TRANSFORMERS: AGE OF PARALLEL MACHINES

(a biased introduction to computer architecture for supercomputing)

Ana Lucia Varbanescu, University of Twente, NL

a.l.varbanescu@utwente.nl

UNIVERSITY OF TWENTE.

What's in a name?

- @AI enthusiasts: this is not about the AI transformers models [1]
- @Movie enthusiasts: this is a word-play on the transformer movies [2]
- @All (the others): this is about how computer architecture and computing systems have been transformed in the past 15 years

[1] Vaswani et al. "Attention Is All You Need" - https://arxiv.org/abs/1706.03762
[2] https://www.imdb.com/list/ls069544665/

Assumptions

- We need computing systems for **high-performance computing**
  - … thus we focus on how machines are built to provide high-performance
  - … and we talk about that in the context of applications

- **Main goal:** best possible performance for our applications in computational science & engineering

- What else is out there (but we won't cover)?
  - Real-time systems – guarantees are everything
  - Embedded systems – efficiency and scale is everything
  - Shared (large) systems (e.g., cloud computing) – sharing is ~~caring~~ everything
  - Computing continuum – a mix of everything from IoT through Edge/Fog to Cloud

Agenda (ambitious)

- Part 1 : The anatomy of supercomputers
- Part 2 : What's in a ~~name~~ node?
- Part 3 : Diversity in parallelism
- Part 4 : One more word about performance
- Part 5 : Summary and beyond
  - Famous last words …

"Larry, do you remember where we buried our hidden agenda?"

PART1: (SUPER)COMPUTING MACHINES

The anatomy of supercomputers

Computer Systems

Simplistic definition

A mix of hardware and software (systems) used to execute applications.

Traditional goals:

- High(er)-performance systems
- Low(er)-power systems
- More efficient systems
- Higher availability systems
- Reliable systems
- Programmable systems
- ….

Computer Systems: examples

Supercomputers

- The most **powerful computers** used for science, technology/engineering and even artificial intelligence

- Typically built as extremely large computer systems, with hundreds of thousands of "basic" components and many billion transistors

- All the processors in a supercomputer can perform computations at the same time => parallel computing.
  - Faster progress than sequential systems …
  - …iff parallel code exists.

Adapted from: https://www.pdc.kth.se/about/what-does-pdc-do-and-how/introduction-to-supercomputers-1.764078

How do we build supercomputers?

- We “replicate” architectural patterns from nodes to blades to racks/cabinets.
- We interconnect each of these components with fast and/or efficient networks.

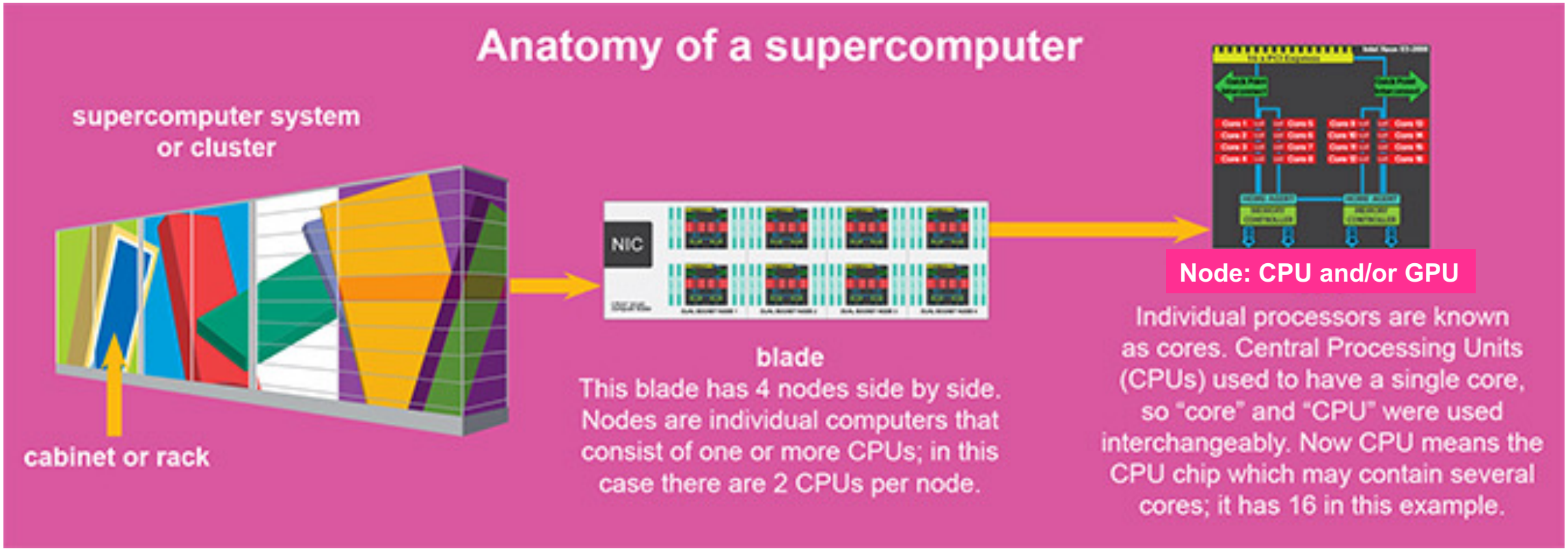


Image from: https://www.pdc.kth.se/about/what-does-pdc-do-and-how/introduction-to-supercomputers-1.764078

