CME 213, ME 339 Spring 2023
Introduction to parallel computing using MPI, openMP, and CUDA

Eric Darve, ICME, Stanford

"Computers are useless. They can only give you answers." (Pablo Picasso)

Stanford University

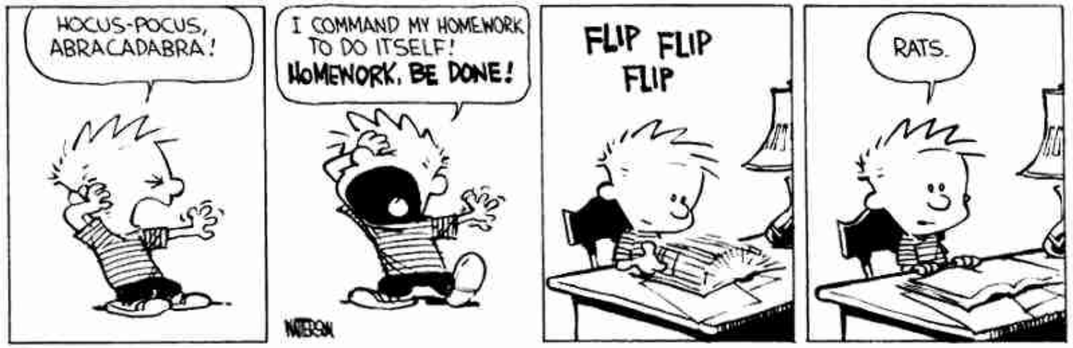
Homework 1

- Pre-requisite homework
- Topics:
  - derived classes
  - polymorphism
  - standard library
  - testing using Google Test

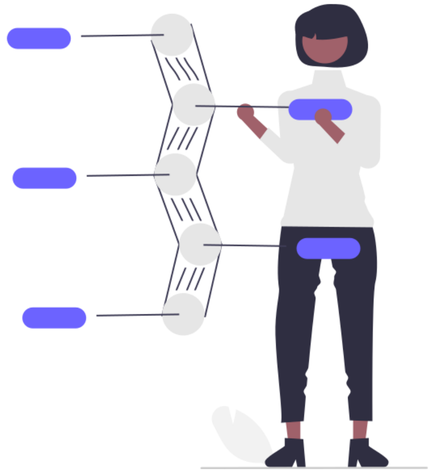
Submission

1. Submit your PDF on gradescope
2. For your computer code, copy your files to cardinal
3. Run a Python script it submit code

Grading is done on gradescope



Testing



- This is key to writing correct programs.
- Each line of code you write should be tested.
- Best practices for debugging:
  - Test short pieces of code
  - Progressively test larger groups of functions and longer pieces of code
  - Test final prediction and full code

Google Unit Test

- https://github.com/google/googletest
- https://google.github.io/googletest/
- https://google.github.io/googletest/primer.html

Test organization

- Test: a short piece of code to test a single functionality
- Test suite: contains one or many related tests; used to group and organize your tests
- Test program: a collection of test suites; this is the complete set of tests you use for a given homework assignment.

Assertions

Test are based on testing simple assertions such:

- Are two numbers equal?
- Is the difference between two numbers small enough?
- Is the sign of a number correct?
- Is a logical condition satisfied?
- Was the expected exception thrown (`std::runtime_error`)?
- Many other tests are available…

Test structure

```
#include "gtest/gtest.h"

// These are our tests
TEST(demoTest, Size_positive) {
  ASSERT_GT(n, 0) << "Integer n should be positive";
}
```

General syntax:

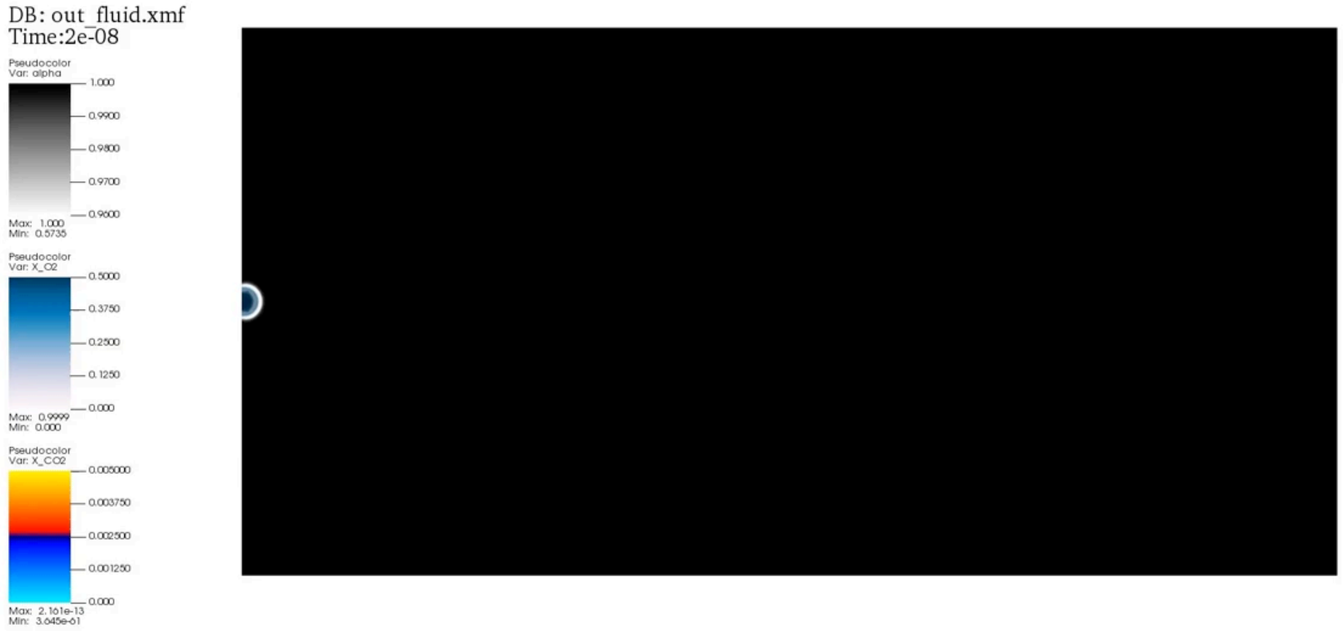
```
TEST(TestSuiteName, TestName) {
  ... test body ...
}
```

Demo code

- Using Google Colab and a notebook to compile and run your code
- How to run shell commands inside a notebook
- IPython built-in magic commands
- Installing GoogleTest on Google Colab
- Running and analyzing the results of GoogleTest
- Editing and syncing files with Google Drive.

Why parallel computing?

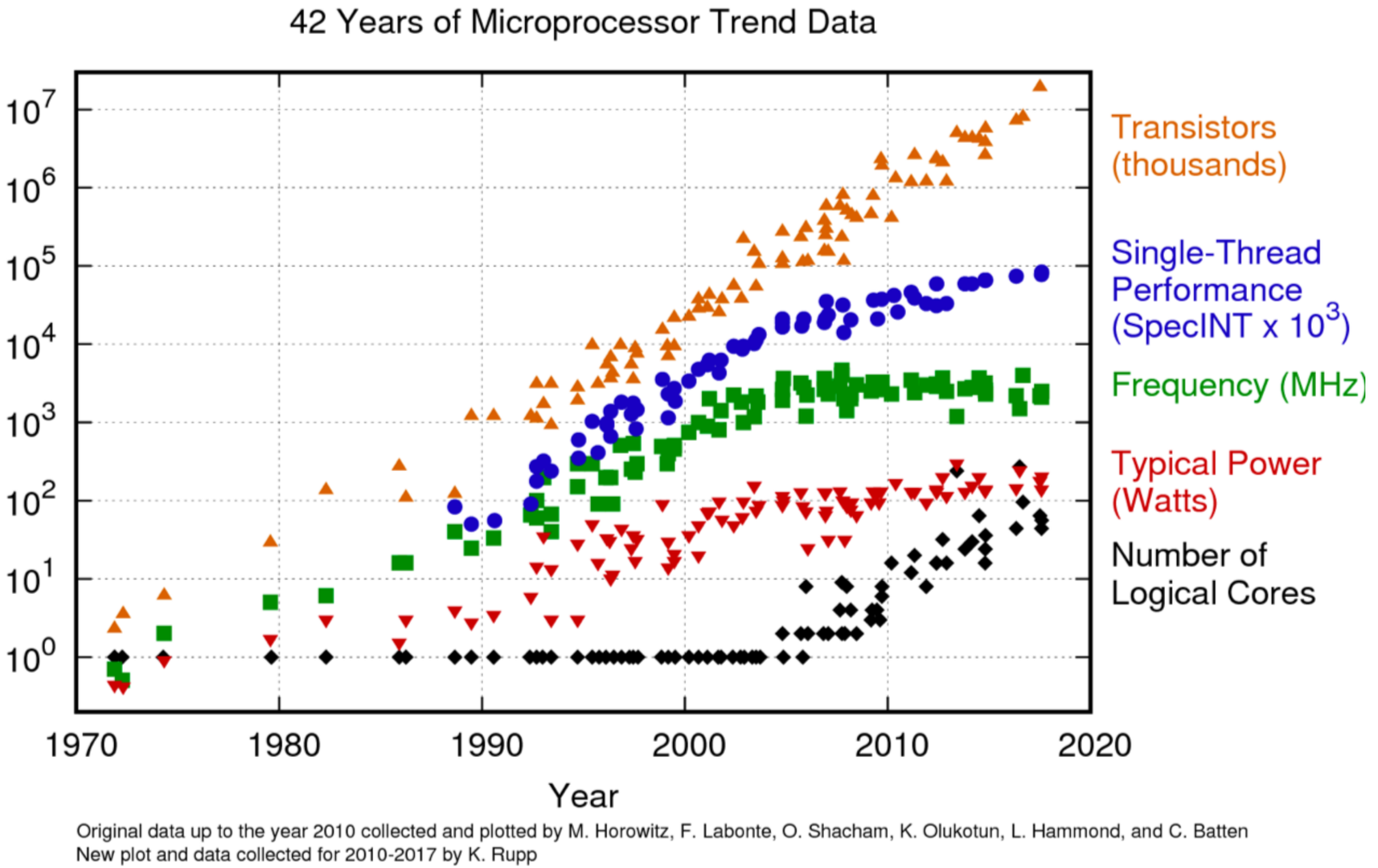
- Parallel computing is omni-present in computational engineering
- Any type of non-trivial computing requires parallel computing
- It happens either on laptop, desktop, or on supercomputers, cloud computing platforms, …



