为什么要并行计算？

任课教师：吴迪

课程内容

- 参考资料：
  - 并行程序设计导论，Peter S Pacheco，机械工业出版社，2016
  - CHAPTER 1 Why Parallel Computing?

Roadmap

- Why we need ever-increasing performance.
- Why we're building parallel systems.
- Why we need to write parallel programs.
- How do we write parallel programs?
- What we'll be doing.
- **Concurrent, parallel, distributed**!

Why we need ever-increasing performance

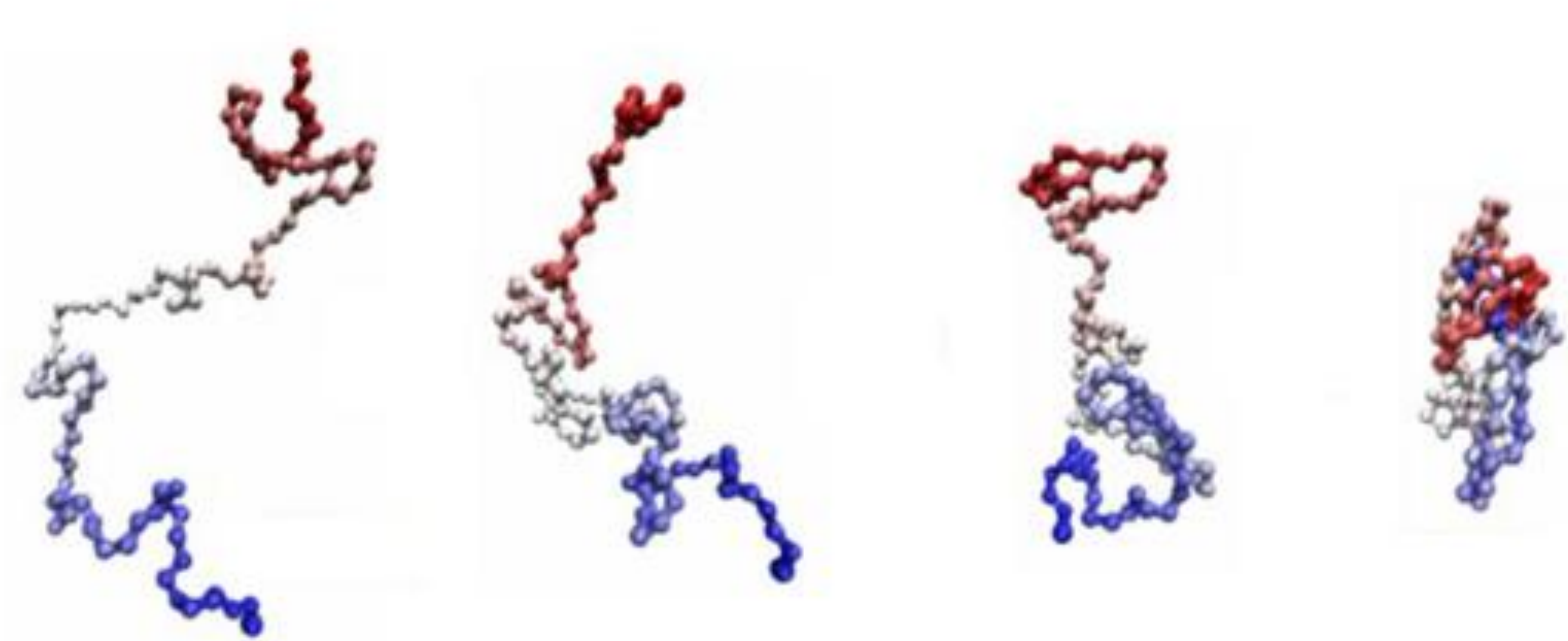
- Computational power is increasing, but so are our computation problems and needs.

- Problems we never dreamed of have been solved because of past increases, such as decoding the human genome.

- More complex problems are still waiting to be solved.

Climate modeling

Protein folding



Drug discovery

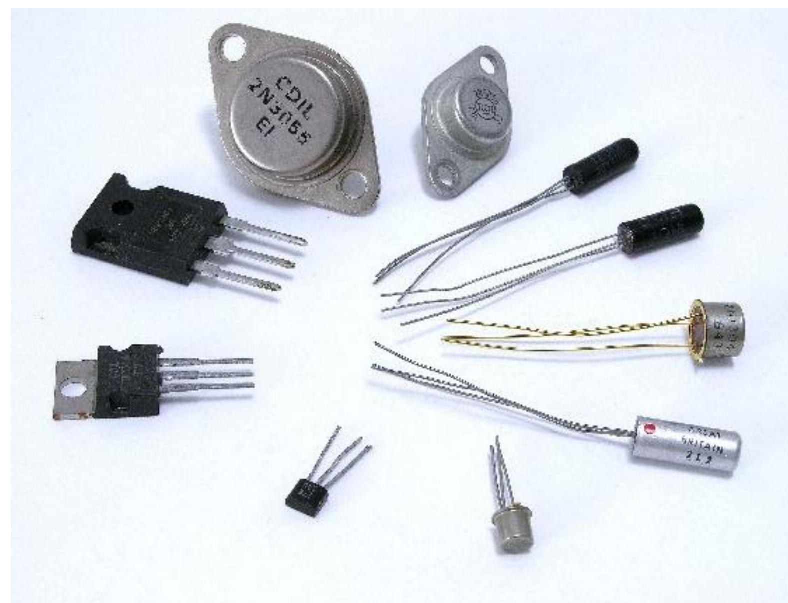
Energy research

Data analysis

Why we’re building parallel systems

- Up to now, performance increases have been attributable to increasing density of transistors.

- But there are inherent problems.



A Brief History of Processor Performance

A Brief History of Processor Performance

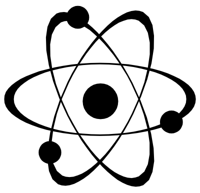
- **Wider data paths**
  - 4 it → 8 bit → 16 bit → 32 bit → 64 bit
- **More efficient pipelining**
  - e.g., 3.5 Cycles Per Instruction (CPI) → 1.1 CPI
- **Exploiting instruction-level parallelism (ILP)**
  - "Superscalar" processing: e.g., issue up to 4 instructions/cycle
  - "Out-of-order" processing: extract parallelism from instruction stream
- **Faster clock rates**
  - e.g., 10 MHz → 200 MHz → 3 GHz

A Brief History of Processor Performance

- From 1986 – 2002, microprocessors were speeding like a rocket, increasing in performance an average of 50% per year.

- Since then, it's dropped to about **20%** increase per year.