(Parallel) Systems Models

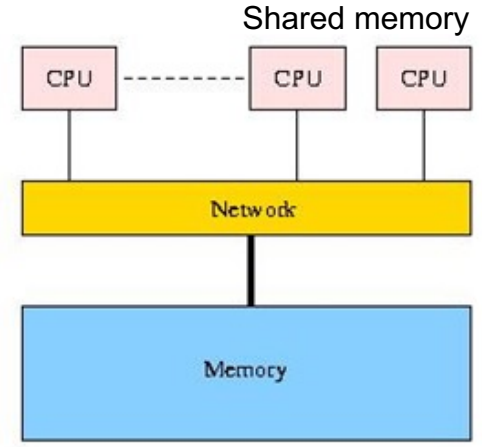
- Why do we need parallel system models?
  - Provide an abstraction of the real machine
  - Dictate the properties of “dedicated” programming models
  - Enable the selection of an appropriate programming model

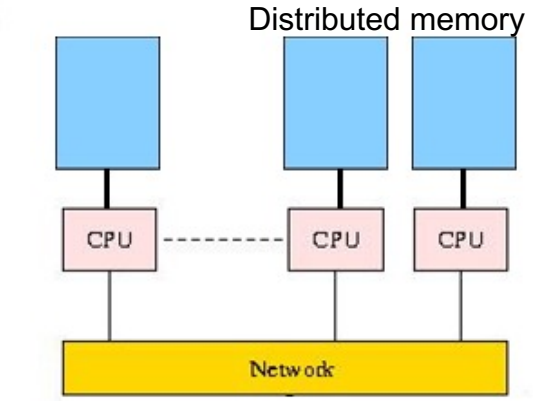
- Organization-based classification
  - Shared Memory
  - Distributed Memory
  - Virtual shared Memory
  - Hybrids

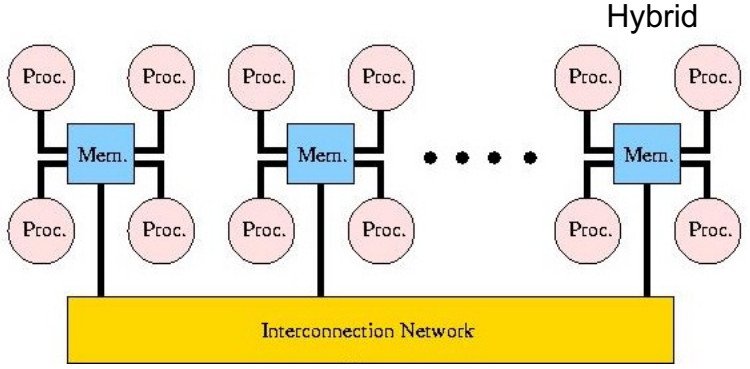
- Processing-based classification
  - Single/Multi Instruction, Single/Multi Data (items)

Parallel Machine Models

- Shared Memory
  - Multiple compute nodes
  - One single shared address space
  - Typical example: multi-cores
- Distributed Memory
  - Multiple compute nodes
  - Multiple, local (disjoint) address spaces
  - **Virtual shared memory:** software/hardware layer “emulates” shared memory
  - Typical example: clusters
- Hybrids
  - Multiple compute nodes, typically heterogeneous
  - Mixed address space(s), some shared, some global memory
  - Typical example: supercomputers







Parallel Machine Models

- Shared Memory

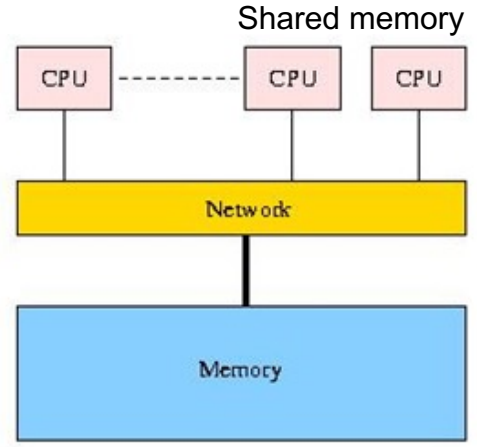
  Programming: multi-threading
  Programming models: OpenMP, pthreads, TBB, …

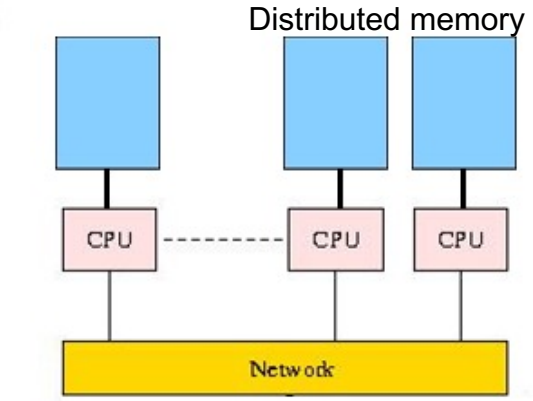
- Distributed Memory

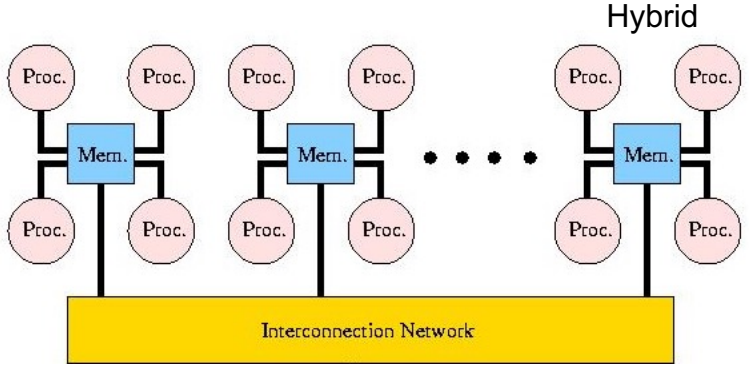
  Programming: message passing
  Programming models: MPI, Big-data models, …