Trends in micro-processors

- Gordon Moore 1965: “the number of transistors on a chip shall double every 18–24 months.” This has remained true until very recently.
- Accompanied by an increase in clock speed
- Consequence of this historical trend: sequential processing cores get faster.
- So it used to be that going parallel was a luxury but not a requirement.

Intel microprocessor trends



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

- **What trends do you observe on this plot?**
- **What does this mean for computer programming?**

Physical limits

But, increase in transistor density is limited by:

- Leakage current increases
- Power consumption increases
- Heat generated increases

This puts a hard limit on how fast computer cores can go. We have hit the technological limits of sequential computing.

The solution

Memory access time has not been reduced at a rate comparable to the processing speed

Go parallel! Multiple cores and memory buses on a processor

Design decisions

Multicore processors

- 4 to 64 cores
- Few but powerful. Optimized for both sequential and parallel computing.
- Examples: Ryzen, Epyc, Xeon
- Characteristics: high single-thread performance (e.g., silicon devoted to out of order execution, deeper pipelines, more superscalar execution units, and larger, more general caches) and shared memory.

The Sea Giant of micro-processors

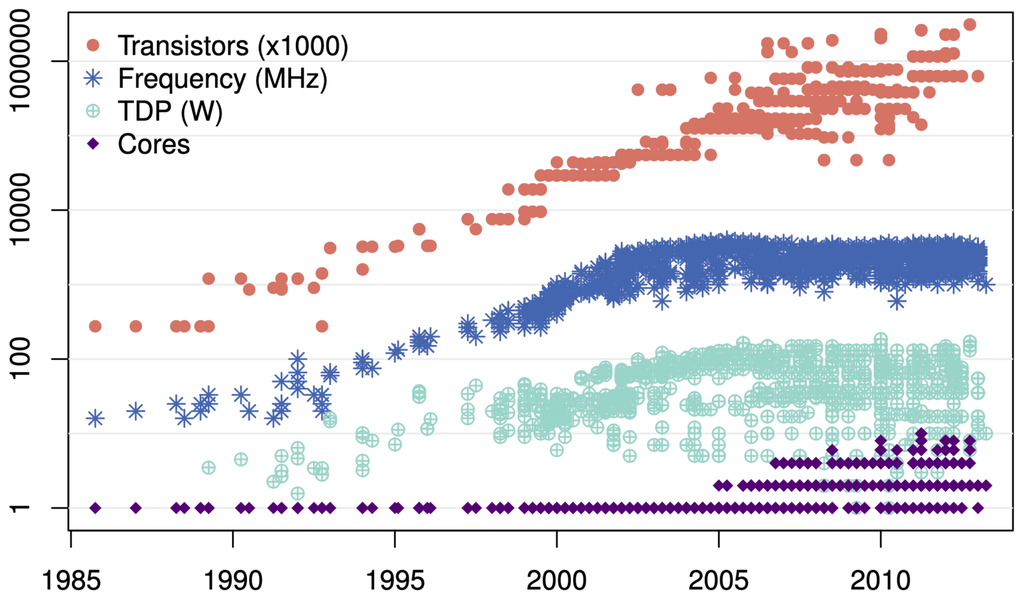
Many core processors

- A lot of cores but not as powerful.
- Often called **accelerators.**
- Designed for higher degree of **explicit parallelism** and for **higher throughput** (or lower power consumption) at the expense of latency and lower single-thread performance.
- Not suitable for sequential computations.
- Examples: NVIDIA and AMD GPUs, TPU (Tensor Processing Units).

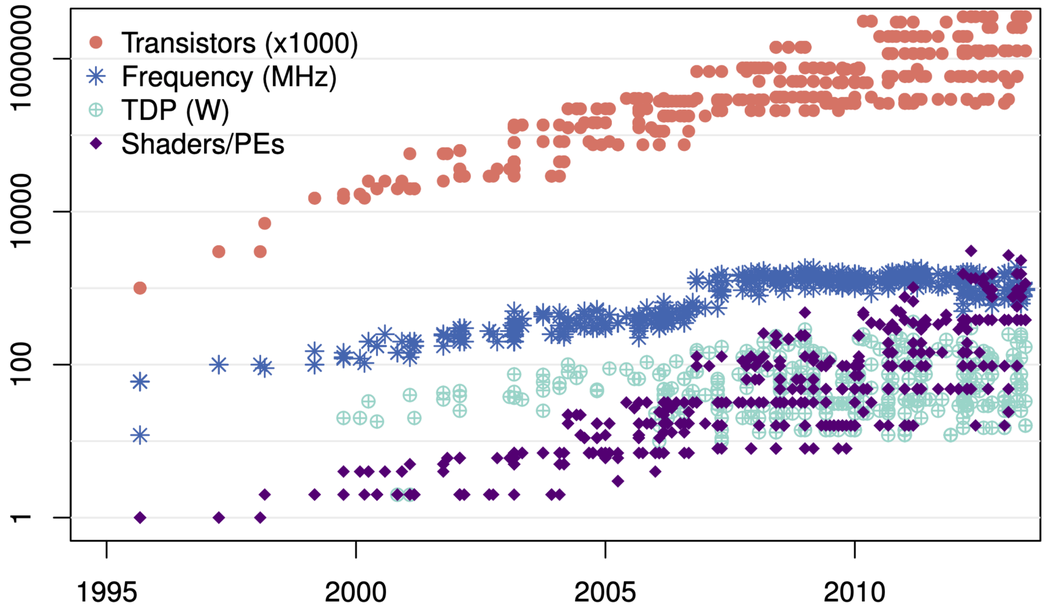
Weak but numerous

Historical data

Number of cores increases but frequency plateaus



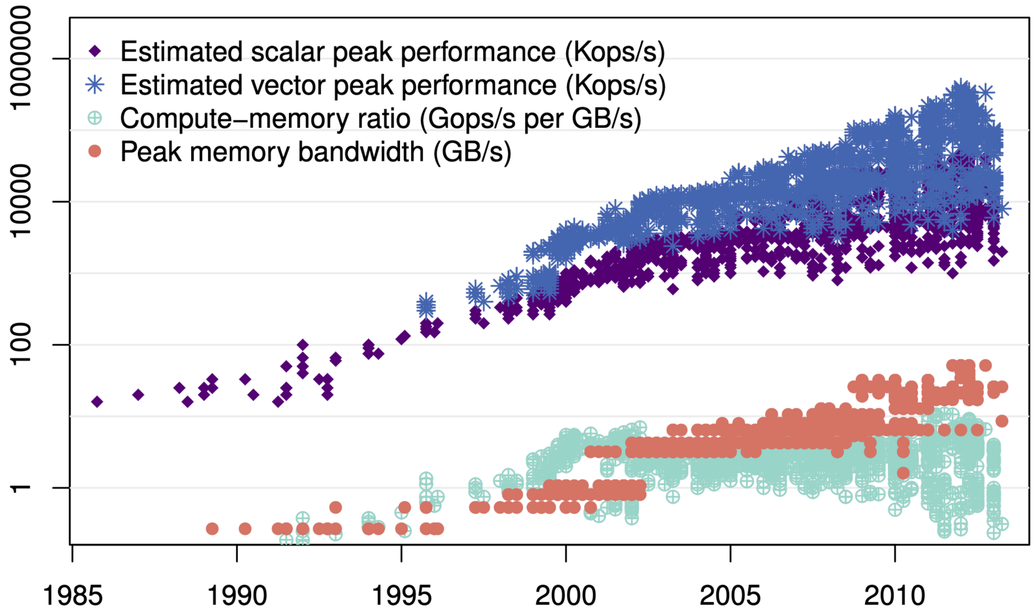
Historical data on 1403 Intel microprocessors



