ship *multicore* chips
- *Moore's Law now is:* double the number of cores per microprocessor generation about every 24 months



Moore's Law reinterpreted:

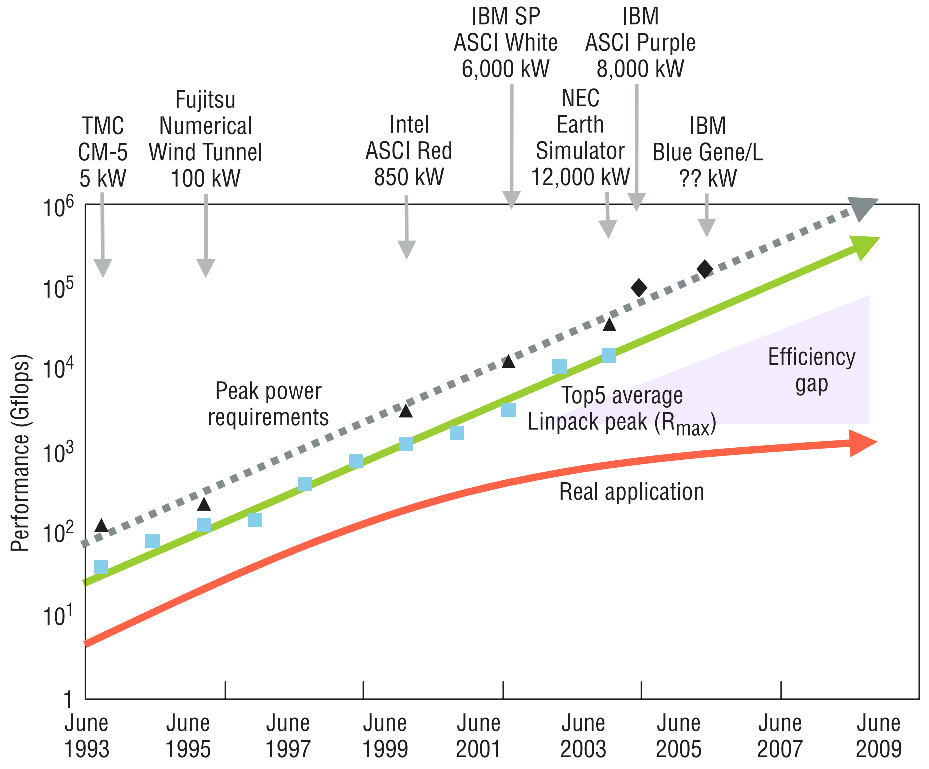
- Number of *cores* per chip double every two years
- Clock speed will *not* increase (possibly decrease)
- Need to deal with systems with Millions of concurrent threads (GPUs)
- Need to deal with *intra*-chip ***and*** *inter*-chip parallelism (shared memory, distributed memory)

WHY USE PARALLEL COMPUTING?

- We have no choice
- All major processor vendors produce *multicore* chips
- Do existing programs have to be rewritten? ***Yes*** (most have been since 2006)
- Will all programmers have to be parallel programmers?
  - There are software models that try to hide the complexity. It is difficult to gain full ROI with these models because hiding the complexities and being generic comes at a cost (e.g. OpenCL, OpenACC)
  - You could entirely rely on such software models instead
  - The better approach is to understand how parallel architectures really work (learning high-level software abstraction layers after you understood it is trivial)

WHY USE PARALLEL COMPUTING?

The problem is illustrated in this graph:



Cameron et al., IEEE 2005

- Hardware performance is following Moore's Law in the parallel era
- Software needs to be designed for parallel architectures
- Gap in domain knowledge (programmers lack sufficient parallel programming knowledge) generates an ***efficiency gap*** where energy, performance and money is wasted

One of the learning outcomes of CS205 is to address this gap.

WHO IS USING PARALLEL COMPUTING?