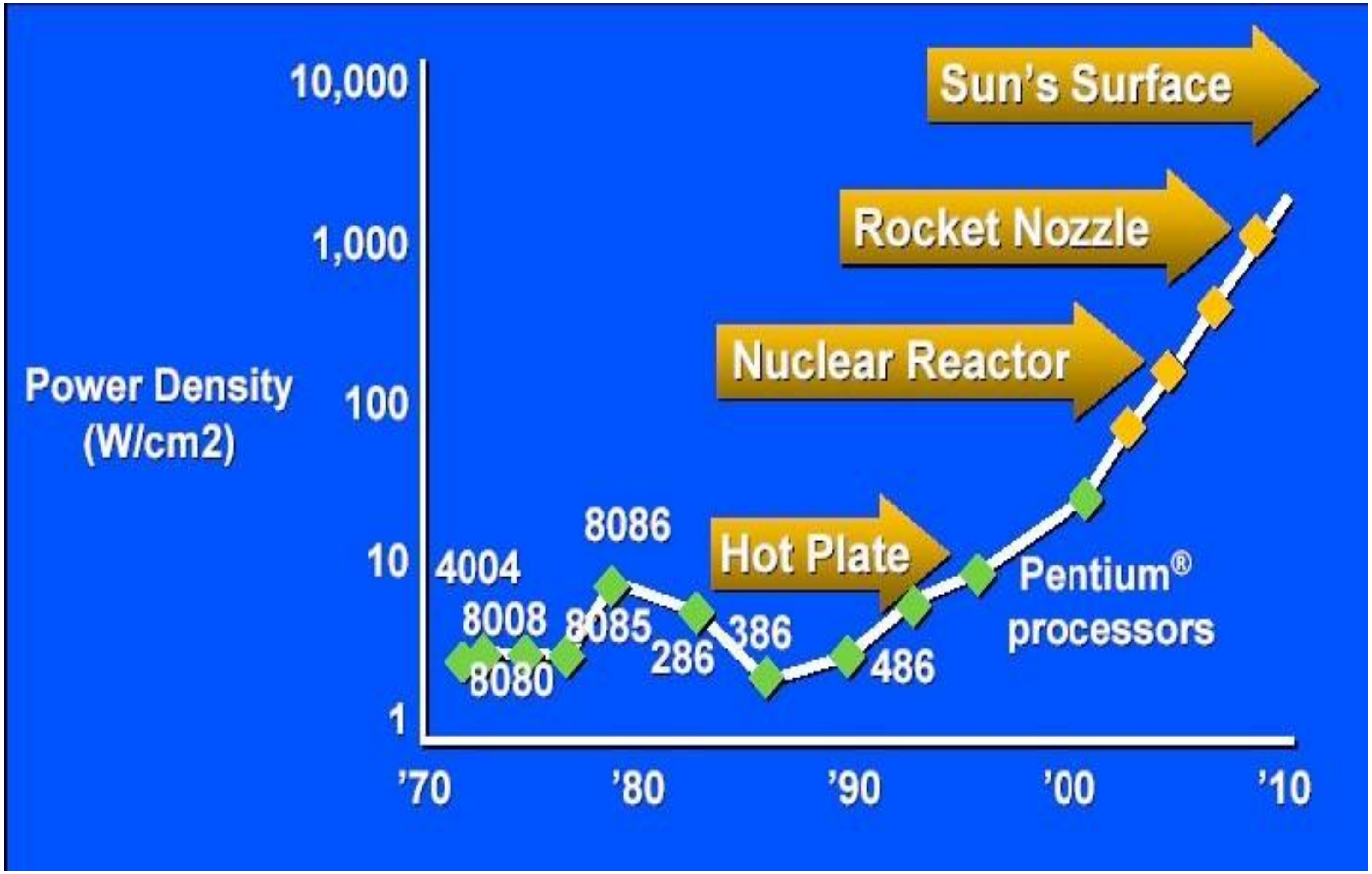
A little physics lesson



- Smaller transistors = faster processors.
- Faster processors = increased power consumption.
- Increased power consumption = increased heat.
- Increased heat = unreliable processors.

Intel hits the Power Density Wall

- Inflection point in 2004



Pat Gelsinger, ISSCC 2001

Intel's Big Shift After Hitting Technical Wall

- Intel ‘s newest microprocessor was running slower and hotter than its predecessor.
- Intel publicly acknowledged that it had hit a "thermal wall" on its microprocessor line.


The New York Times

TECHNOLOGY

TECHNOLOGY; Intel's Big Shift After Hitting Technical Wall

Give this article

By John Markoff

May 17, 2004

End of frequency scaling

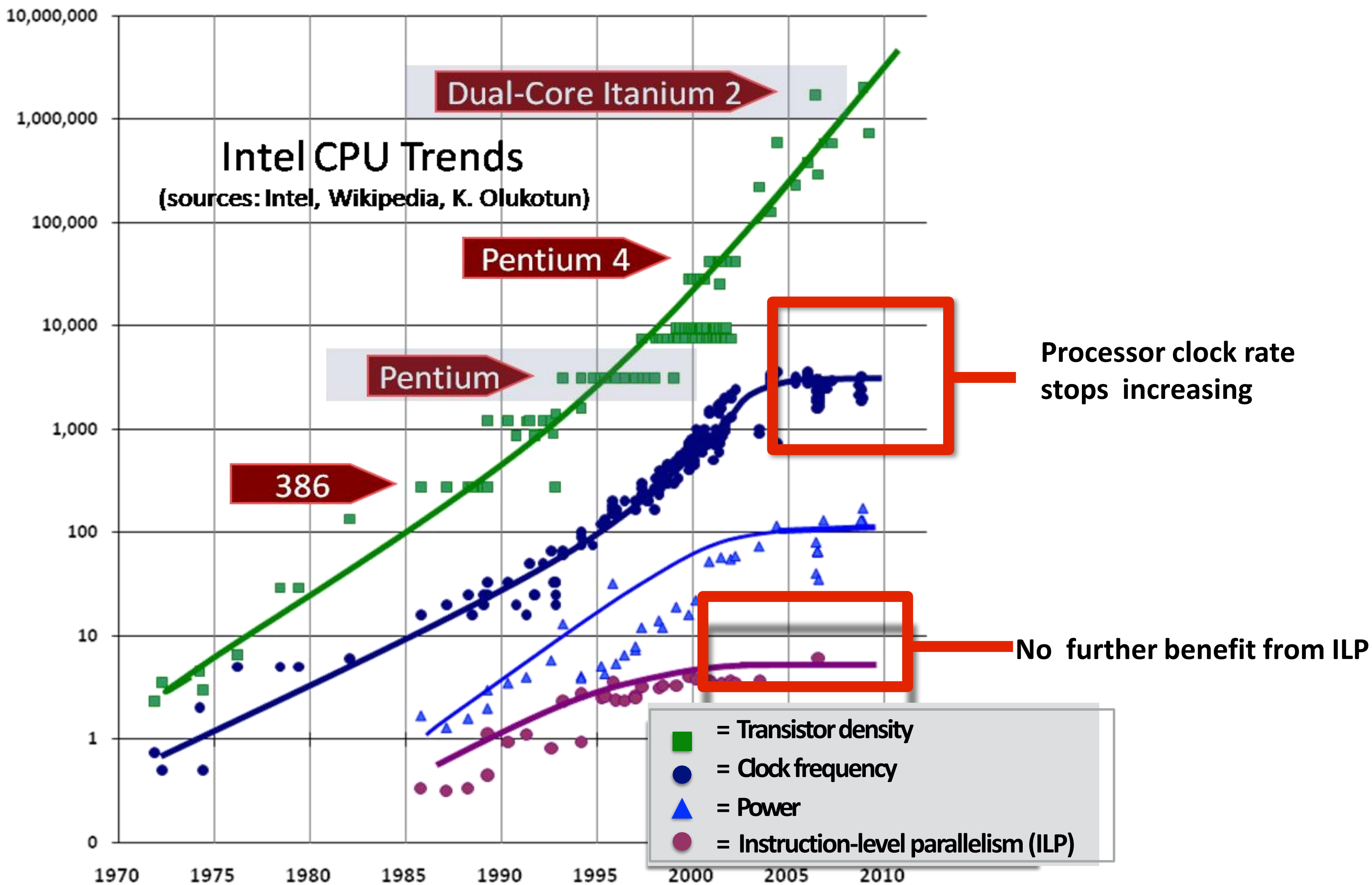
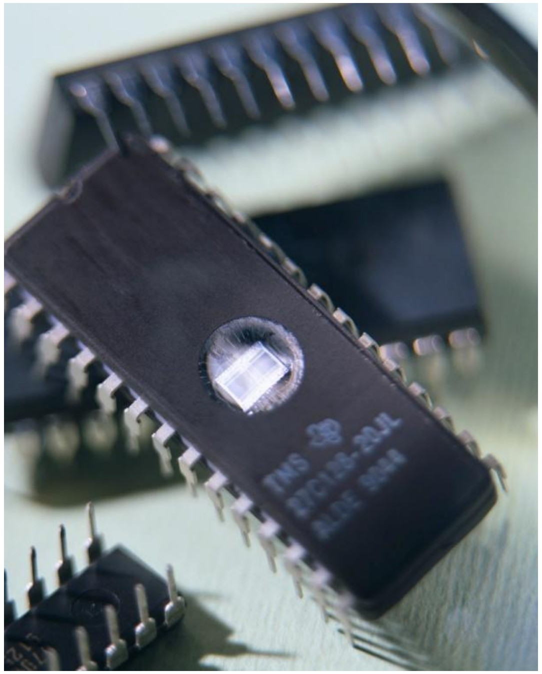


Image credit: “The free Lunch is Over” by Herb Sutter, Dr. Dobbs 2005

Solution

- Move away from single-core systems to **multicore processors.**
- "core" = central processing unit (CPU)



- Introducing parallelism!!!