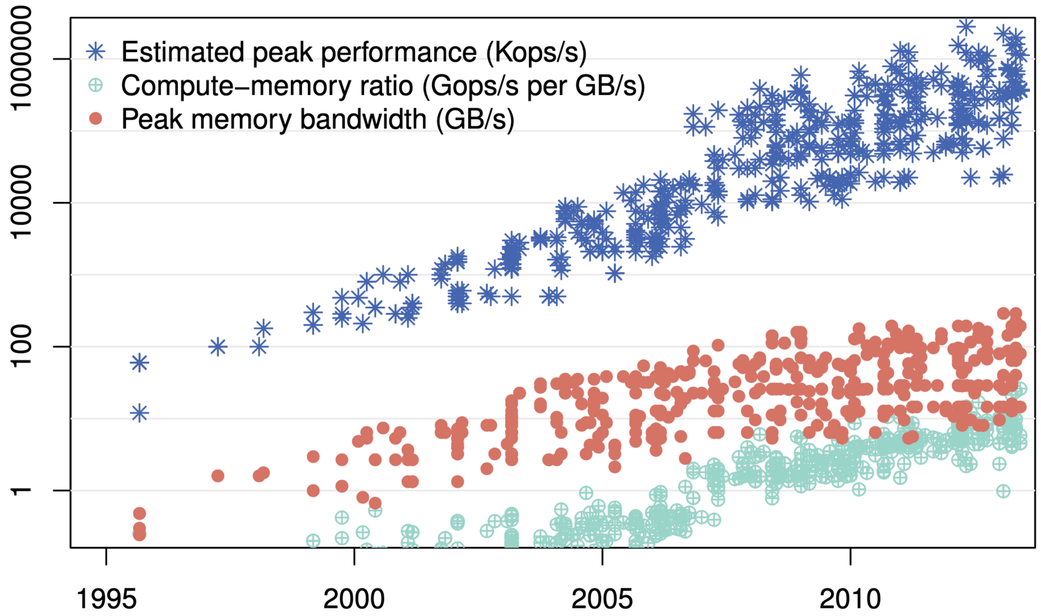
Historical data on 566 NVIDIA GPUs

What trends do you observe for the number of cores?
For the frequency?
For the number of transistors?
TDP = thermal design power = maximum amount of heat generated by a computer chip

Memory wall; bandwidth and latency



Intel microprocessors



NVIDIA GPUs

Which processors have the best performance?
Which processors have the best sequential performance?
Which processors are easiest to optimize?

More info at

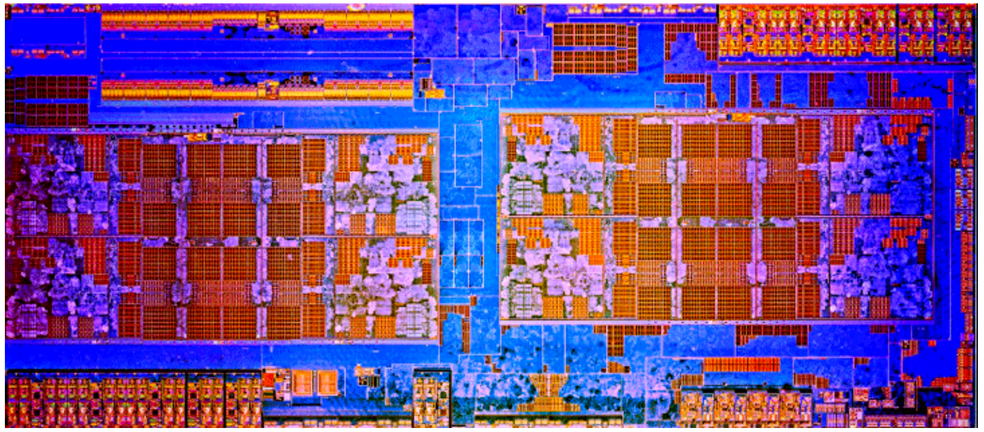
https://pure.tue.nl/ws/portalfiles/portal/3942529/771987.pdf

Parallel computing everywhere!

Summit supercomputer

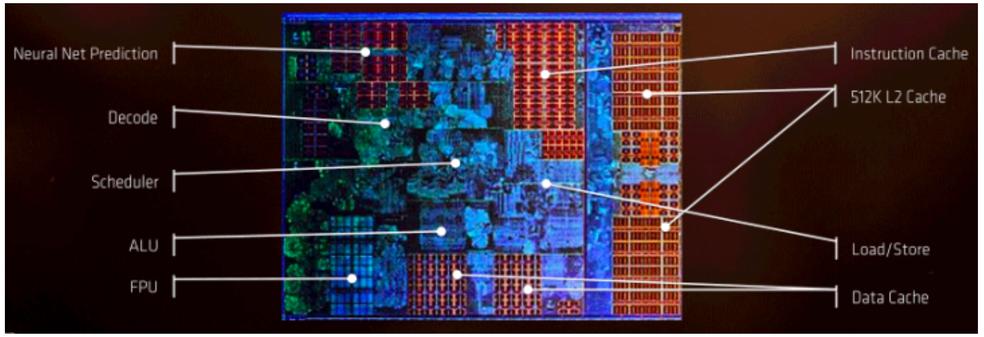
Parallel computing exists at many scales, price tags, and programming complexity

Examples of processor architectures
Multi and many core processors



Ryzen; 8-core processor

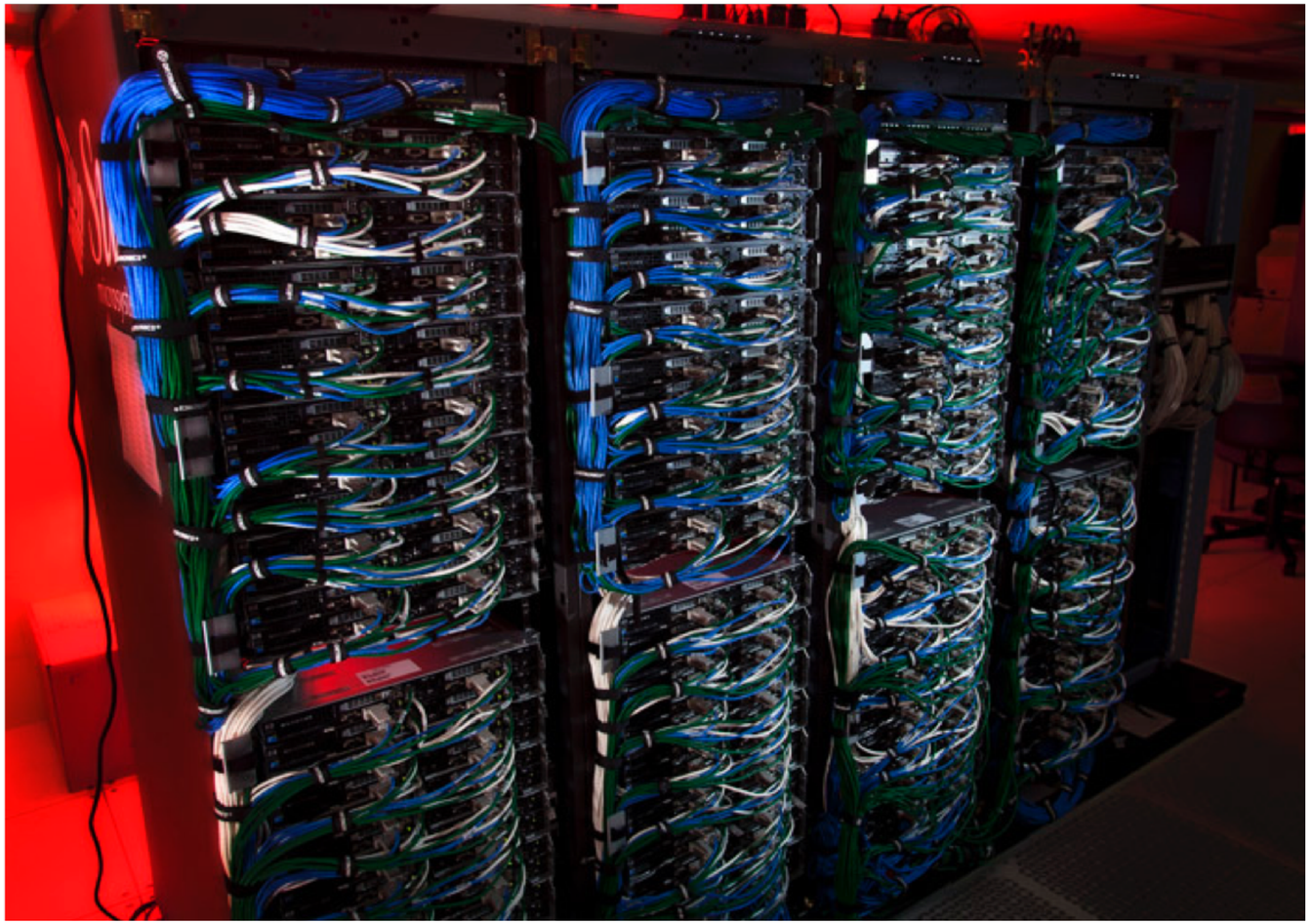
NVIDIA Turing TU102 architecture



One of the 8 cores

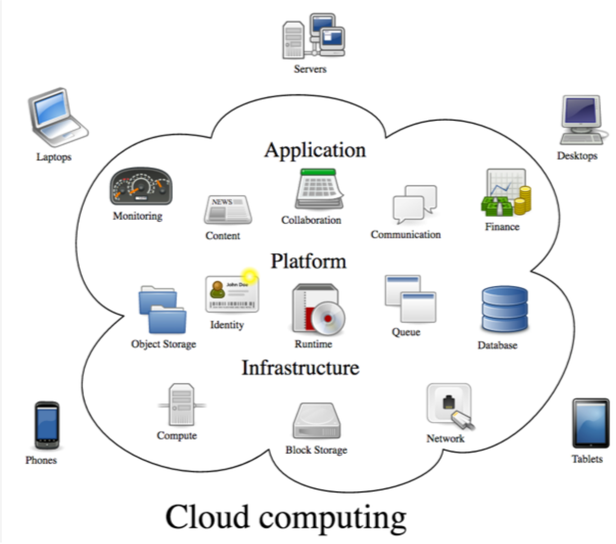
How would you describe the differences between these architectures?
How many cores do you see?
How much memory and cache do you see?