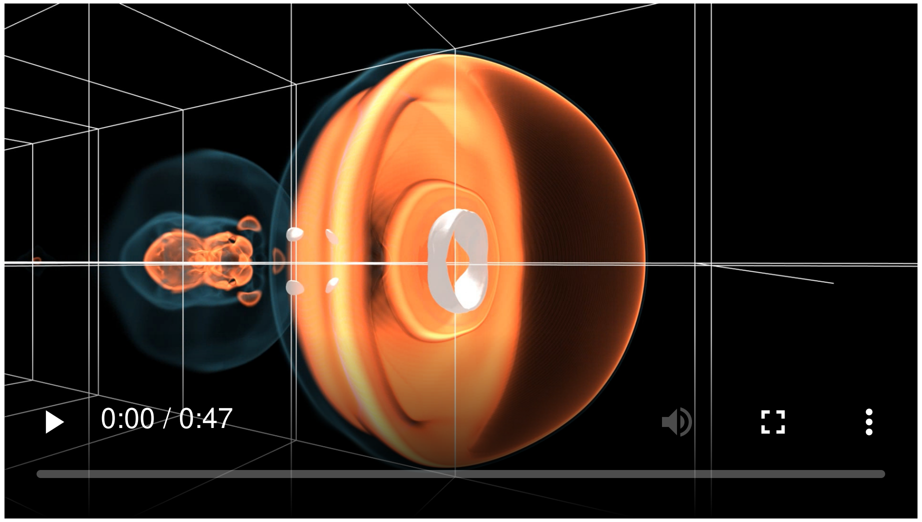
The real world is massively complex! Tackling these problems (in academia ***or*** industry) is not possible without going parallel.

(parallel computing was high-end science in the past, it is ubiquitous today!)

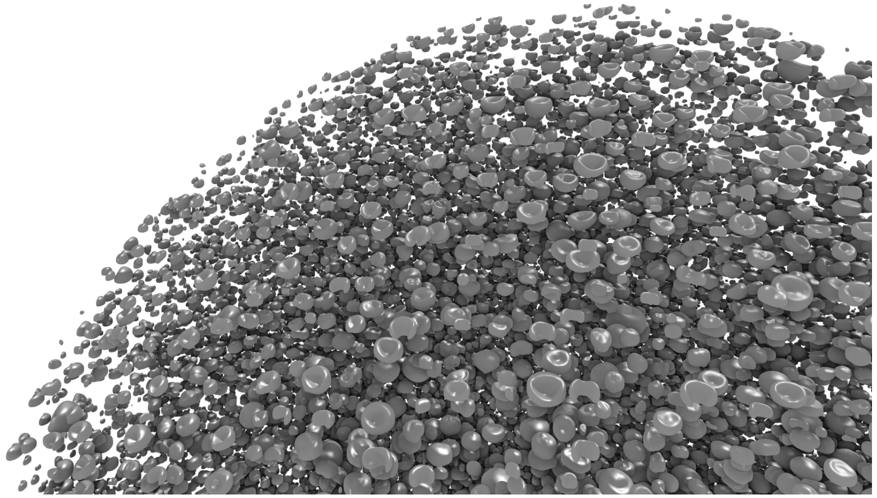
- Science and Engineering
  - Atmosphere, earth, environment, bioscience, biotechnology, genetics, chemistry, molecular sciences
  - Physics: applied, nuclear, particle, condensed matter, high pressure, fusion, quantum
  - Mechanical/Electrical Engineering: fluid mechanics, circuit design, microelectronics
- Industrial and commercial
  - "Big Data", databases, data mining, artificial intelligence (AI), oil exploration, web search engines
  - Medical imaging and diagnosis
  - Pharmaceutical design, financial and economic modeling

AIR BUBBLE COLLAPSE IN LIQUID WATER

Collapse of bubble array in water



Many bubbles collapse in a cloud



https://journals.aps.org/prfluids/abstract/10.1103/PhysRevFluids.4.063602

Microjet formation in bubble collapse and cloud cavitation

- Presence of bubbles nearby cause non-spherical collapse with formation of a high-energy microjet
- Impact on nearby surfaces of such microjets cause material erosion
- Used in cancer treatment (tissue penetration for precise drug placement), kidney stones, dental cleansing, mantis shrimp to stun prey, marine applications

Cloud cavitation collapse with 12'500 bubbles can not be investigated without parallel computing. *232 Billion computational cells at 12 Million core hours.*

REASONS FOR USING PARALLEL COMPUTING

- Save time and money
- Solve larger / more complex problems
  - Many problems are so large or complex that solving them in serial is not possible (especially from the memory perspective)
  - Examples: Grand challenge problems
- Provide concurrency
  - A single compute resource can only do one thing at a time. Multiple compute resources can do many things simultaneously
- Take advantage of non-local resources
  - Must be able to work on a remote machine and exploit remote resources
- Make better use of underlying parallel hardware
  - Modern computers, even laptops, are parallel in architecture with multiple processors/cores
  - Parallel software is specifically intended for parallel hardware with multiple cores, threads, etc.
  - In most cases, serial programs run on modern computers "waste" potential computing power