Parallel Machines Today

Examples from Apple's product line:

Mac Pro
8 Intel Xeon E5 cores

MacBook Pro Retina 15"
6 Intel Core i9 cores

iPhone XR

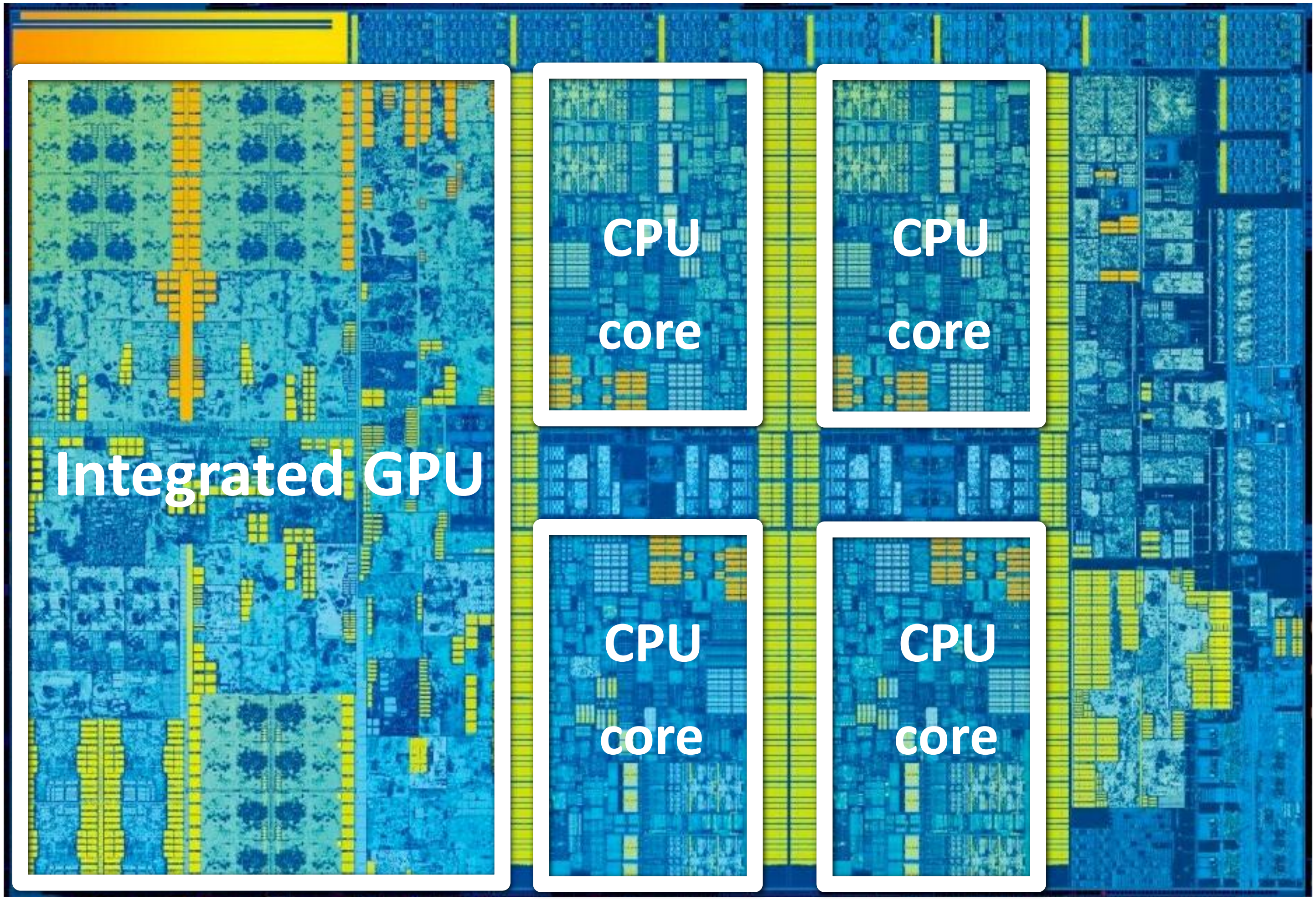
4 CPU cores
6 GPU cores

iMac Pro
18 Intel Xeon W cores

(images from apple.com)

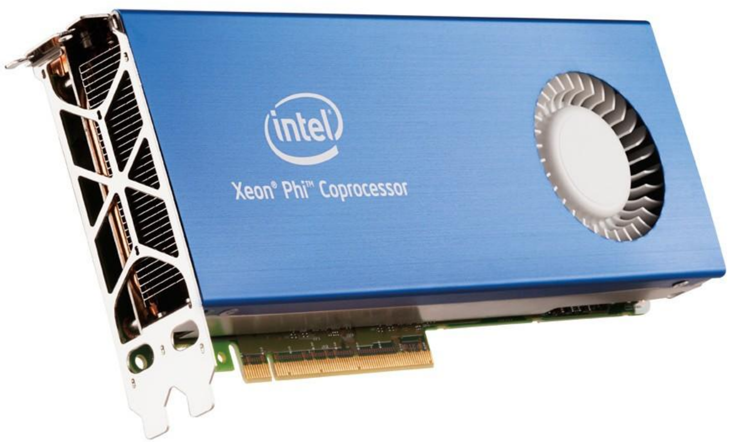
Intel Skylake (2015) (aka "6th generation Core i7")

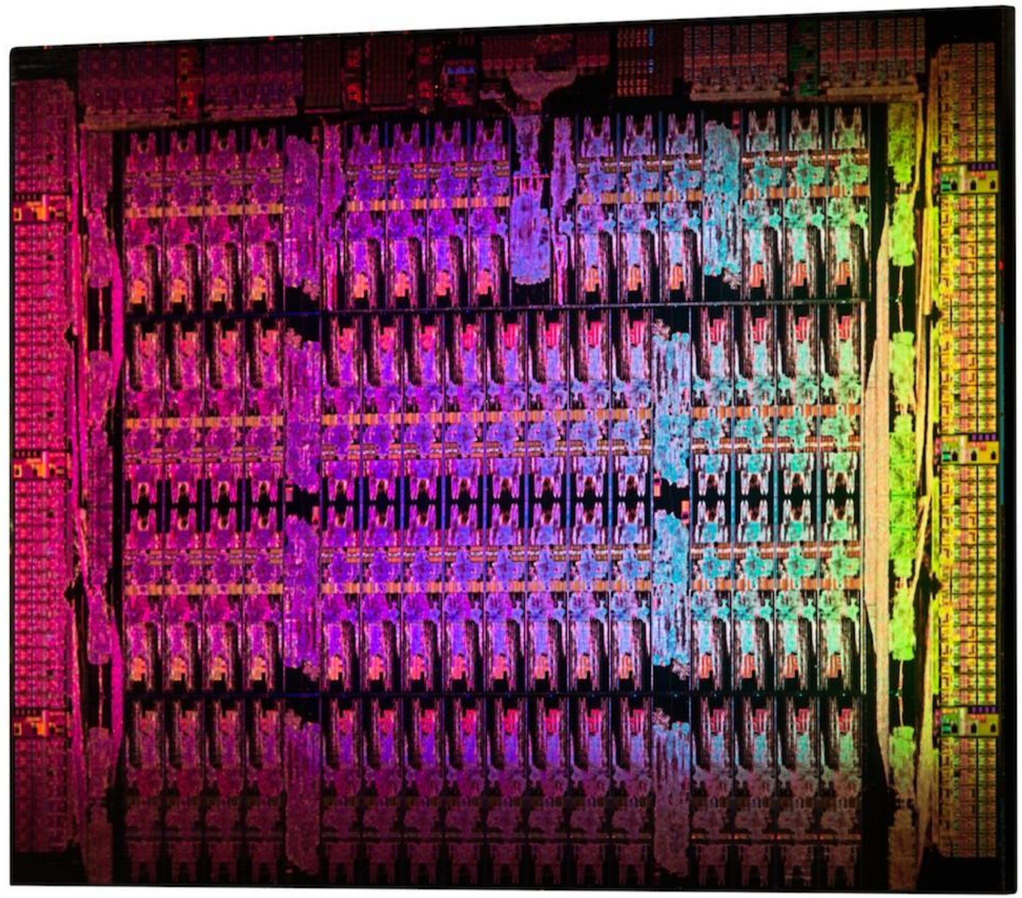
Quad-core CPU + multi-core GPU integrated on one chip