Computer cluster



As you can see, the interconnect is a critical part of the cluster.
So is cooling.
The design favors a compact assembly.

Is cloud computing the future of high-performance computing (HPC)?



- The many advantages of cloud computing: resources are immediately available, computer nodes can be configured with any software, “infinite” scalability, reliability.

- But performance can be lower when many cores connected by a fast network are required.

Frontier—Oak Ridge National Laboratory 1.6 Exaflop Peak Performance

JAGUAR	TITAN	SUMMIT	FRONTIER
No. 1 in 2009, 2010	No. 1 in 2012	No. 1 in 2018	No. 1 in 2022

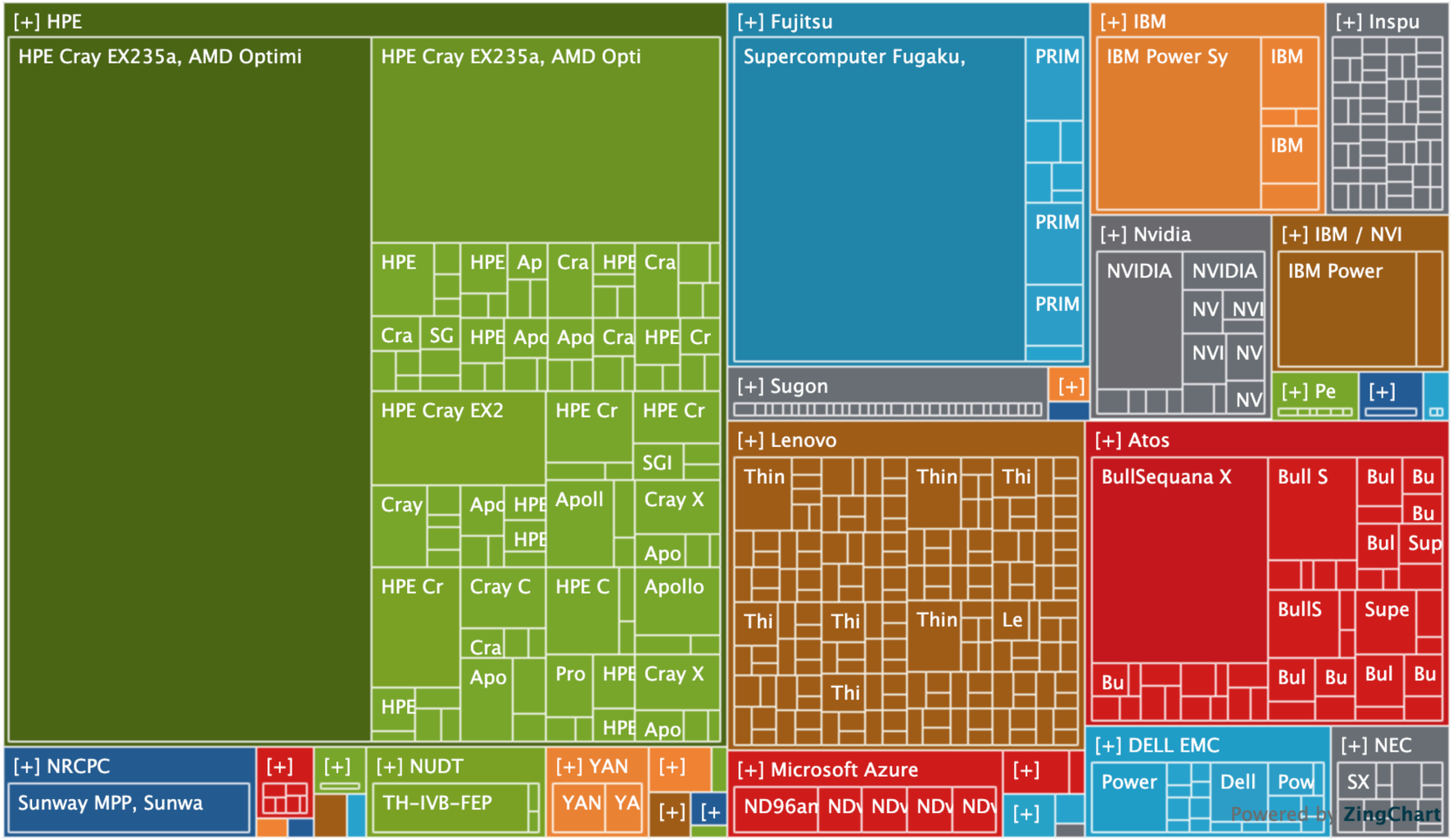
SYSTEM SPECS	TITAN	SUMMIT	FRONTIER
Peak Performance	27 PF	200 PF	1.6 EF
Cabinets	200	256	74
Node	1 AMD Opteron CPU 1 NVIDIA K20X Kepler GPU	2 IBM POWER9™ CPUs 6 NVIDIA Volta GPUs	1 HPC and AI Optimized 3rd Gen AMD EPYC CPU 4 Purpose Built AMD Instinct 250X GPUs
CPU-GPU Interconnect	PCI Gen2	NVLINK Coherent memory across the node	AMD Infinity Fabric
System Interconnect	Gemini	2x Mellanox EDR 100G InfiniBand Non-Blocking Fat-Tree	Multiple Slingshot NICs providing 100 GB/s network bandwidth. Slingshot network which provides adaptive routing, congestion management and quality of service.
Storage	32 PB, 1 TB/s, Lustre Filesystem	250 PB, 2.5 TB/s, GPFS™	2-4x performance and capacity of Summit's I/O subsystem. Frontier will have near node storage like Summit.

Top 500 Supercomputers

Top 500 supercomputers
https://www.top500.org/

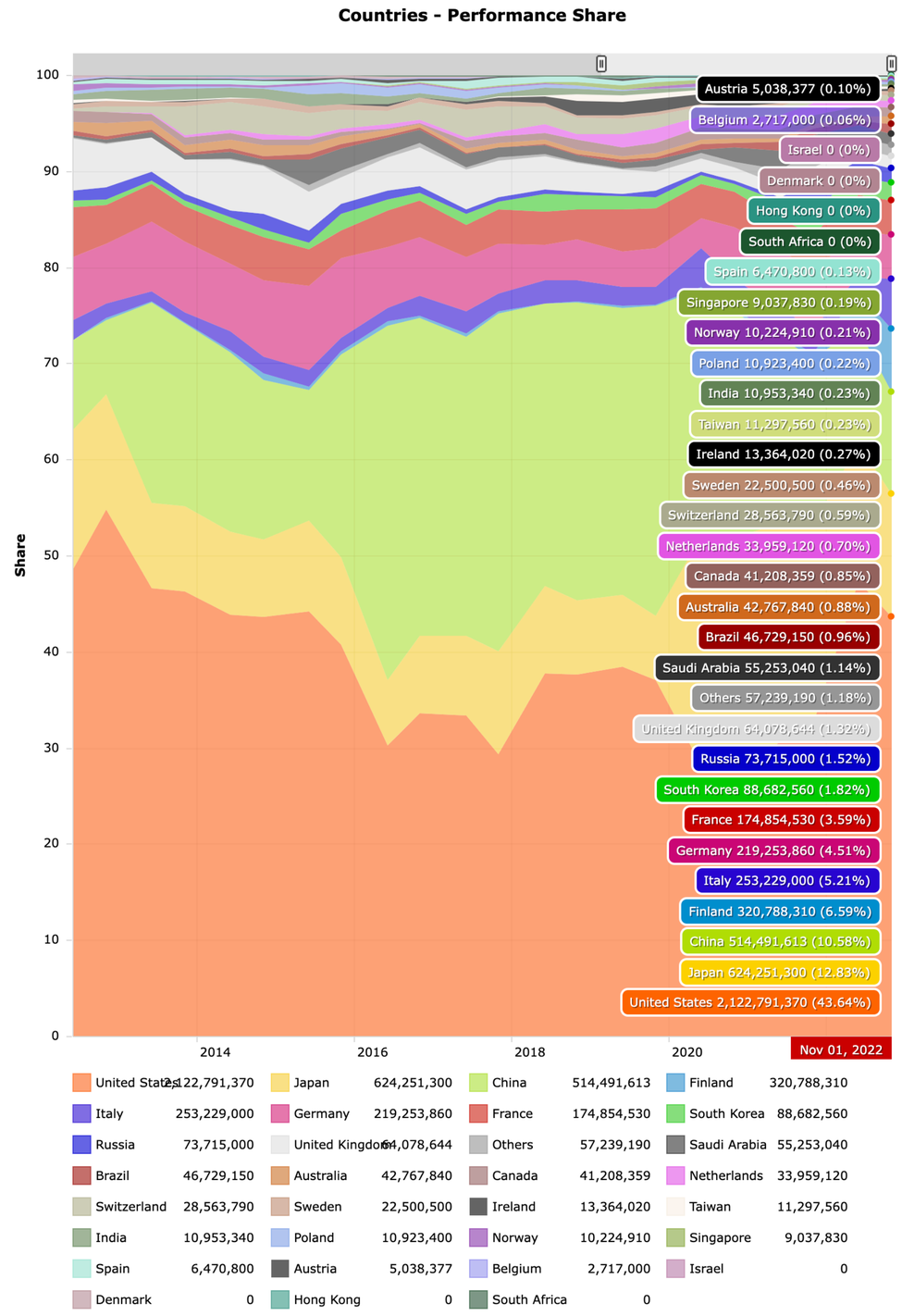
- Frontier is the clear winner of the race to exascale, and it will require a lot of work and innovation to knock it from the top spot.
- The Fugaku system at the Riken Center for Computational Science (R-CCS) in Kobe, Japan, previously held the top spot for two years in a row before being moved down by the Frontier machine. With an HPL score of 0.442 EFlop/s, Fugaku has retained its No. 2 spot from the previous list.
- The LUMI system, which found its way to the No. 3 spot on the last list, has retained its spot. However, the system went through a major upgrade to keep it competitive. The upgrade doubled the size of the machine, which allowed it to achieve an HPL score of 0.309 EFlop/s.
- HPL: High-Performance Linpack Benchmark for Distributed-Memory Computers