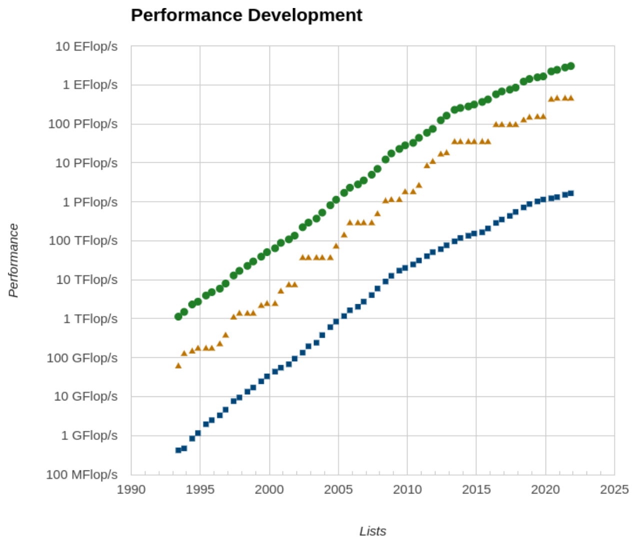
REASONS FOR USING PARALLEL COMPUTING

The Future

- In the past 20+ years, trends indicated by ever faster networks, distributed systems, and multi-processor computer architectures (even at the desktop level) clearly show that ***parallelism is the future of computing***
- In this same time period, there has been a greater than ***500'000x*** increase in supercomputer performance, with no end currently in sight.
- We are entering the Exascale era: https://www.exascaleproject.org (One Exaflop is 10^{18} operations per second 😅)