- Hybrids

  Programming: very diverse, depending on the hardware configuration







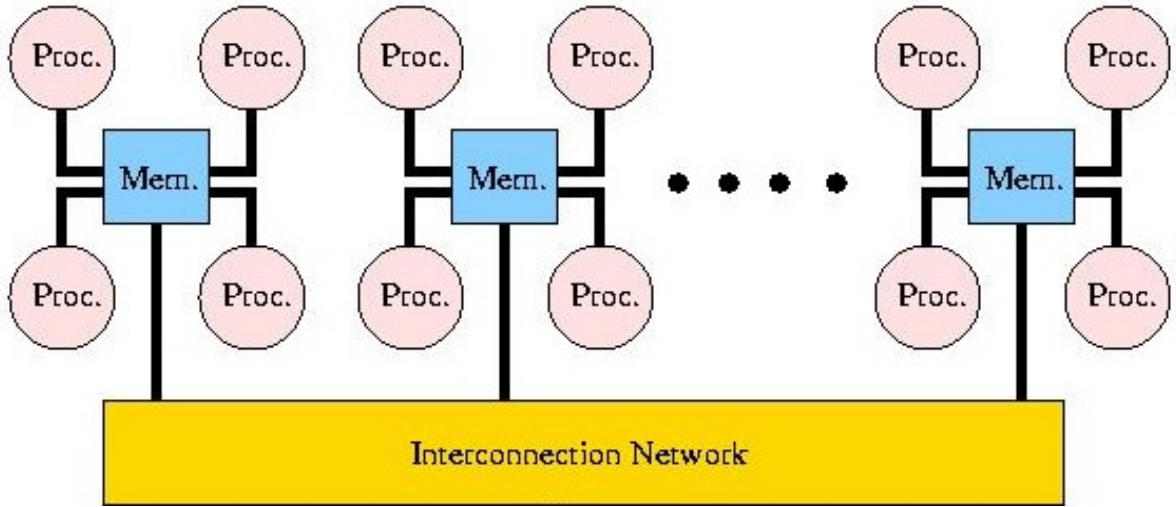
Examples

- Multi-core CPUs ?
  - Shared memory with respect to system memory
  - Hybrid when taking caches into account
- Clusters ?
  - Distributed memory
  - Could be shared if middleware for virtual shared space is provided
- Supercomputers ?
  - Usually hybrid
- GPUs ?
- Architectures with GPUs?
  - Distributed for traditional, off-chip GPUs
  - Shared for new APUs

Main challenge: scaling to ExaFLOPS and beyond

- Peak performance = sum of capabilities of all machines
  - E.g.: 100 nodes x 128 cores x 100GFLOPs/core
- Scaling options:
  - More nodes = scale out
  - More powerful nodes = scale up (or acceleration/heterogeneity)
- Limitations to actual performance
  - Memory, I/O, networking bottlenecks
  - Load-imbalance
  - Non-uniform behaviour
  - Programmability