Intel Xeon Phi 7120A “coprocessor”

- 61 “simple” x86 cores (1.3 Ghz, derived from Pentium)
- Targeted as an accelerator for supercomputing applications

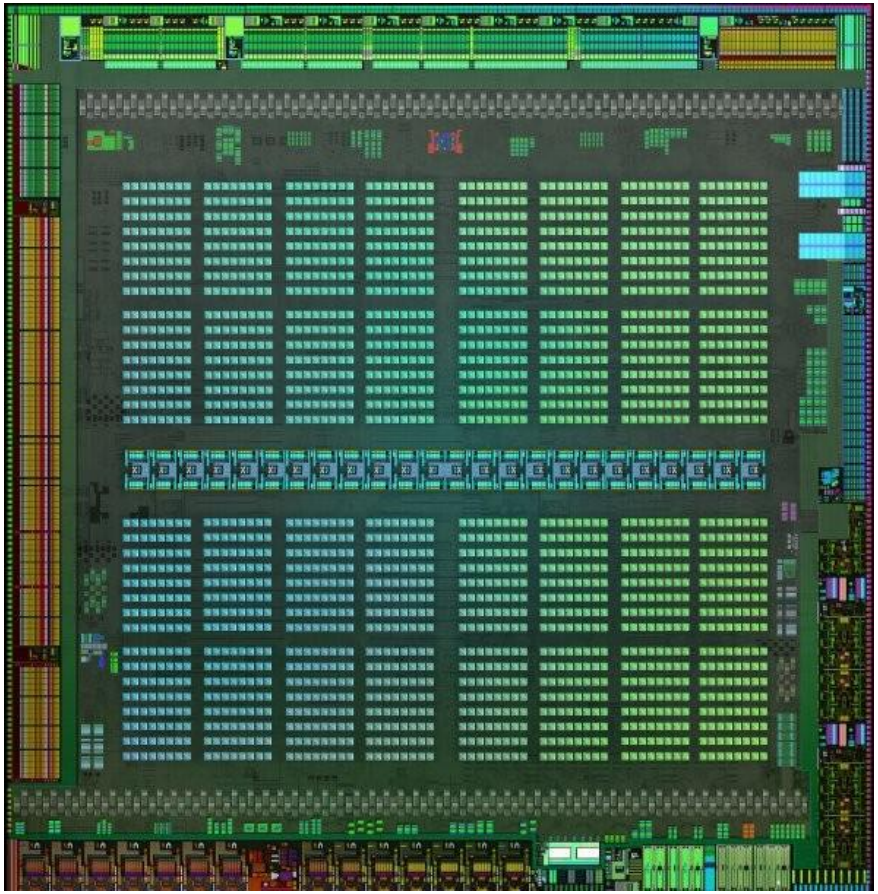




NVIDIA GV100 Volta GPU (2017)

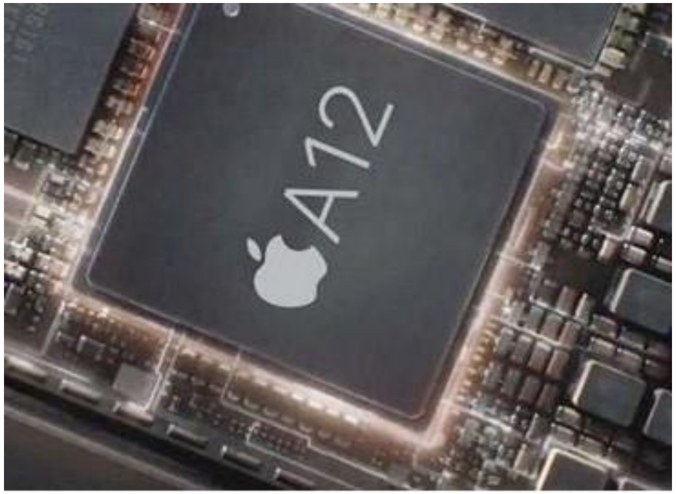
80 major processing blocks

(but much, much more parallelism available... details coming soon)



Mobile parallel processing

Power constraints heavily influence design of mobile systems



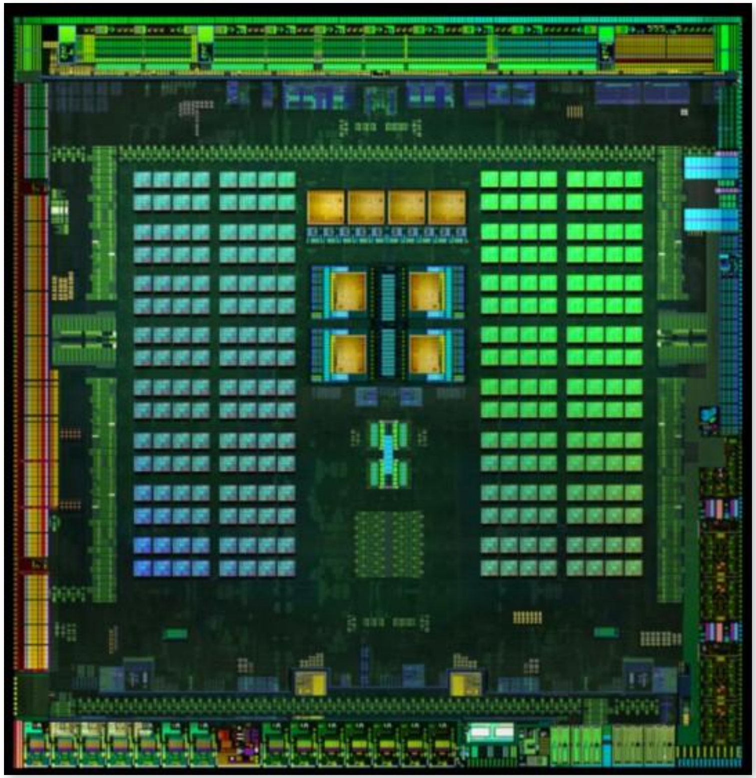
Apple A12: (in iPhone XR)

4 CPU cores

4 GPU cores

Neural net engine

+ much more



NVIDIA Tegra K1:

Quad-core ARM A57 CPU + 4 ARM A53 CPUs +

NVIDIA GPU + image processor...

Supercomputing

- Today: clusters of multi-core CPUs + GPUs
- Oak Ridge National Lab: Summit
  - 4,608 nodes
  - Each with two 22-core CPUs + 6 GPUs

Programmer's Perspective on Performance

Question: **How do you make your program run faster?**

Answer before 2004:

- **Just wait 6 months, and buy a new machine!**

Answer after 2004:

- **You need to write parallel software.**

Now it's up to the programmers

- Adding more processors **doesn't help** much if programmers aren't aware of them…
- … or don't know how to use them.