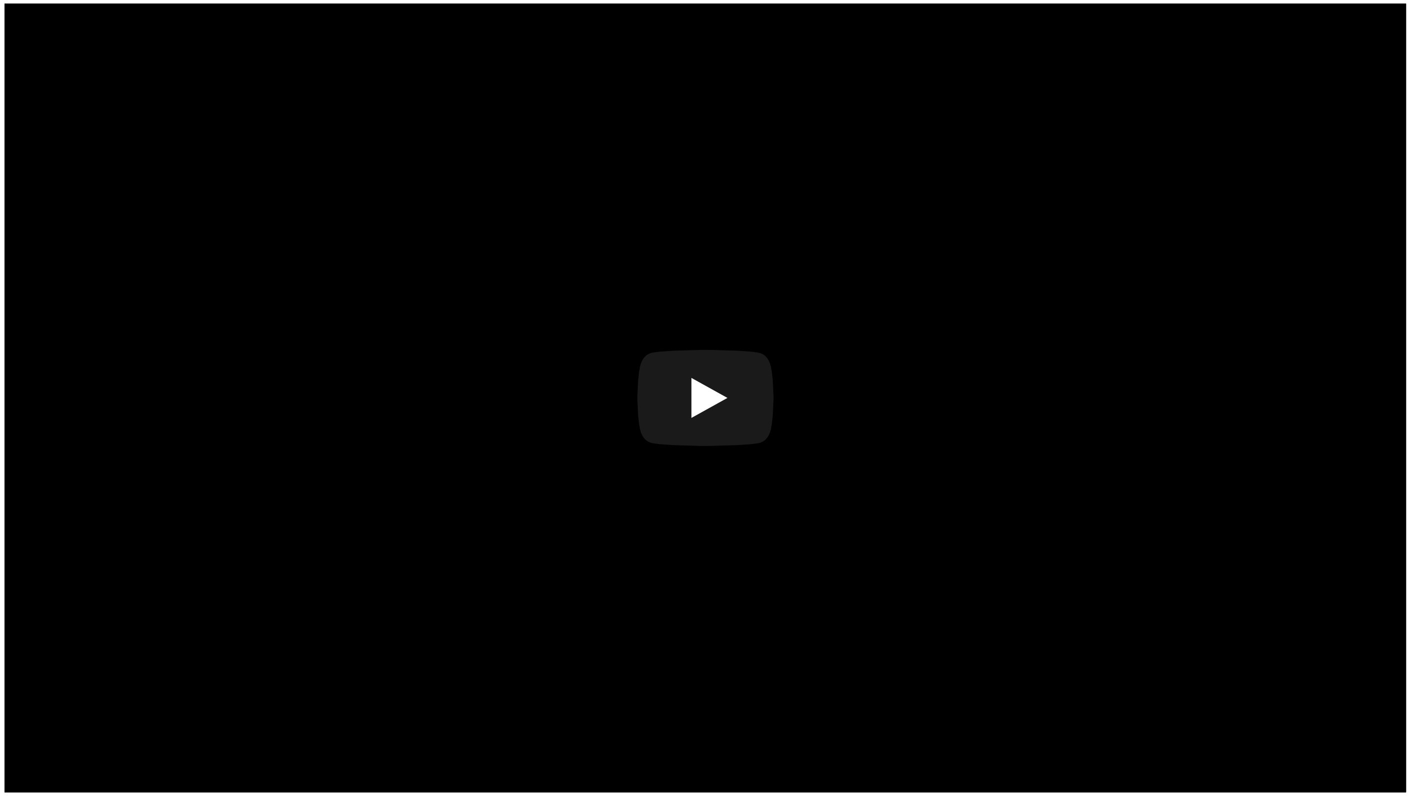
Fastest supercomputer in the world

Fugaku (Top500)



Fastest supercomputer in the world performs about 0.4 Exaflop/s

WHAT IS A SUPERCOMPUTER?



https://www.youtube.com/watch?v=9M99STmu-vI

TERMINOLOGY

This is an overview of commonly used terminology io parallel computing. We will discuss most of them in detail later in the class.

Node/Cluster

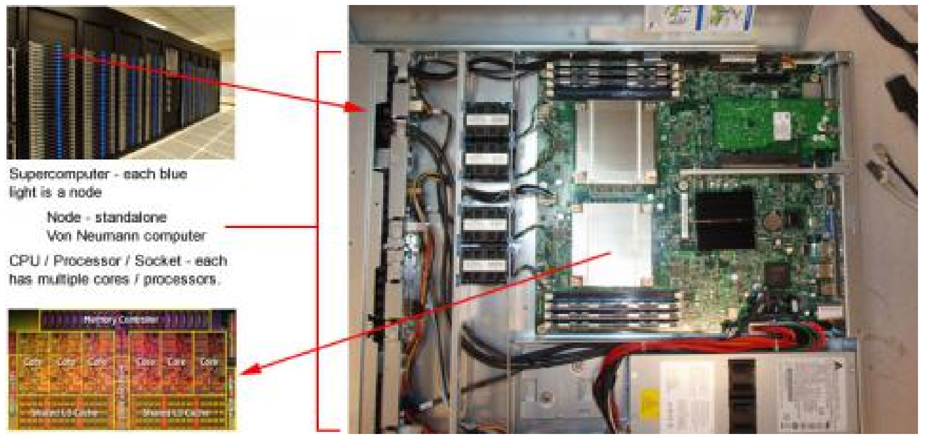
A standalone "computer in a box." Usually comprised of multiple CPUs/processors/cores, memory, network interfaces, etc. Nodes are networked together to comprise a supercomputer or cluster.

CPU/Processor/Core

A CPU (*central processing unit*) is an entity that performs data transformations in a *core*. A core therefor processes data and is sometimes called a *processor*. Recent CPUs have multiple cores and are sometimes called multi-processors.

Task

A logically discrete section of computational work. A task is typically a program or program-like set of instructions that is executed by a processor. A parallel program consists of multiple tasks running on multiple processors.


Supercomputer - each blue light is a node
Node - standalone Von Neumann computer
CPU / Processor / Socket - each has multiple cores / processors.
Memory Controller
Core
Core
Core
Core
Core
Core
Shared L3 Cache
Shared L3 Cache

The internals of a typical node on a supercomputing cluster

Socket

A socket is where a CPU is mounted on the motherboard. Usually there is one but there can be multiple. Access to memory modules is routed through sockets.

Pipelining

Breaking a task into steps performed by different *processor units*, with inputs streaming through, much like an assembly line; a type of parallel computing.

TERMINOLOGY

Shared memory

Describes a computer architecture where all processors have direct (usually bus based) access to ***common physical memory***. In a programming sense, it describes a model where parallel tasks all have the ***same "picture" of memory*** and can directly address and access the same logical memory locations regardless of where the physical memory actually exists.

Symmetric Multi-Processor (SMP)

Shared memory hardware architecture where multiple processors share a single address space and have equal access to all resources - memory, disk, etc.

Distributed memory

In hardware, refers to ***network based memory access*** for physical memory that is not common. As a programming model, *tasks can only logically "see" local machine memory* and must use ***communications*** to access memory on other machines where other tasks are executing.