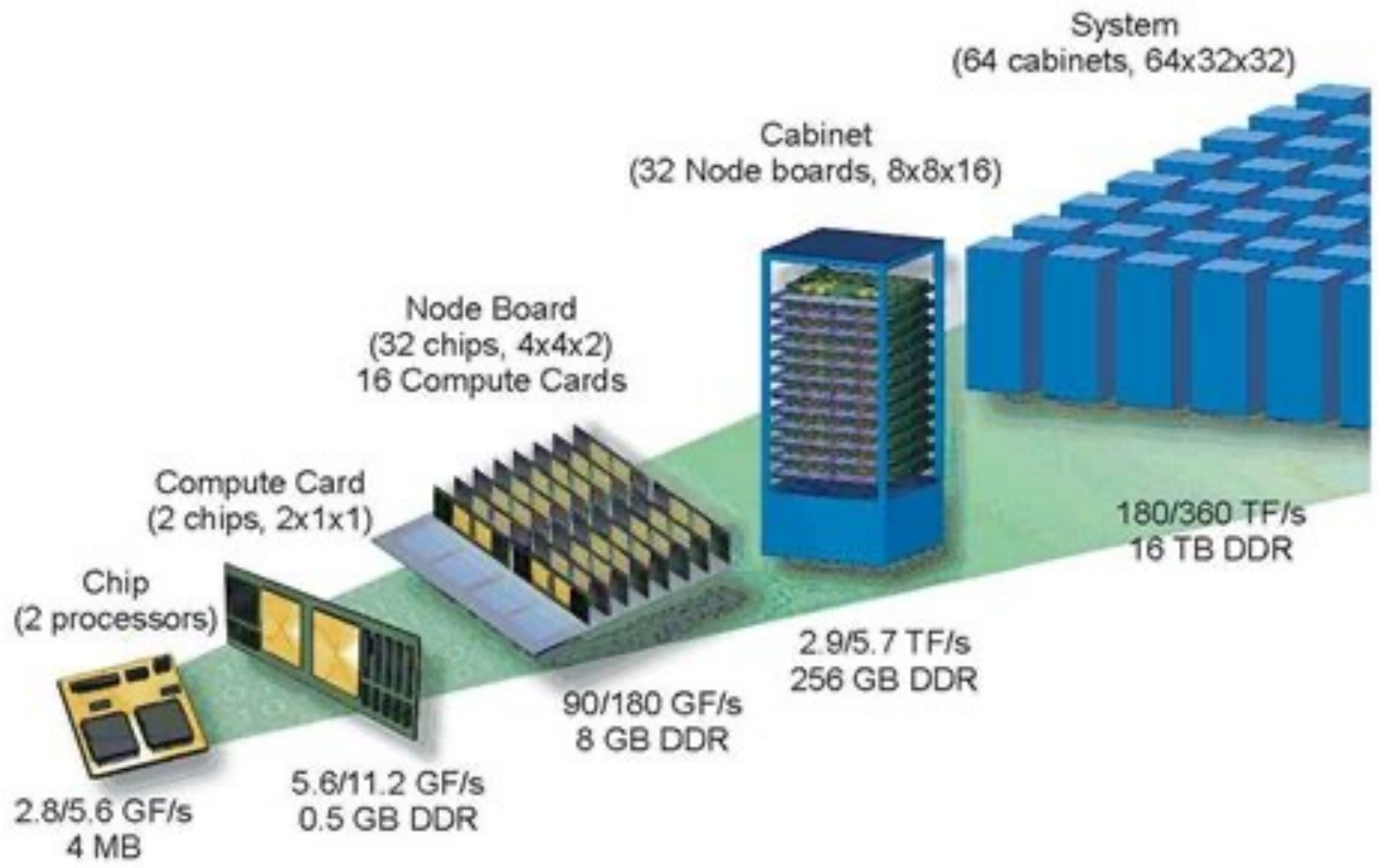
Hybrid



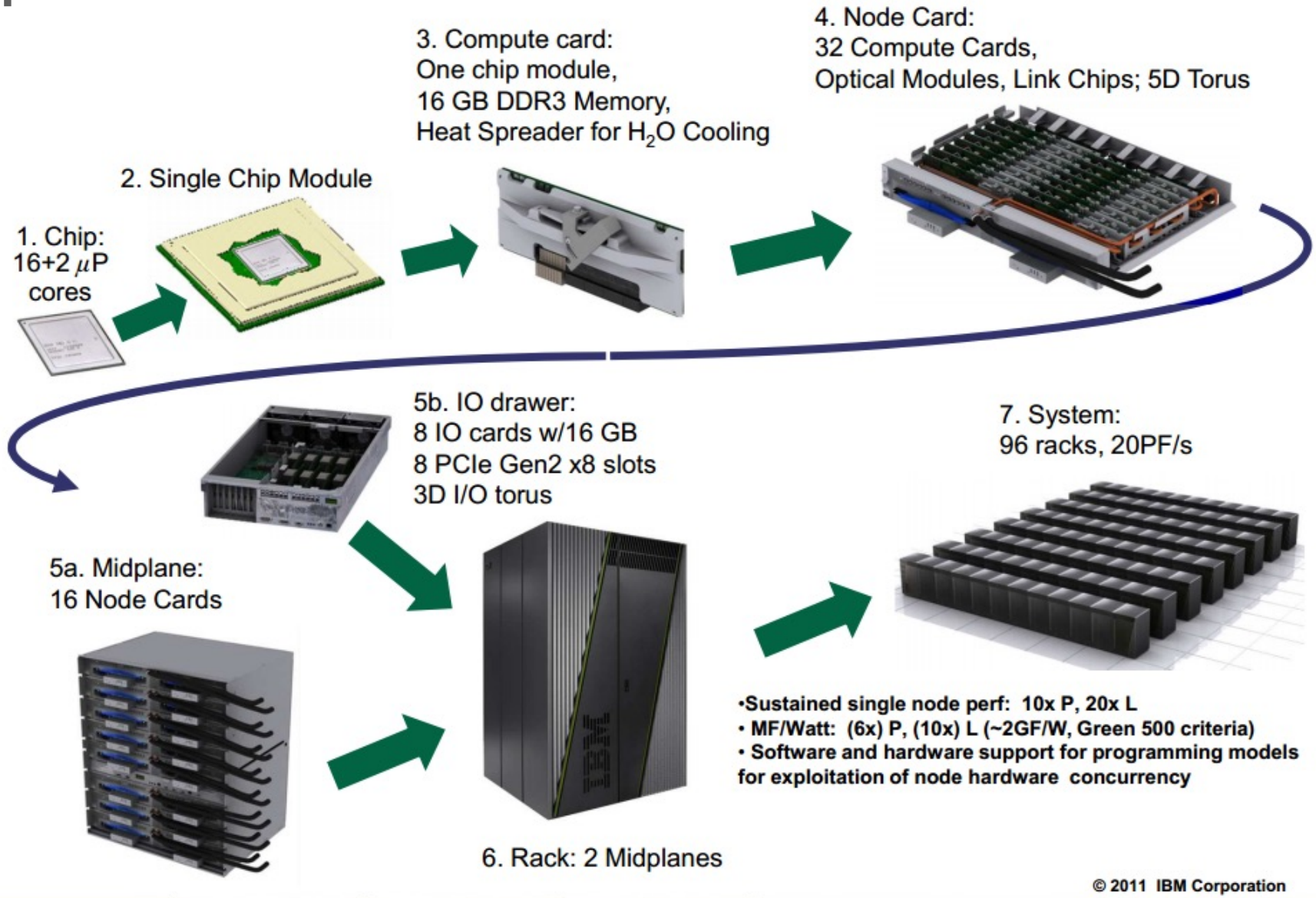
How to scale-up/-out?