- Serial programs don't benefit from this approach (in most cases).

Why we need to write parallel programs

- Running multiple instances of a serial program often isn’t very useful.
- Think of running multiple instances of your favorite game.
- What you really want is for it to **run faster**.

Approaches to the serial problem

- Rewrite serial programs so that they're parallel.

- Write translation programs that automatically convert serial programs into parallel programs.
  - This is **very difficult** to do.
  - Success has been limited.

More problems

- Some coding constructs can be recognized by an automatic program generator, and converted to a parallel construct.

- However, it’s likely that the result will be a very inefficient program.

- Sometimes the best parallel solution is to step back and devise an entirely new algorithm.

Example

- Compute n values and add them together.

- Serial solution:

```
sum = 0;
for (i = 0; i < n; i++) {
    x = Compute_next_value(. . .);
    sum += x;
}
```

Example (cont.)

- We have p cores, p much smaller than n.
- Each core performs a partial sum of approximately n/p values.

```
my_sum = 0;
my_first_i = . . . ;
my_last_i = . . . ;
for (my_i = my_first_i; my_i < my_last_i; my_i++) {
   my_x = Compute_next_value( . . .);
   my_sum += my_x;
}
```

Each core uses it's own private variables and executes this block of code independently of the other cores.

Example (cont.)

- After each core completes execution of the code, a private variable my_sum contains the sum of the values computed by its calls to Compute_next_value.

- Ex., 8 cores, n = 24, then the calls to Compute_next_value return:

1,4,3, 9,2,8, 5,1,1, 5,2,7, 2,5,0, 4,1,8, 6,5,1, 2,3,9

Example (cont.)

- Once all the cores are done computing their private my_sum, they form a global sum by sending results to a designated "master" core which adds the final result.

Example (cont.)

```
if (I'm the master core) {
   sum = my_x;
   for each core other than myself {
      receive value from core;
      sum += value;
   }
} else {
   send my_x to the master;
}
```

Example (cont.)

Core	0	1	2	3	4	5	6	7
my_sum	8	19	7	15	7	13	12	14

<u>Global sum</u>

8 + 19 + 7 + 15 + 7 + 13 + 12 + 14 = 95

Core	0	1	2	3	4	5	6	7
my_sum	95	19	7	15	7	13	12	14

But wait!

There's a much better way to compute the global sum.

Better parallel algorithm

- Don't make the master core do all the work.
- Share it among the other cores.

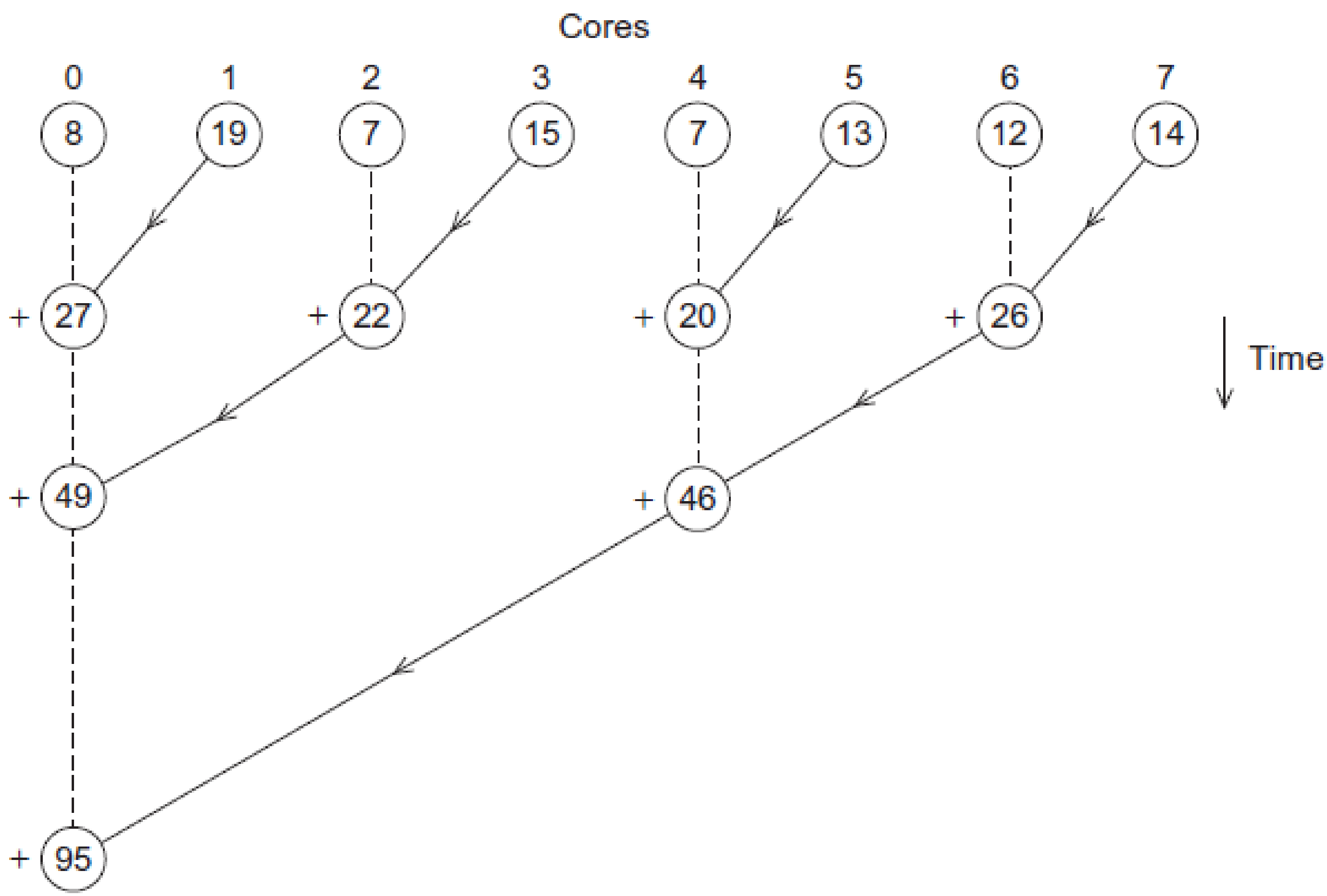
- Pair the cores so that core 0 adds its result with core 1's result.
- Core 2 adds its result with core 3's result, etc.
- Work with odd and even numbered pairs of cores.

Better parallel algorithm (cont.)

- Repeat the process now with only the evenly ranked cores.
- Core 0 adds result from core 2.
- Core 4 adds the result from core 6, etc.

- Now cores divisible by 4 repeat the process, and so forth, until core 0 has the final result.

Multiple cores forming a global sum



Analysis

- In the first example, the master core performs 7 receives and 7 additions.

- In the second example, the master core performs 3 receives and 3 additions.

- The improvement is more than a factor of 2!

Analysis (cont.)

- The difference is more dramatic with a larger number of cores.

- If we have 1000 cores:
  - The first example would require the master to perform 999 receives and 999 additions.
  - The second example would only require 10 receives and 10 additions.

- That's an improvement of almost a factor of 100!

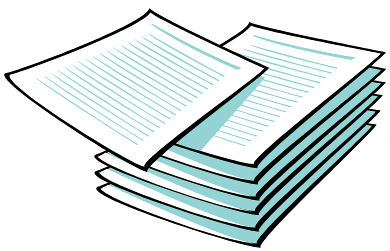
How do we write parallel programs?

- **Task parallelism**
  - Partition various tasks carried out solving the problem among the cores.
- **Data parallelism**
  - Partition the data used in solving the problem among the cores.
  - Each core carries out similar operations on it’s part of the data.