Communications

Parallel tasks typically need to exchange data. There are several ways this can be accomplished, such as through a shared memory bus or over a network. The actual event of data exchange is commonly referred to as ***communication*** regardless of the method employed.

Synchronization

The coordination of parallel tasks in real time, very often associated with communications. Synchronization usually involves waiting by at least one task, and *can therefore cause a parallel application's wall clock execution time to increase*. Synchronization points *serialize* a parallel application.

TERMINOLOGY

Granularity

In parallel computing, granularity is a qualitative measure of the ratio of computation to communication.

- ***Coarse grained:*** relatively large amounts of computational work are done between communication events
- ***Fine grained:*** relatively small amounts of computational work are done between communication events

Observed speedup

Observed speedup of a code which has been parallelized, defined as:

$$S = \frac{T(1)}{T(p)} = \frac{\text{Wall-clock time of serial execution}}{\text{Wall-clock time of parallel execution}}$$

One of the simplest and most widely used indicators for a parallel program's performance.

Parallel overhead

The amount of time required to coordinate parallel tasks, as opposed to doing useful work. Parallel overhead includes:

- Task start-up time
- Synchronizations
- Data communications
- Software overhead imposed by parallel languages, libraries, operating systems, etc.
- Task termination time

TERMINOLOGY

Massively parallel

Refers to the hardware that comprises a given parallel system - having many processing elements. The meaning of "many" keeps increasing, but currently, the largest parallel computers are comprised of processing elements numbering in the hundreds of thousands to millions (e.g. GPUs).

Embarrassingly parallel

Solving many similar, but independent tasks simultaneously; *little to no need for coordination between the tasks.*

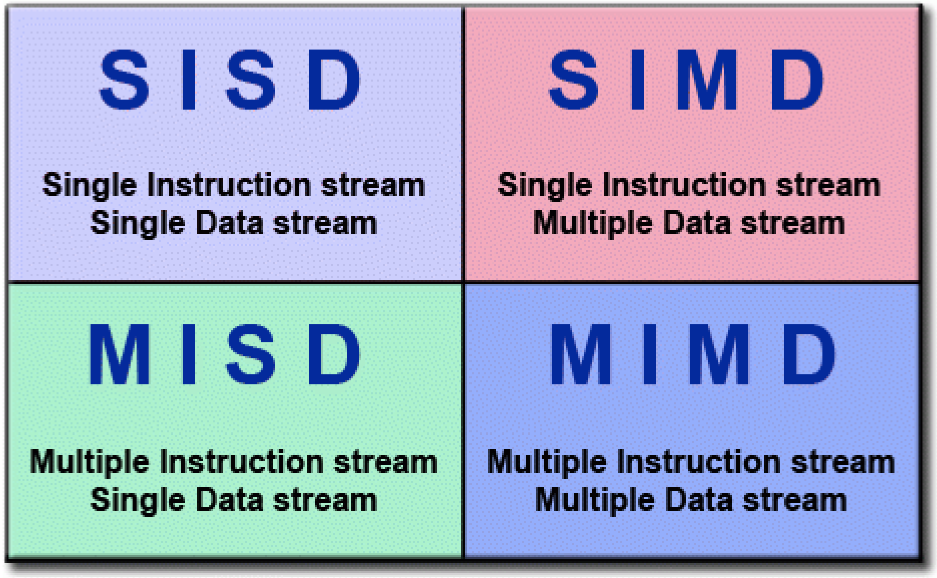
Scalability

Refers to a parallel system's (hardware and/or software) ability to demonstrate a proportionate increase in parallel speedup with the addition of more resources. Factors that contribute to scalability include:

- Hardware - particularly memory-cpu bandwidths and network communication properties
- Application algorithm
- Parallel overhead related
- Characteristics of your specific application and coding

FLYNN'S TAXONOMY

- A widely used ***classification of parallel computers*** is due to Flynn (1966)
- Flynn's taxonomy distinguishes multi-processor computer architectures according to how they can be classified along the two independent dimensions of ***Instruction Stream*** and ***Data Stream***. Each of these dimensions can have only one of two possible states: ***Single*** or ***Multiple***.