- Shared Memory model <= typical for scale-up, limited for scale-out
  - Interconnect scalability problems & uniform accesses
  - Programming challenge: RD/WR Conflicts
- Distributed Memory model <= typical for scale-out, inefficient for scale-up
  - Data distribution is mandatory
  - Programming challenge: remote accesses, consistency
- Virtual Shared Memory model <= increased programmability and overhead
  - Significant virtualization overhead
  - Easier programming
- Hybrid models <= trade-offs at different levels!
  - Local/remote data more difficult to trace

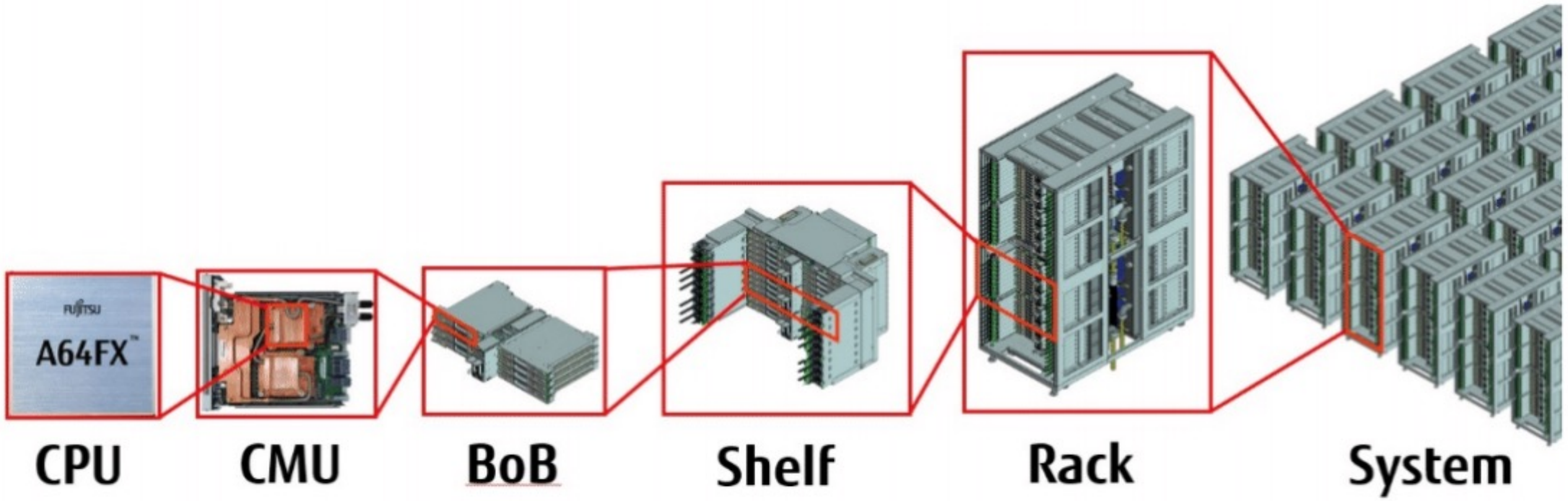
Example: IBM's BLUGENE/L



Example: IBM's BlueGene/Q



Example: FUGAKU



Example: SUMMIT

Summit Overview

OpenPOWER™

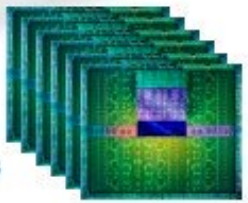
Components

IBM POWER9

- 22 Cores
- 4 Threads/core
- NVLink

NVIDIA GV100

- 7 TF
- 16 GB @ 0.9 TB/s
- NVLink



Compute Node

2 x POWER9
6 x NVIDIA GV100
NVMe-compatible PCIe 1600 GB SSD

25 GB/s EDR IB- (2 ports)
512 GB DRAM- (DDR4)
96 GB HBM- (3D Stacked)
Coherent Shared Memory

Compute Rack

18 Compute Servers
Warm water (70°F direct-cooled components)
RDHX for air-cooled components

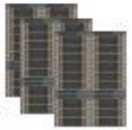
39.7 TB Memory/rack
55 KW max power/rack

Compute System

10.2 PB Total Memory
256 compute racks
4,608 compute nodes
Mellanox EDR IB fabric
200 PFLOPS
~13 MW

GPFS File System

250 PB storage
2.5 TB/s read, 2.5 TB/s write



OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Example: Dardel's CPU partition

- 1278 compute nodes
- 2x AMD EPYC™ Zen2 2.25 GHz 64-core processors/node
  - 128 physical CPU cores/node
  - 2 hardware threads per core => 256 virtual CPU cores/node
- Different memory sizes
  - 700 × 256 GB (NAISS thin nodes)
  - 268 × 512 GB (NAISS large nodes)
  - 8 × 1024 GB (NAISS huge nodes)
  - 18 × 2048 GB (NAISS giant nodes)
  - 36 × 256 GB (KTH industry/business research nodes)
  - 248 × 512 GB (KTH industry/business research nodes)