Professor P

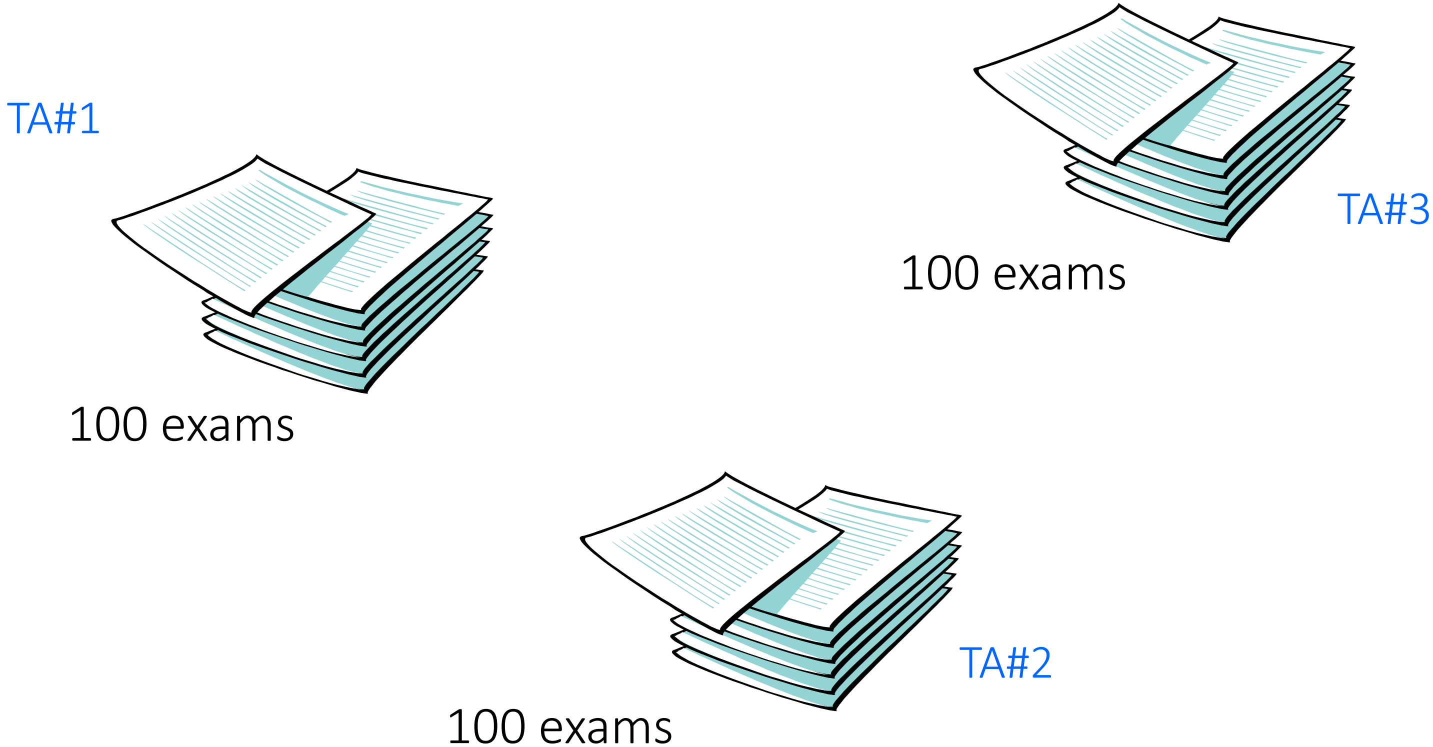
15 questions

300 exams



Professor P's grading assistants

Division of work – data parallelism



Division of work – task parallelism

TA#1

Questions 1 - 5

TA#3

Questions 11 - 15

TA#2

Questions 6 - 10

Division of work – data parallelism

```
sum = 0;
for (i = 0; i < n; i++) {
    x = Compute_next_value(. . .);
    sum += x;
}
```

Division of work – task parallelism

```
if (I'm the master core) {
    sum = my_x;
    for each core other than myself {
        receive value from core;
        sum += value;
    }
} else {
    send my_x to the master;
}
```

Tasks

1) Receiving
2) Addition

Coordination

- Cores usually need to **coordinate** their work.

- Communication – one or more cores send their current partial sums to another core.
- Load balancing – share the work evenly among the cores so that one is not heavily loaded.
- Synchronization – because each core works at its own pace, make sure cores do not get too far ahead of the rest.

What we'll be doing

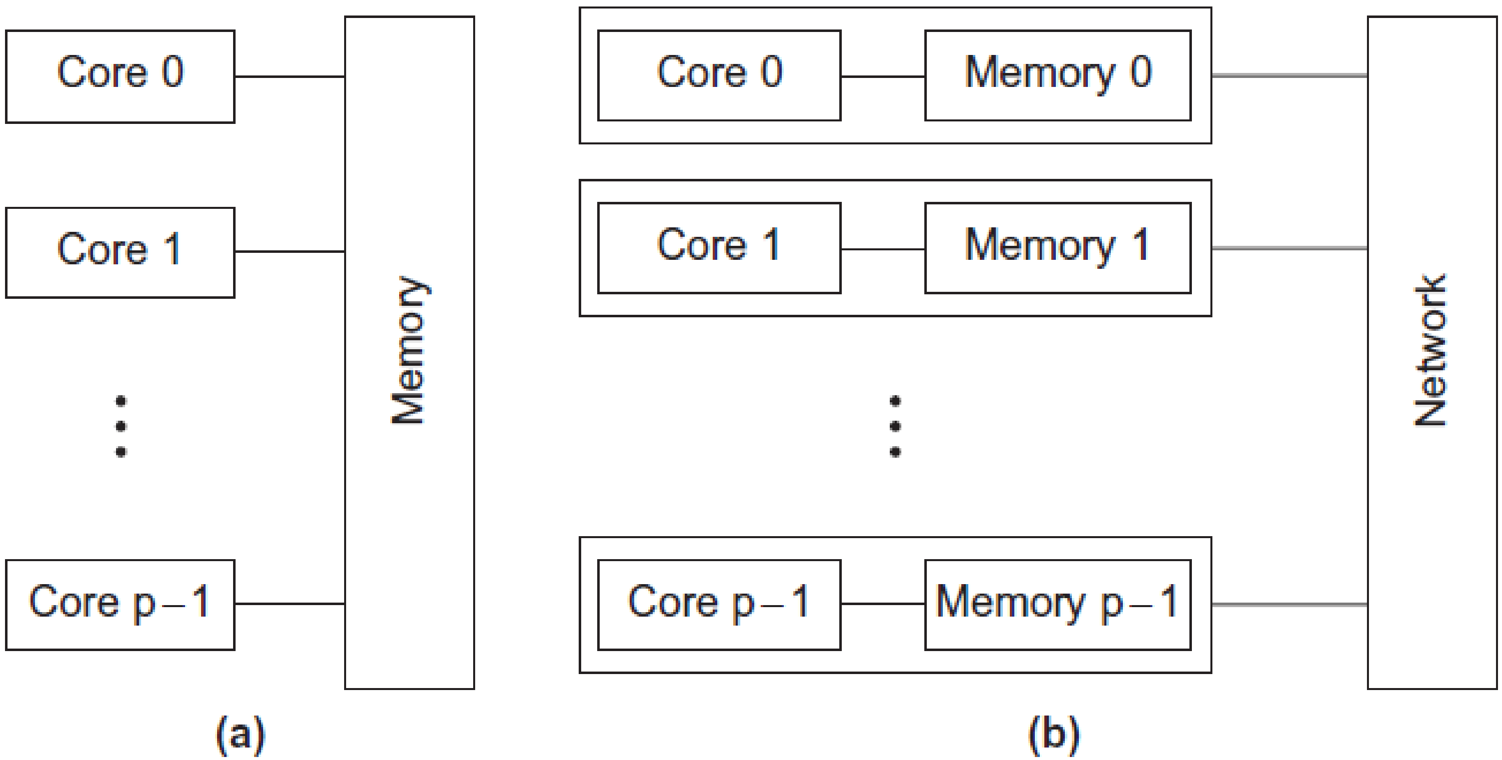
- Learning to write programs that are explicitly parallel.
- Using the C language.
- Using three different extensions to C.
    - Message-Passing Interface (MPI)
    - Posix Threads (Pthreads)
    - OpenMP

Type of parallel systems

- **Shared-memory**
  - The cores can share access to the computer's memory.
  - Coordinate the cores by having them examine and update shared memory locations.