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,730,112	1,102.00	1,685.65	21,100
2	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
3	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,220,288	309.10	428.70	6,016
4	**Leonardo** - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband, Atos EuroHPC/CINECA Italy	1,463,616	174.70	255.75	5,610
5	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148.60	200.79	10,096
6	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94.64	125.71	7,438



- Vendor share
- What are the dominant vendors in this market?

- What main trends do you observe?
- What are the top 5 dominant players in HPC?
- More at https://www.top500.org/

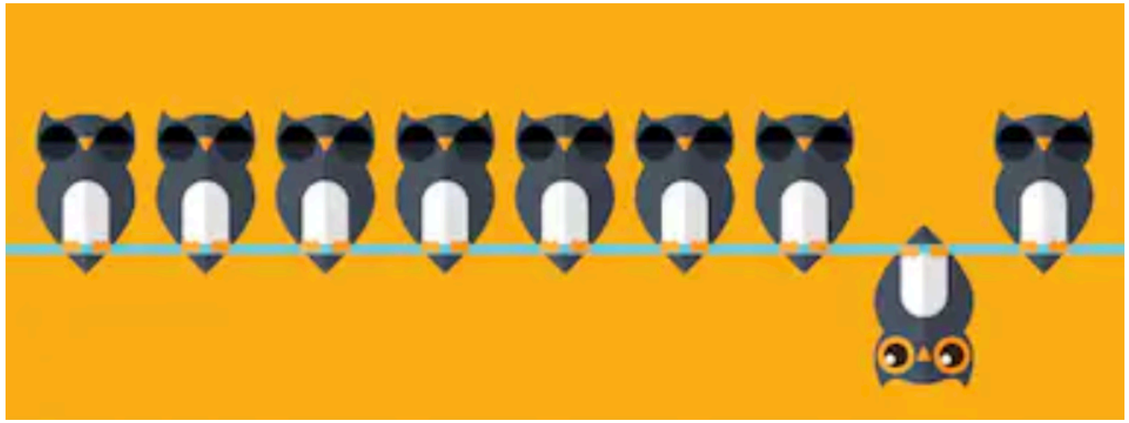


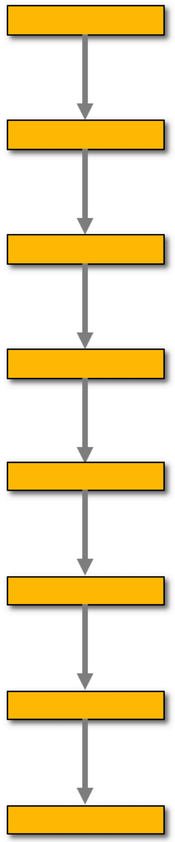
ity

Example of a Parallel Computation

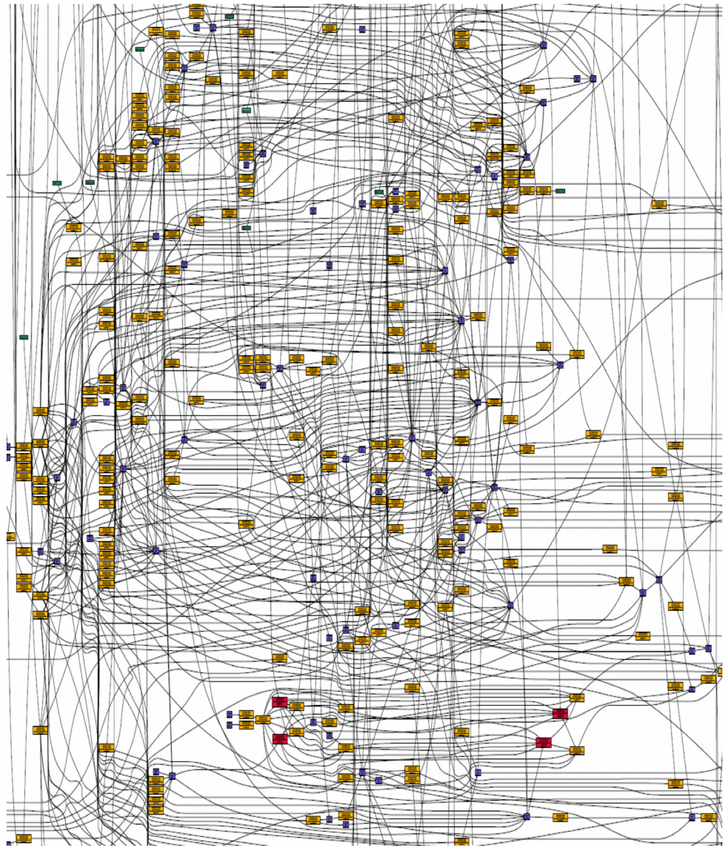
Parallel programs often look very different from sequential programs

- Parallel programming requires a different type of thinking.
- It’s more complicated and difficult to debug that sequential programs.





Sequential



Parallel

- Sequential programming is easy to understand.
- Parallel programming requires understanding dependencies between computational tasks and how data flow from one task to another.

Example: program to sum numbers

- Assume we want to calculate the sum of n numbers.
- This seems simple enough with a sequential code.

```
for (int i = 0; i < n; ++i) {
  float x = ComputeNextValue();
  sum += x;
}
```

Parallel computing

- We have p cores that can compute and exchange data.
- Can we accelerate our calculation by splitting the work among the cores?

Split the work

- First, we split the work across the p processing units (PU).
- Each PU computes a chunk of the sum.
- The chunk size is b.

```
int r; /* thread number */
int b; /* number of entries processed */
int my_first_i = r * b;
int my_last_i = (r + 1) * b;
for (int my_i = my_first_i; my_i < my_last_i; my_i++) {
  float my_x = ComputeNextValue();
  my_sum += my_x;
}
```

Final sum

- Not that simple
- At this point, each core has computed a partial sum.
- All these partial sums need to summed up together.
- We have p numbers that need to be added using p PUs or cores.