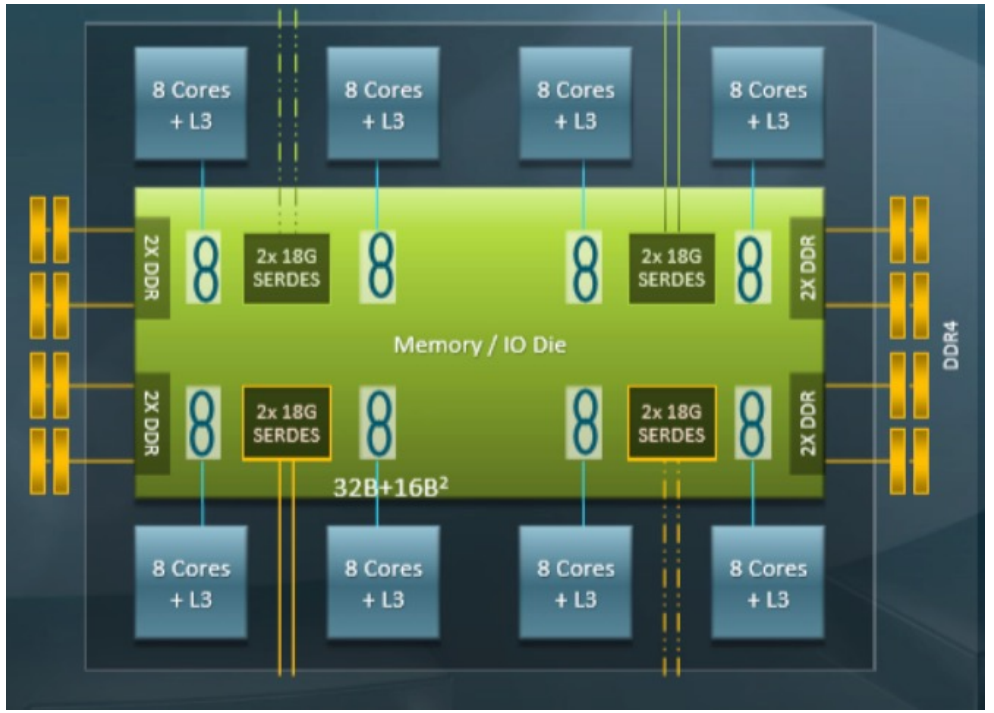
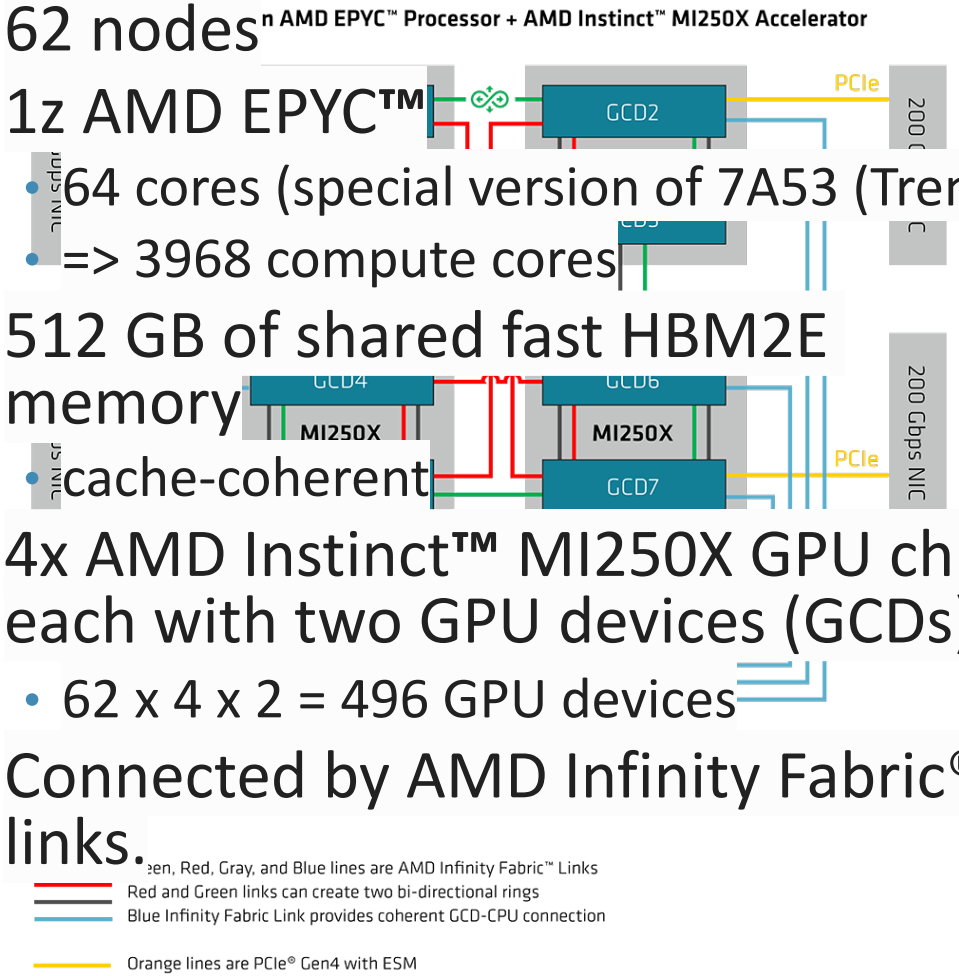


Image from: https://www.nextplatform.com/2019/08/15/a-deep-dive-into-amds-rome-epyc-architecture/
Data from: https://www.pdc.kth.se/hpc-services/computing-systems/about-the-dardel-hpc-system-1.1053338

Example: Dardel's GPU partition

- 62 nodes
- 1z AMD EPYC™
  - 64 cores (special version of 7A53 (Trento)
  - => 3968 compute cores
- 512 GB of shared fast HBM2E memory
  - cache-coherent
- 4x AMD Instinct™ MI250X GPU chips, each with two GPU devices (GCDs)
  - 62 x 4 x 2 = 496 GPU devices
- Connected by AMD Infinity Fabric® links.