- **Distributed-memory**
  - Each core has its own, private memory.
  - The cores must communicate explicitly by sending messages across a network.

Type of parallel systems



Shared-memory

Distributed-memory

Terminology

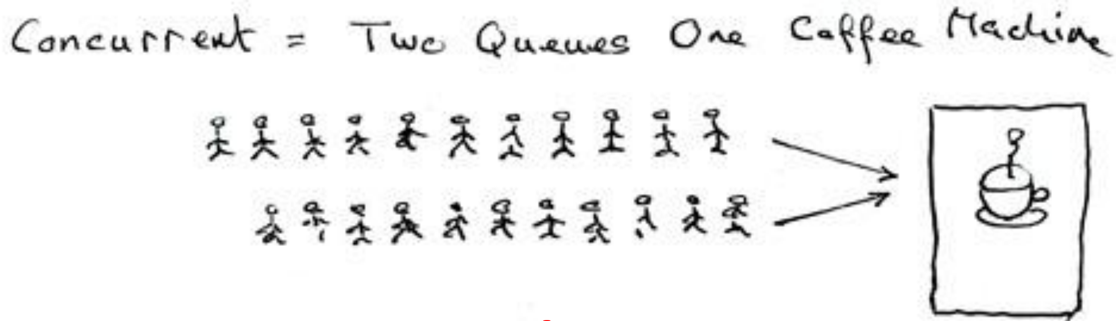
- Concurrent computing – a program is one in which multiple tasks can be in progress at any instant.
- Parallel computing – a program is one in which multiple tasks cooperate closely to solve a problem
- Distributed computing – a program may need to cooperate with other programs to solve a problem.

Concurrent vs. Parallel

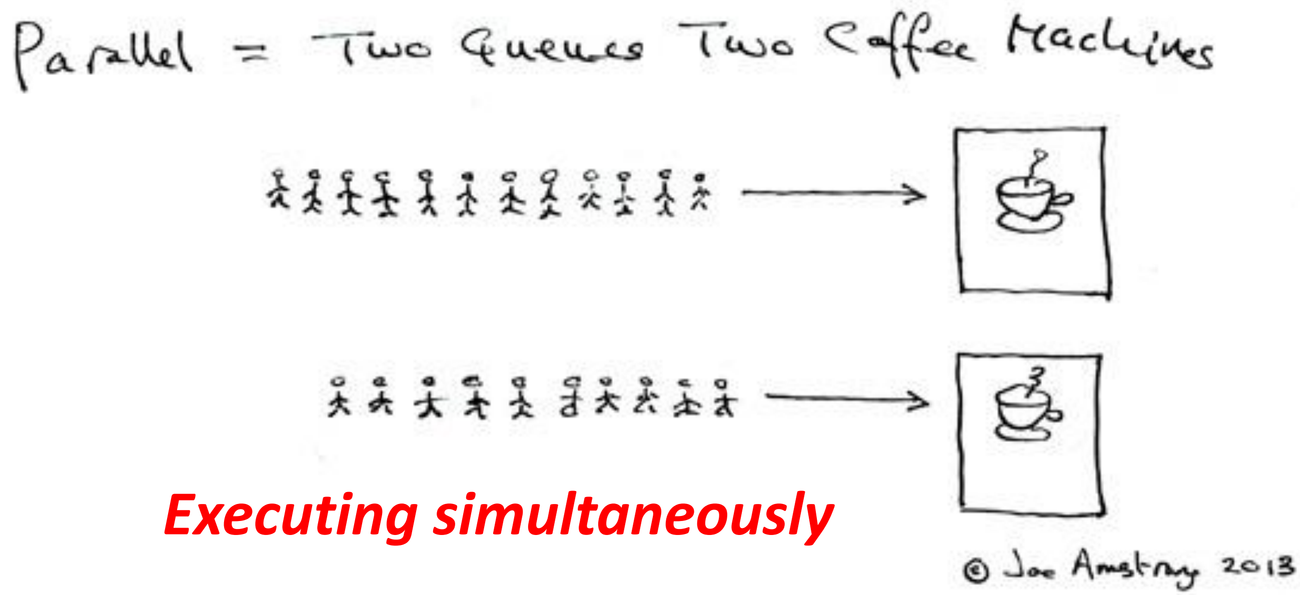
- A system is said to be **concurrent** if it can support two or more actions ***in progress at the same time***.

- A system is said to be **parallel** if it can support two or more actions ***executing simultaneously***.

- **Concurrent** = Two queues and one coffee machine.
- **Parallel** = Two queues and two coffee machines.



in progress at the same time.



Executing simultaneously

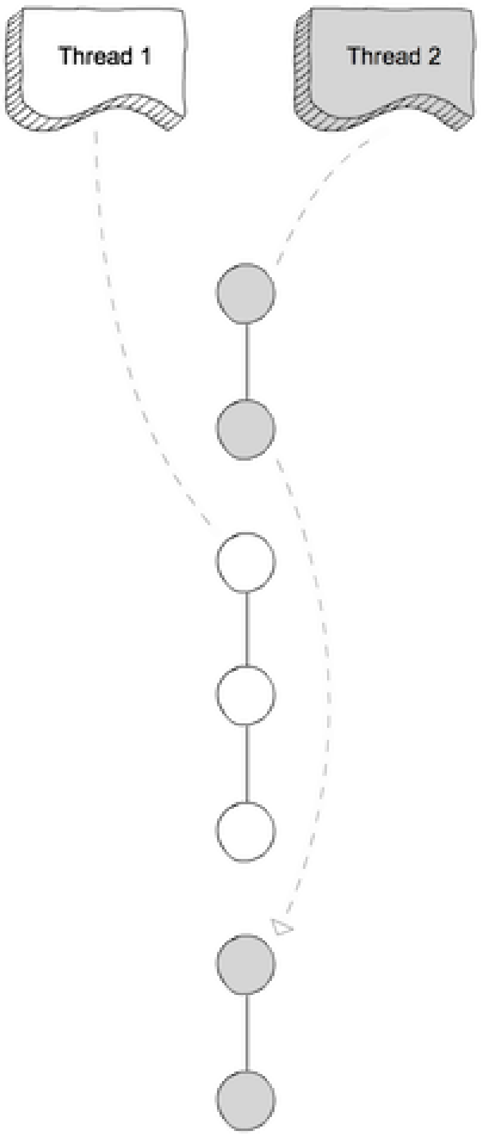
Parallel Computing vs. Concurrent Computing

- In parallel computing, execution occurs at the *same physical instant*
  - parallel computing is impossible on a (one-core) single processor

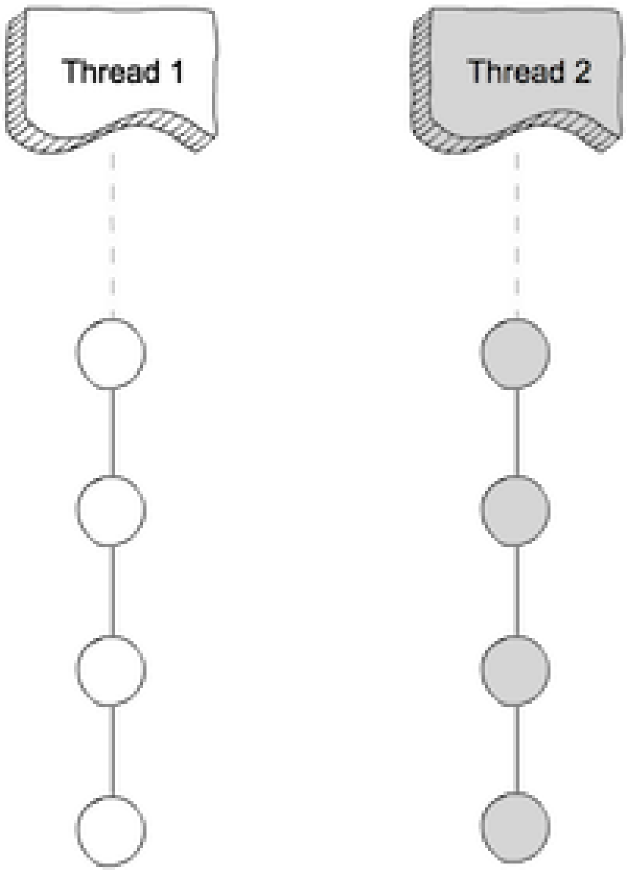
- concurrent computing consists of process lifetimes overlapping, but *execution need not happen at the same instant*
  - concurrent processes can be executed on one core by interleaving the execution steps of each process via time-sharing slices