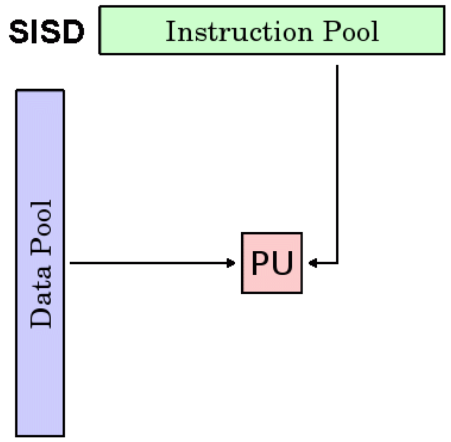

S I S D
Single Instruction stream
Single Data stream
S I M D
Single Instruction stream
Multiple Data stream
M I S D
Multiple Instruction stream
Single Data stream
M I M D
Multiple Instruction stream
Multiple Data stream

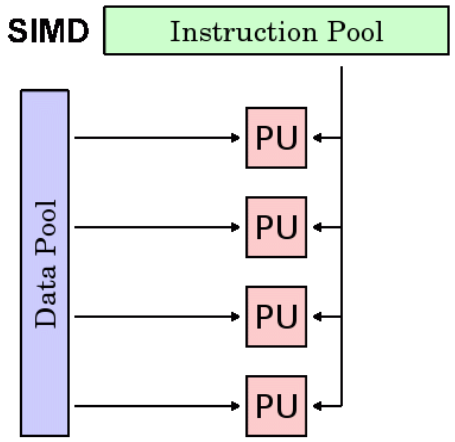
Flynn's taxonomy

FLYNN'S TAXONOMY

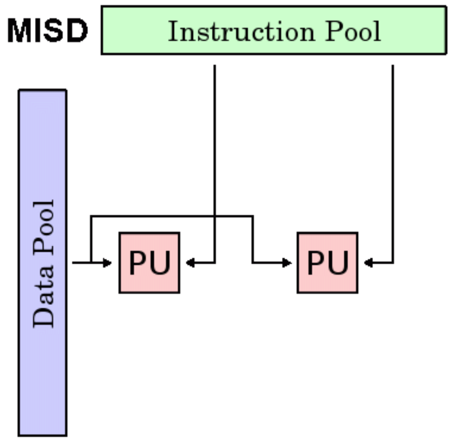
Single Data *Multiple Data*

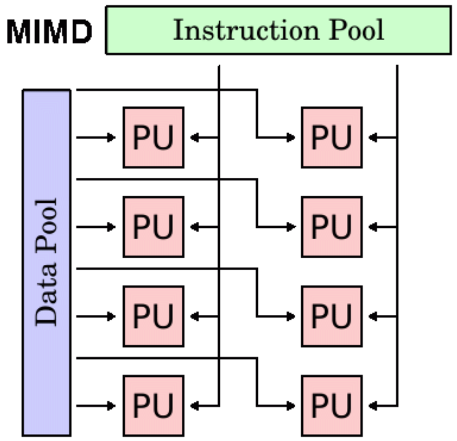
Single Instruction





Multiple Instruction

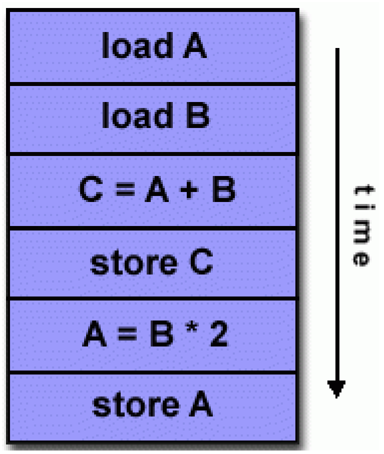




SISD

A serial (non-parallel) computer

- ***Single instruction:*** Only one instruction stream is being acted on by the CPU during any one clock cycle
- ***Single data:*** Only one data stream is being used as input during any one clock cycle
- Deterministic execution
- This is the oldest type of computer