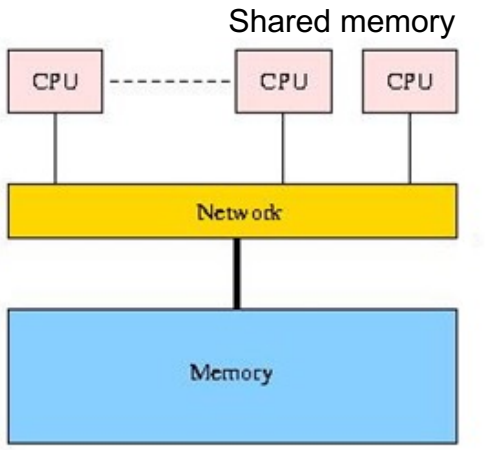


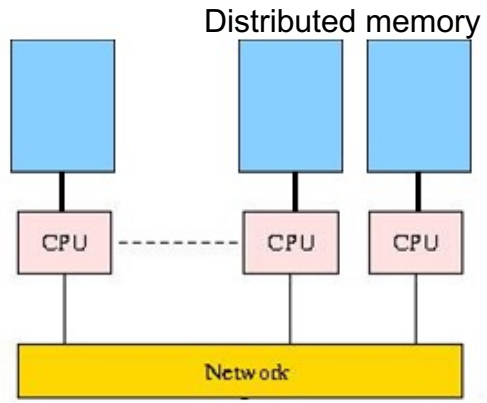
Data from: https://www.pdc.kth.se/hpc-services/computing-systems/about-the-dardel-hpc-system-1.1053338

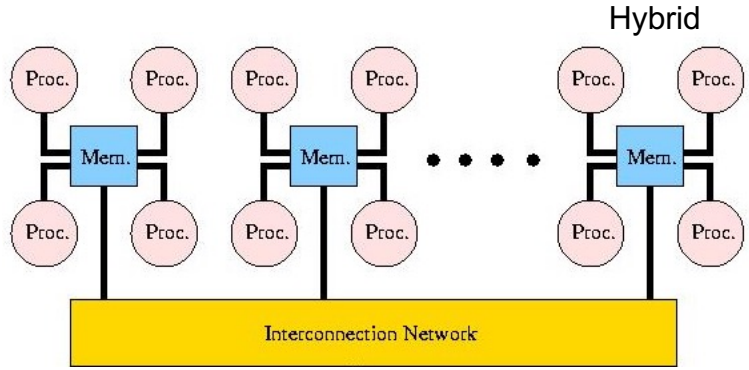
Why should we care?

- Application scaling + programming ⇔ system architecture

- E.g.: calculate the histogram of a very large dataset in a small number of bins.







Programmer vs. runtime/OS vs. job scheduler

- Programmer exposes parallelism at application level
  - Job = application + dataset
  - Application = set of tasks
  - Tasks = execute in some sequential order and/or in parallel
- Runtime/OS map the tasks on resources
  - In both space and time
  - Possibly with programmer's restrictions
- (Job) Scheduler ensures jobs are allocated resources
  - Ideally sufficient and "localized"

How to split and program the tasks? How is data accessed?

Knowledge of node architecture is essential for effective optimization.

What runs where and when?

Decisions by a runtime system and/or OS; require deep knowledge system architecture.

What resources are allocated?

Decisions by a job scheduler to maximize utilization/performance.