Sequential approach

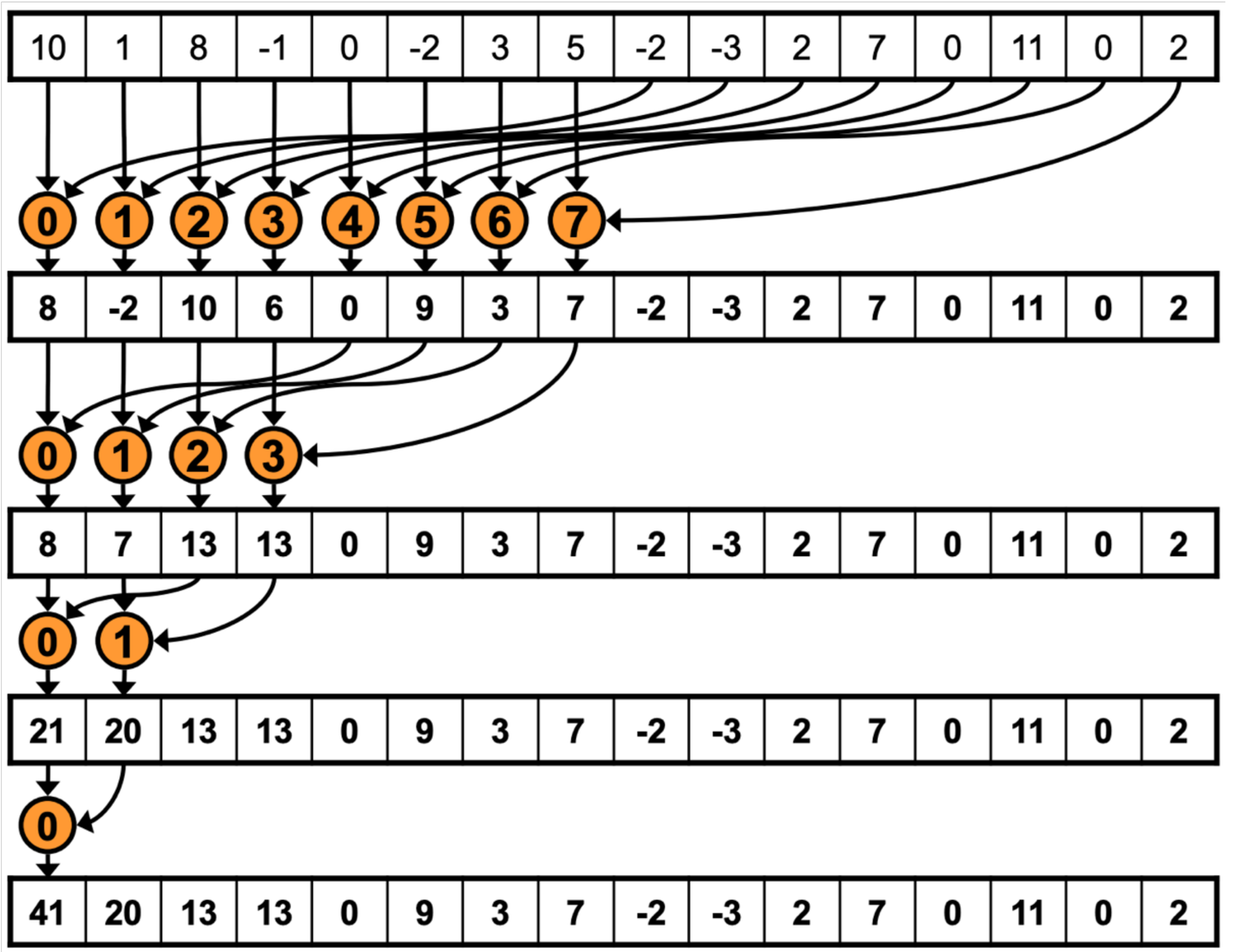
- The simplest.
- Have one main thread do all the work.
- Thread = PU or core doing work.
- ReceiveFrom: receive data from remote thread.
- SendTo: send data to main thread (with ID 0).

```
if (r == 0) /* main thread */ {
  float sum = my_sum;

  for (int r_mote = 1; r_mote < p; ++r_mote) {
    float sum_r;
    ReceiveFrom(&sum_r, r_mote);
    sum += sum_r;
  }

} else /* worker thread */ {
  SendTo(&my_sum, 0);
}
```

Sequential bottleneck

- But that may not be enough.
- **If we have many cores, this final sum may take a lot of time.**
- This is true for example on GPUs where the number of cores is very large.
- Example: NVIDIA H100 SXM5 has 16,896 FP32 cores.
- In many applications, it may become comparable with the size of the loop.

- Binary tree can be used to perform the reduction efficiently.
- $p = 8$ in this example.
- Number of passes or stages is $\ln_2(p) + 1$.
- This algorithm is used to compute reductions on GPUs.



Parallel algorithms

- This simple example illustrates the fact that it is difficult for a compiler to parallelize a program.

- Instead the programmer must often re-write his code having in mind that multiple cores will be computing in parallel.

- Key questions:

1. How can the workload be distributed across the different cores? That is: what is each core going to compute?

2. How can we estimate the runtime of an execution in parallel?

3. What are the performance bottlenecks?

Shared Memory Processors

Memory organization

- Despite the availability of many cores, performance is often driven by memory accesses.
- **Assigning 2x the number of cores to a calculation does not typically result in a 2x speedup!**
- Memory access times need to be accounted for and in most cases are the bottleneck.

Problems with memory access time

- **Memory access time** can not be reduced at the same rate as the processor clock period. This leads to an increased number of machine cycles for a memory access.

- **The speed of signal transfers** within the wires is a limiting factor. For example, a 3 GHz processor has a cycle time of 0.33 ns. Assuming a signal transfer at the speed of light, a signal can cross a distance of ~10 cm in one processor cycle. This is not significantly larger than the typical size of a processor chip. **Wire lengths become an important issue.**

- The physical size of a processor chip limits the **number of pins** that can be used, thus limiting the bandwidth between CPU and main memory.

- This may lead to a processor-to-memory performance gap which is sometimes referred to as **memory wall.**

- Many hardware improvements have been made to mitigate these shortcomings.

- They mostly involve using multiple cores and a distributed hierarchical memory.

Overview of key components of the architecture

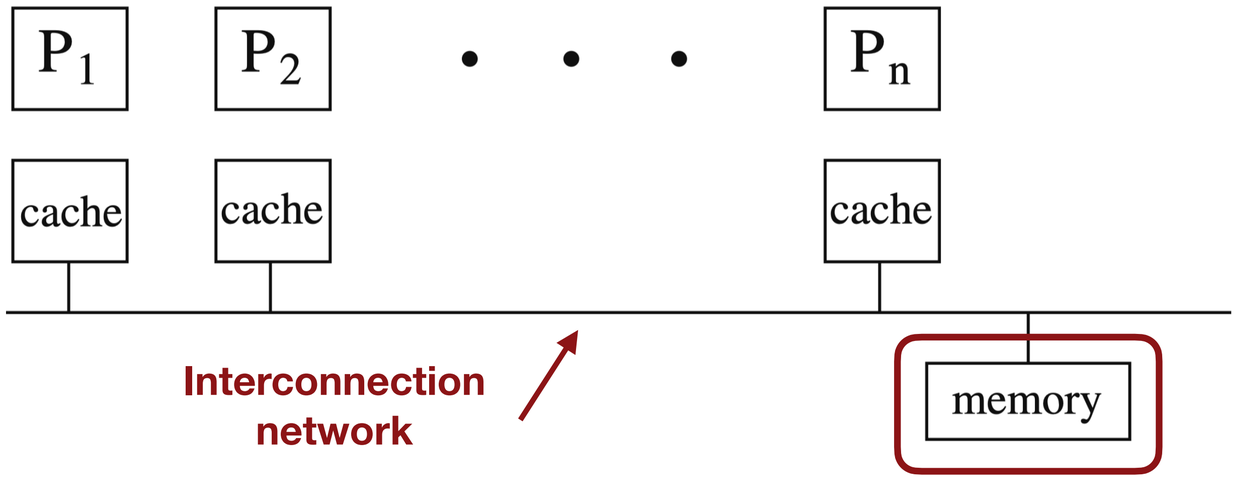
Components of a multicore computer node:

1. A number of processors or cores (single or multiple CPU sockets)
2. A shared physical memory (single or distributed global memory)
3. Cache memory
4. An interconnection network to connect the processors with the memory

Typically designed to deliver performance for both sequential and parallel programs.

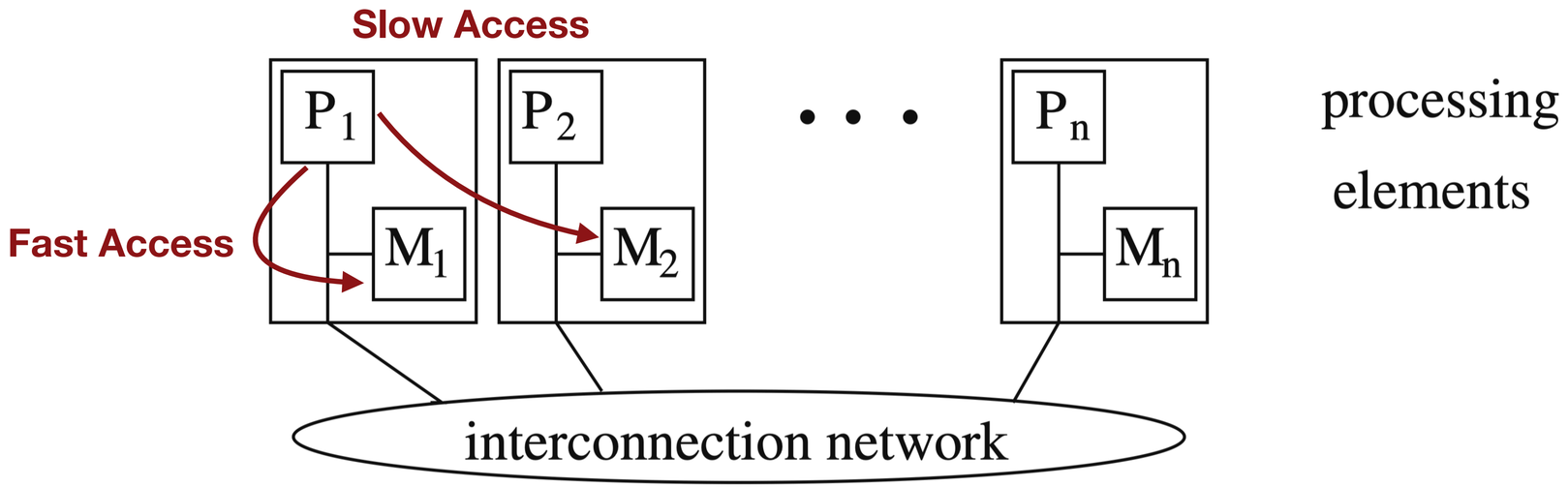
Symmetric multiprocessors

- The simplest but not the one with highest performance.
- A single global memory is shared among all processors.
- All processors have the **same access time** to memory = symmetric.