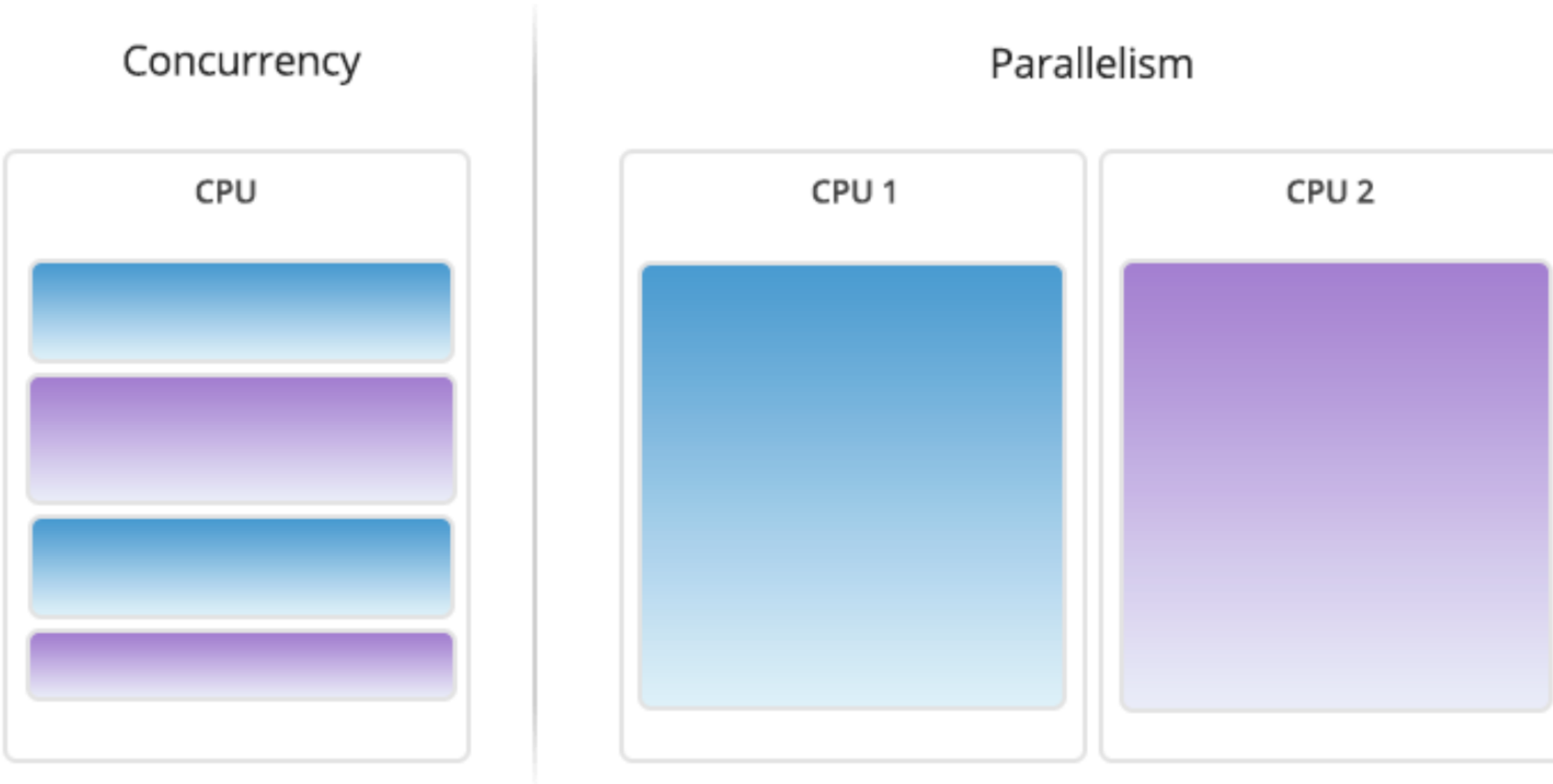
https://en.wikipedia.org/wiki/Parallel_computing

Concurrent Computing



Parallel Computing




Concurrency
Parallelism
CPU
CPU 1
CPU 2

豆包版

- **Q：并发计算、并行计算、分布式计算有什么区别？**

- **并发计算**：是指在**同一时间段内，系统能处理多个任务，但任务不一定同时执行**。在单处理器系统中，通过**快速切换上下文，使多个任务看似同时进行**，实则轮流使用处理器资源。如一个服务器同时处理多个用户请求，在不同时间片分别处理。
- **并行计算**：指在**同一时刻，多个任务在不同的处理器或处理核心上同时执行，强调真正意义上的同时计算**，需多个硬件资源支持，像超级计算机的大量处理器同时处理不同数据。
- **分布式计算**：是把一个大的计算任务分解成多个子任务，**分配到不同的计算节点（可以是不同的计算机、服务器等）上进行处理**，这些计算节点通过网络相互连接和通信，共同完成整个计算任务。

DeepSeek版本

1. 并发计算（Concurrent Computing）

- **定义**：多个任务交替执行，通过时间片轮转或事件驱动实现。
- **特点**：
  - **任务交替**：任务在同一处理器上交替执行。
  - **资源共享**：任务共享处理器、内存等资源。
  - **响应性**：提高系统响应能力，适合I/O密集型任务。
- **应用场景**：操作系统进程调度、Web服务器处理请求。

DeepSeek版本

2. 并行计算（Parallel Computing）

- **定义**：多个任务同时执行，通常使用多处理器或多核系统。
- **特点**：
  - **任务同时执行**：任务在多个处理器或核心上并行运行。
  - **资源独立**：每个任务有独立的处理器和内存。
  - **高性能**：适合计算密集型任务，如科学计算。
- **应用场景**：大规模数值模拟、图像处理。

DeepSeek版本

3. 分布式计算（Distributed Computing）

- **定义**：任务在多个计算节点上执行，节点通过网络连接。
- **特点**：
  - **任务分布**：任务分布在多个节点上。
  - **资源分散**：节点有独立的处理器和内存，通过网络通信。
  - **可扩展性**：适合大规模计算任务，如大数据处理。
- **应用场景**：云计算、分布式数据库、区块链。

对比总结

特性	并发计算	并行计算	分布式计算
任务执行方式	交替执行	同时执行	分布执行
资源管理	共享资源	独立资源	分散资源
主要目标	提高响应性	提高性能	提高可扩展性和容错性
典型应用	操作系统、Web服务器	科学计算、图像处理	云计算、分布式数据库

Concluding Remarks (1)

- The laws of physics have brought us to the doorstep of multicore technology.
- Serial programs typically don't benefit from multiple cores.
- Automatic parallel program generation from serial program code isn't the most efficient approach to get high performance from multicore computers.

Concluding Remarks (2)

- Learning to write parallel programs involves learning how to coordinate the cores.

- Parallel programs are usually very complex and therefore, require sound program techniques and development.

- What you have learnt:

1+1=2

- **What you need to solve:**