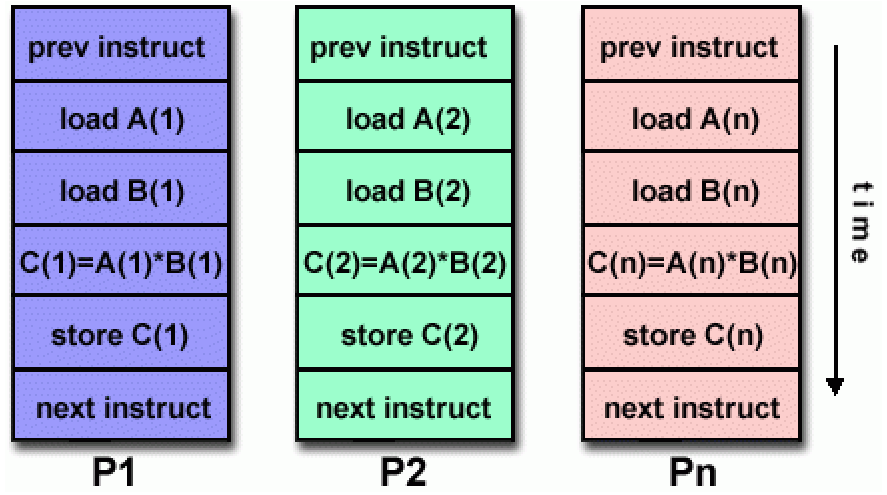


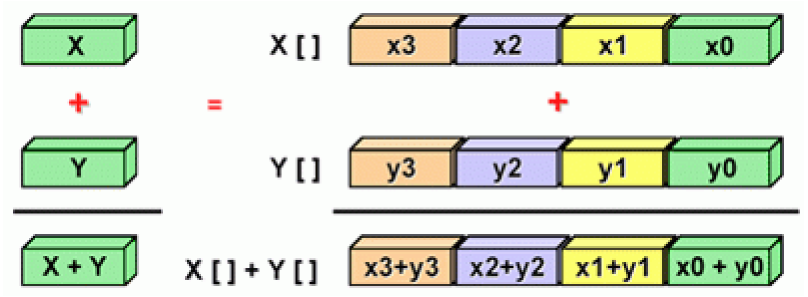
IBM 360

SIMD

A type of parallel computer (***treated in class***)

- ***Single instruction:*** All processing units execute the same instruction at any given clock cycle
- ***Multiple data:*** Each processing unit can operate on a different data element





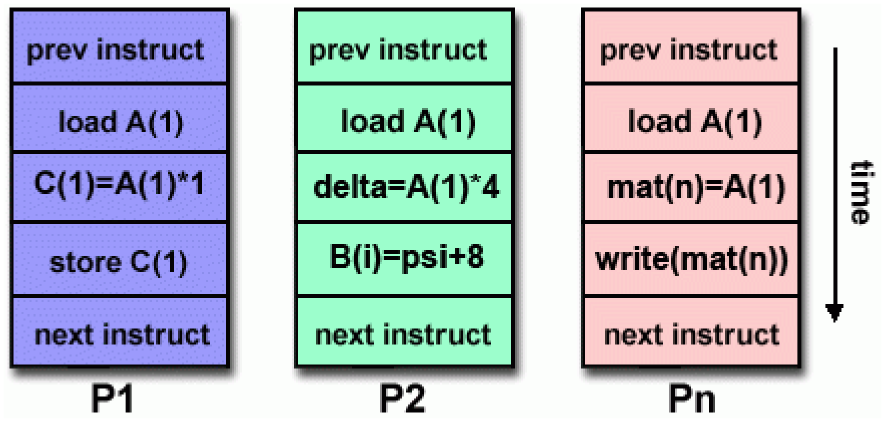
- Best suited for specialized problems characterized by a high degree of regularity, such as graphics/image processing
- Synchronous (lockstep) and deterministic execution
- *Two varieties:* Processor Arrays and Vector Pipelines

- Examples:
  - Processor Arrays: Thinking Machines CM-2, MasPar MP-1 & MP-2, ILLIAC IV
  - Vector Pipelines: IBM 9000, Cray X-MP, Y-MP & C90, Fujitsu VP, NEC SX-2, Hitachi S820, ETA10
- Most modern computers, particularly those with graphics processor units (GPUs) employ SIMD instructions and execution units

MISD

A type of parallel computer (***not treated in class***)

- ***Multiple instruction:*** Each processing unit operates on the data independently via separate instruction streams
- ***Single data:*** A single data stream is fed into multiple processing units