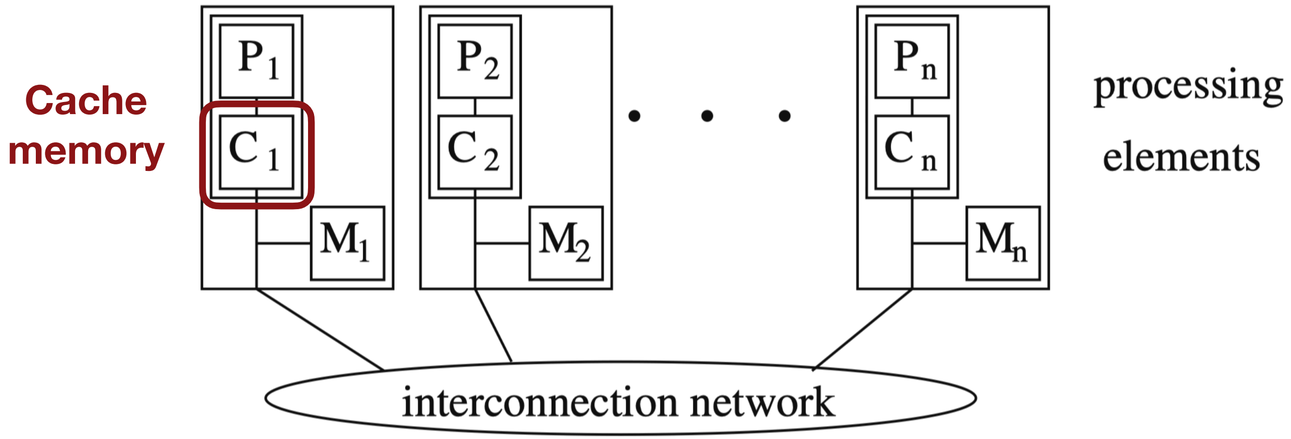
NUMA

- **Non-uniform memory access** = NUMA
- It is difficult to ensure high performance with a single shared memory.
- Instead, in most systems, the memory is **distributed,** and processors are connected to the different memories using a network.



CC-NUMA

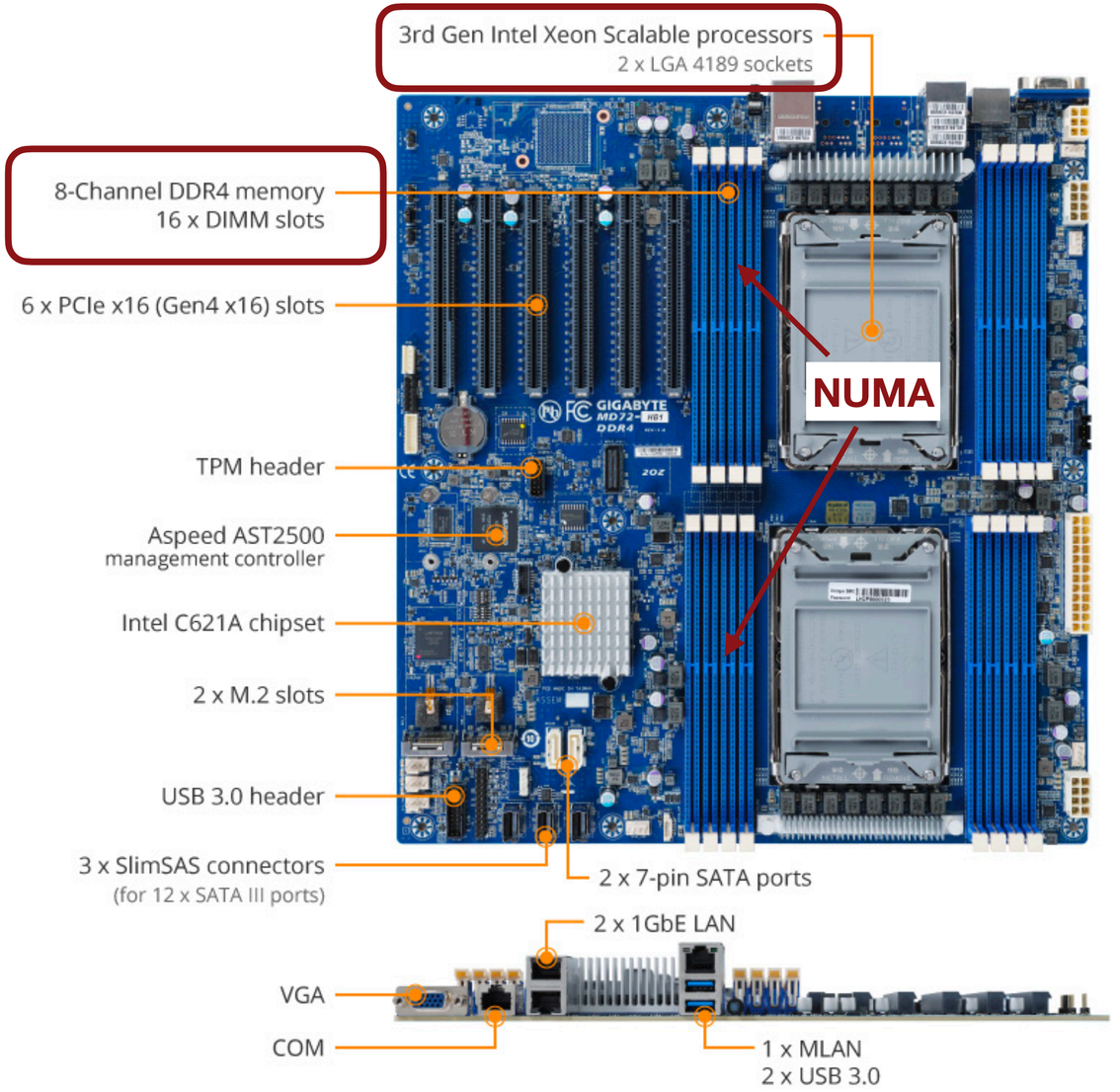
- Cache-coherent NUMA = CC-NUMA
- Most memory systems make use of **cache memory** to reduce the memory traffic.
- Cache is a **small** memory that is **much faster** than global memory.
- When a piece of data is read from global memory, it is stored in the cache. Subsequent reads use the value in the cache rather than global memory. As a result, this reduces the memory traffic significantly and speeds-up the program.



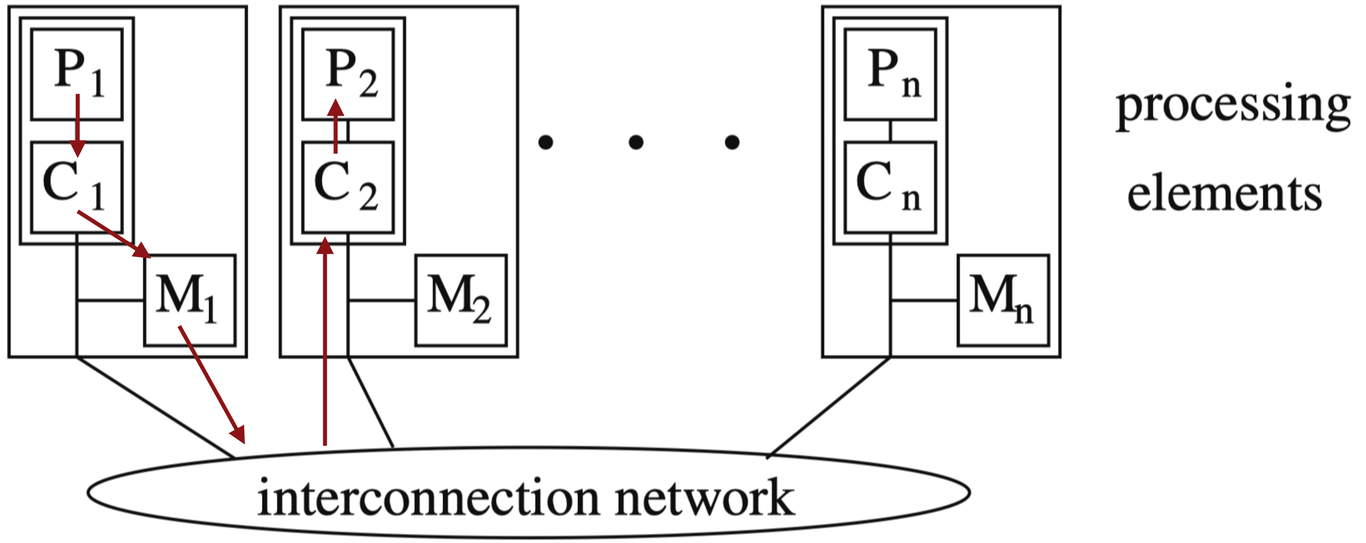
Cache-coherent NUMA or CC-NUMA

Example of CC-NUMA: Bulldozer server (AMD)

Gigabyte motherboard
3rd Gen Intel Xeon—“Ice Lake”

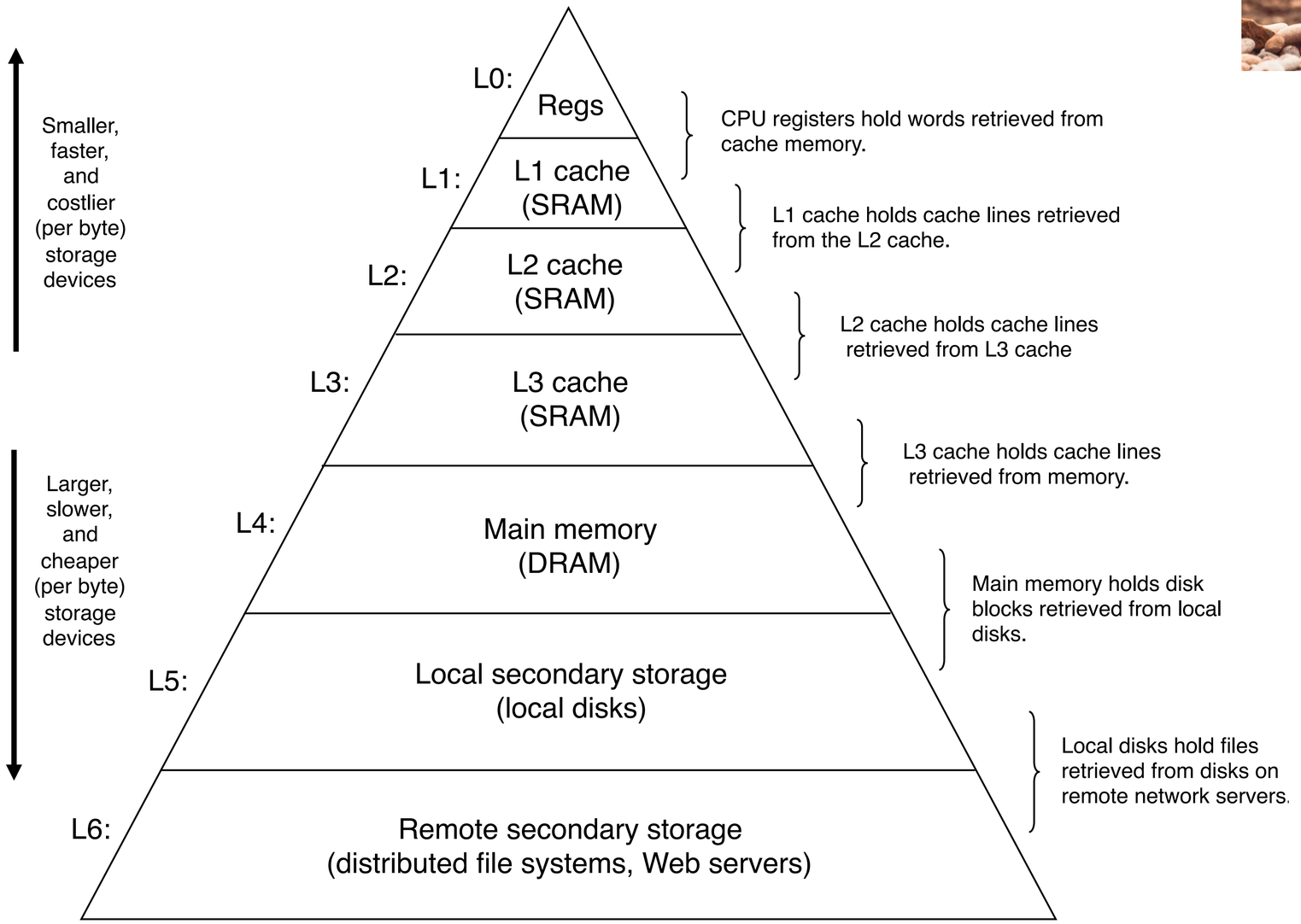


Cache coherence



- **Cache coherence** is a complex problem.
- Basic difficulty: processor P_1 writes data to memory x. Processor P_2 reads from memory x at a later instruction in the program.
- How can we make sure that P_2 gets the most recent value?
- Data x needs to be copied from cache to global memory, then from global memory to the cache of P_2.
- To ensure cache coherency, a **cache coherency protocol** must be used.
- This topic will not be further covered in this class.

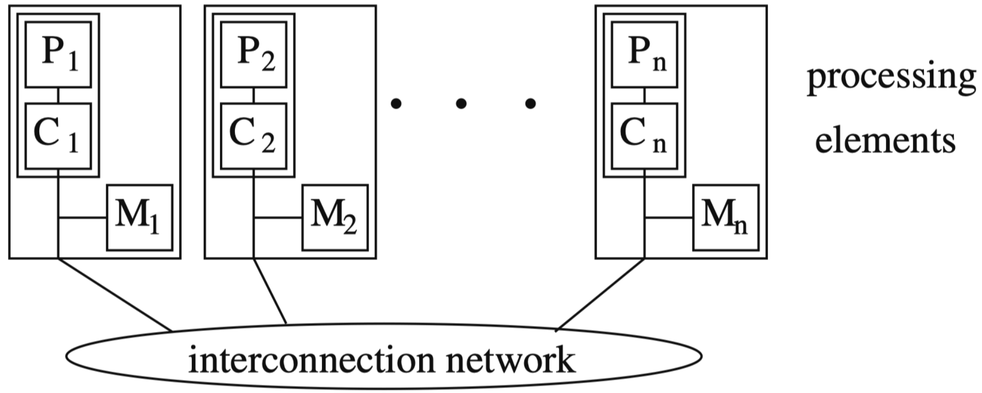
Memory hierarchy



Latency and bandwidth

Memory	Size	Latency	Bandwidth
L1 cache	32 KB	1 nanosecond	1 TB/second
L2 cache	256 KB	4 nanoseconds	1 TB/second Sometimes shared by two cores
L3 cache	8 MB or more	10x slower than L2	>400 GB/second
MCDRAM		2x slower than L3	400 GB/second
Main memory on DDR DIMMs	4 GB-1 TB	Similar to MCDRAM	100 GB/second
Main memory on Cornelis* Omni-Path Fabric	Limited only by cost	Depends on distance	Depends on distance and hardware
I/O devices on memory bus	6 TB	100x-1000x slower than memory	25 GB/second
I/O devices on PCIe bus	Limited only by cost	From less than milliseconds to minutes	GB-TB/hour Depends on distance and hardware

Where is my data?



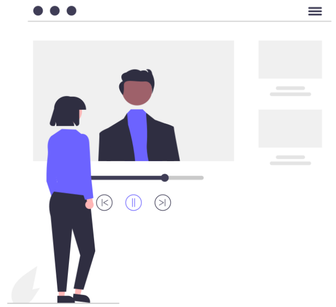
- When running a multi-threaded program, you have multiple processing units working on your computation at the same time.
- They all read from/write to memory.
- In general, it is best if a processing unit works with data that is “close,” that is access to that memory is fast.
- How can we know which memory a piece of data is in and whether it is close to the processing unit?

Memory allocation

- When you allocate dynamic memory, the memory is not immediately allocated in physical memory.
- The OS waits until the first write occurs.
- At that point, the page associated with that piece of data is allocated. By default, it is allocated closest to the processing unit that executes the write.
- **Performance tip: the processing unit working with a memory location the most should be the one touching it first.**
- We will see later how this can be used to write efficient programs.

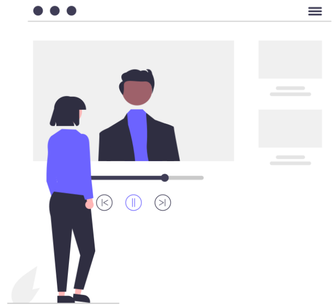
```
void first_touch() {
  const int n = 10000;
  float *A;
  A = new float[n * n];
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j) A[i * n + j] = 0.0f;
}
```

Performance tips: summary



- Memory is key to developing high-performance multicore applications.
- **Memory traffic and time to access** memory are often more important than flops.
- Memory is **hierarchical and complex.**
- Memory traffic should be reduced through algorithmic changes. Goals are:
  - Increase access to **data in cache**
  - **Reduce traffic** between cache and global memory

Advanced tips



1. **Spatial locality:** the memory accesses of a program have a high spatial locality if the program often accesses memory locations with **neighboring addresses** at successive points in time during program execution.

2. **Temporal locality:** the memory accesses of a program have a high temporal locality if it often happens that the same memory location is **accessed multiple times at successive points in time** during program execution.

3. **Memory affinity:** it is best to execute instructions on cores that are closest to the memory they need to access.

Summary

- In this class, we won't have time to dive into these more advanced topics.
- They are important to write high-performance linear algebra library and for applications that need to be highly tuned for performance.
- In practice, it is difficult to optimize complex engineering programs based on all these considerations.
- **Take-home message: performance of multicore processors is a complex topic; assigning more cores does not necessarily lead to computational speed-up.**