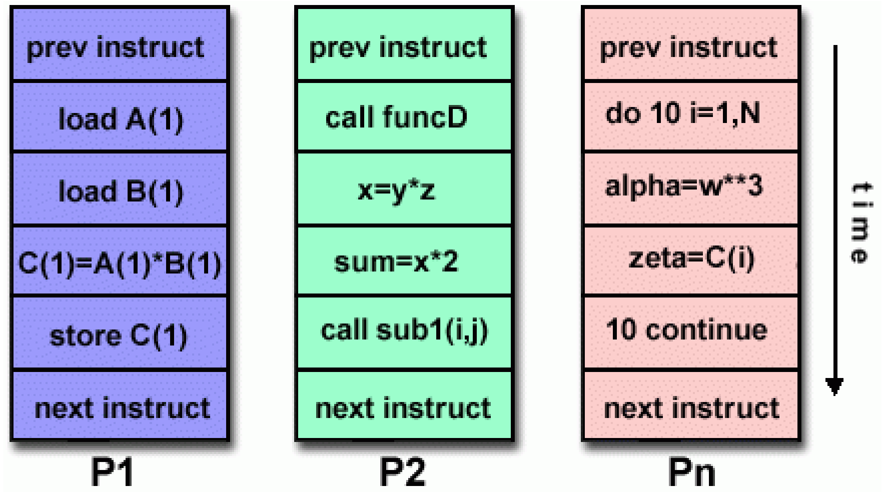


- Few (if any) actual examples of this class of parallel computer have ever existed
- Some conceivable uses might be:
  - Redundancy in critical missions (e.g. space)
  - multiple frequency filters operating on a single signal stream
  - multiple cryptography algorithms attempting to crack a single coded message

MIMD

A type of parallel computer (***treated in class***)

- ***Multiple instruction:*** Every processor may be executing a different instruction stream
- ***Multiple data:*** Every processor may be working with a different data stream

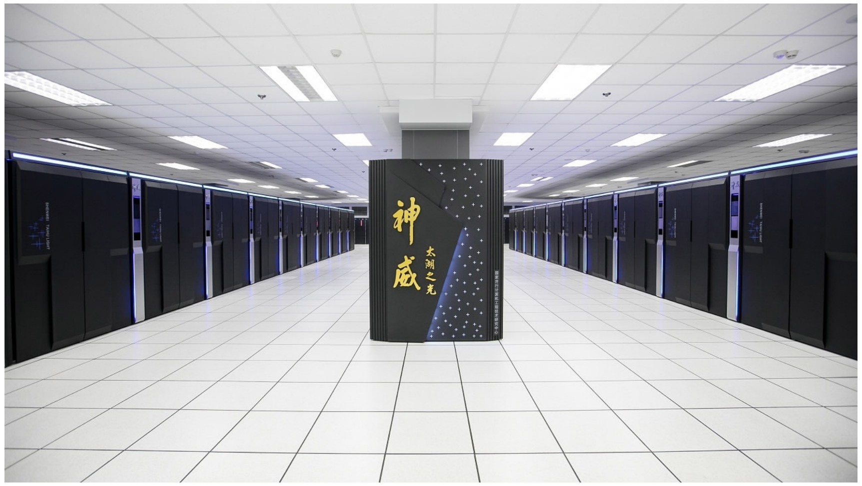


- Execution can be synchronous or asynchronous, deterministic or non-deterministic
- The ***most common type of parallel computer*** - most modern supercomputers fall into this category
- *Examples:* most current supercomputers, networked parallel computer clusters and "grids", multi-processor SMP computers, multi-core PCs
- ***Note:*** Many MIMD architectures also include SIMD execution sub-components

MIMD MACHINES

Piz Daint, CSCS Switzerland

Summit, ORNL United States



Sunway TaihuLight, NSCCWX China

Cannon, Harvard University United States

READING FOR NEXT CLASS

- You can find the reading for the next class (see schedule) in the class `git` repository.

  (https://code.harvard.edu/CS205/main/blob/master/reading/01_leiserson2020a.pdf)

- Try to make the following connections to today's class:
  - The relation to Moore's Law
  - Does this paper address the efficiency gap between hardware and software? If so, what are some of the technologies discussed to alleviate the gap?
  - Did you already know some of these technologies?
  - Do not worry if you do not understand everything the authors are talking about. We will address these topics in the lecture. The reading is motivational to highlight the current issues.
  - The take-away message is that you will waste a lot of resources if you do not consider all of the parallelism available on the hardware.

RECAP

- It is essential to know the hardware to get the best out of software
- You must find suitable parallelism in your application/algorithm
- Coordination and *synchronization*: data must be shared safely among processors
- Writing fast parallel programs is therefore harder than sequential. CS205 will provide you with the fundamentals. It is not very hard but you must practice as with all things in life.

Further reading:

- Chapter 1 in Pacheco, *An Introduction to Parallel Programming*, Morgan Kaufmann, 2011
- Cameron et al., *High-performance, power-aware distributed computing for scientific applications*, IEEE, 38(11):40-47, 2005
- https://www.energy.gov/science-innovation/science-technology/computing