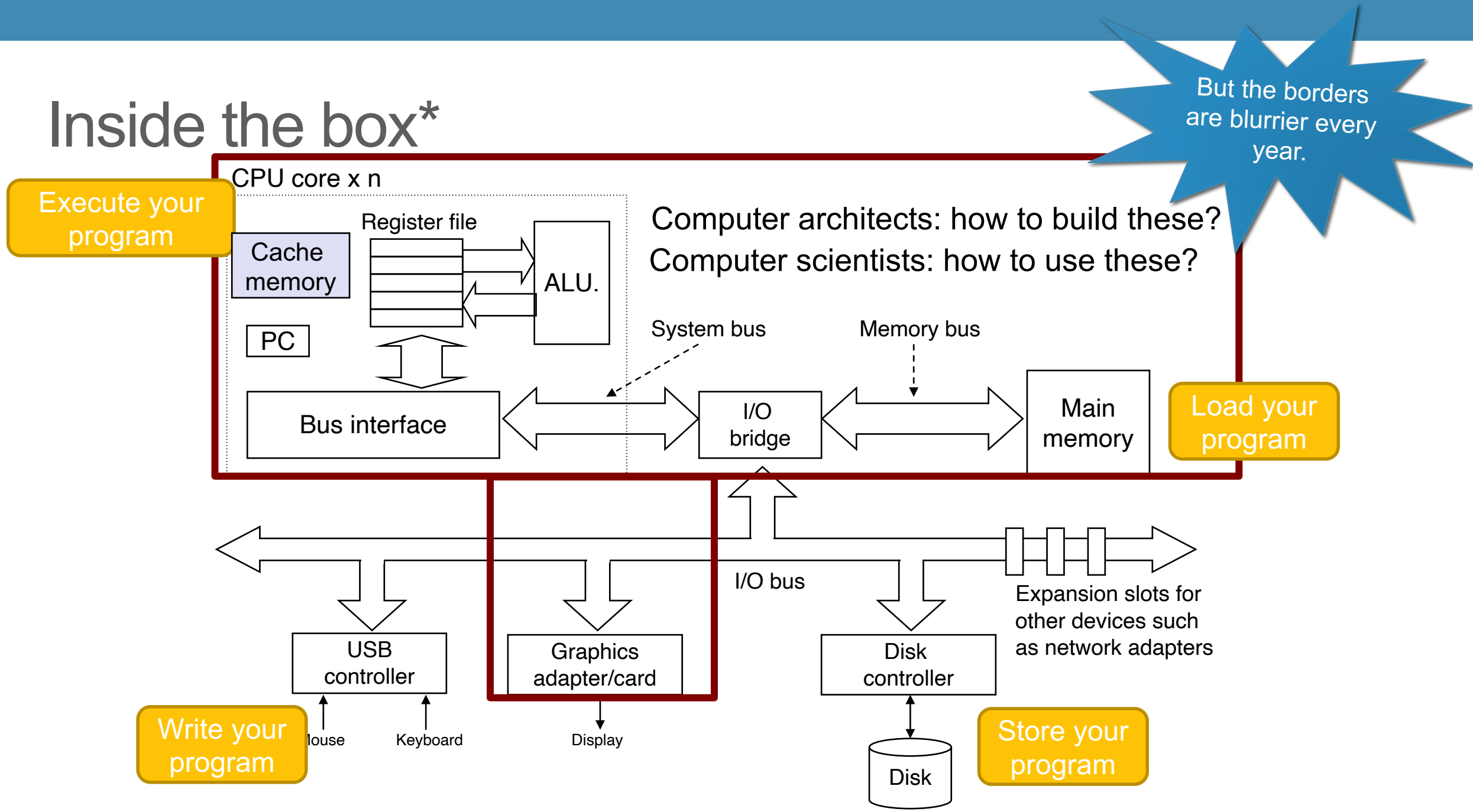
In summary

- Supercomputers are “organized” collections of compute nodes
- Compute nodes are “organized” collections of compute cores, possibly heterogeneous
- “Organized” = architectural patterns + communication technologies
- Overall theoretical performance = “peak performance” is the sum of the parts
  - The assumption is they *all* *work independently* *in parallel*
- Performance = peak performance – Σ(system bottlenecks, app overhead)

PART2: WHAT'S IN A ~~NAME~~ NODE?

Computer systems basics.

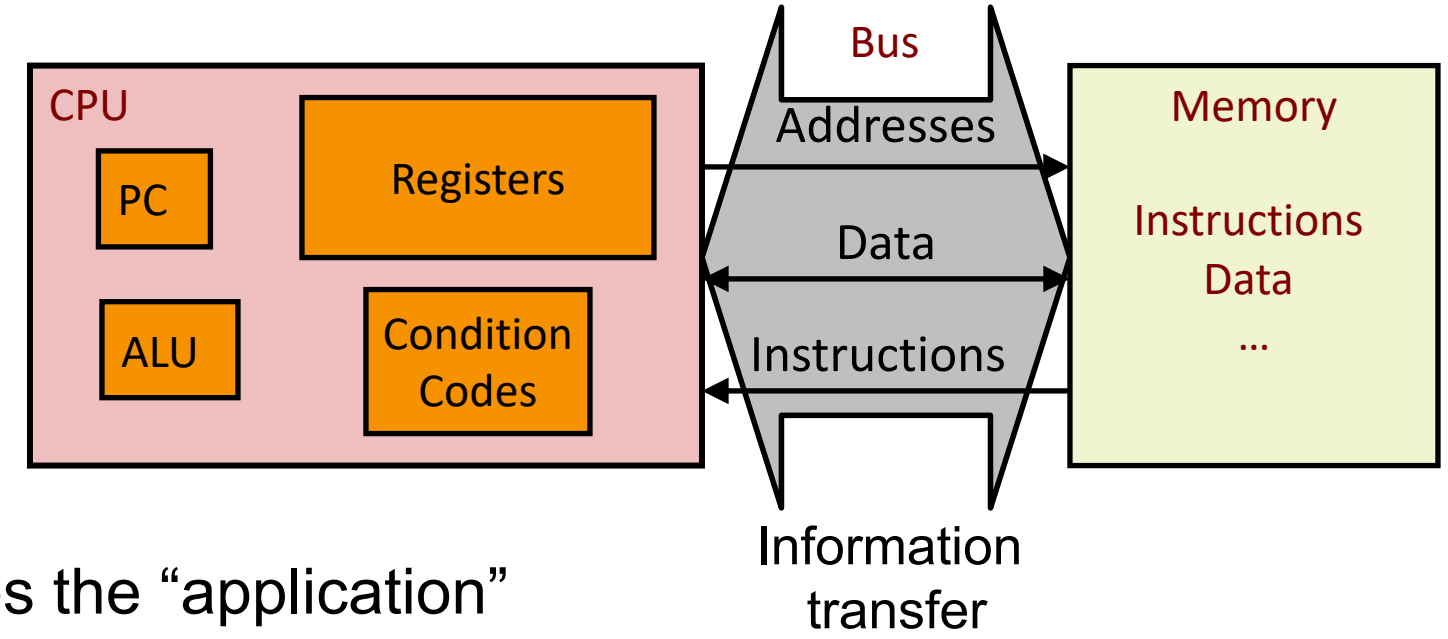
Inside the box*



*Adapted from “Computer Systems: A Programmer’s View” (Ch1) by Bryant and O’Hallaron

Processor basic operation

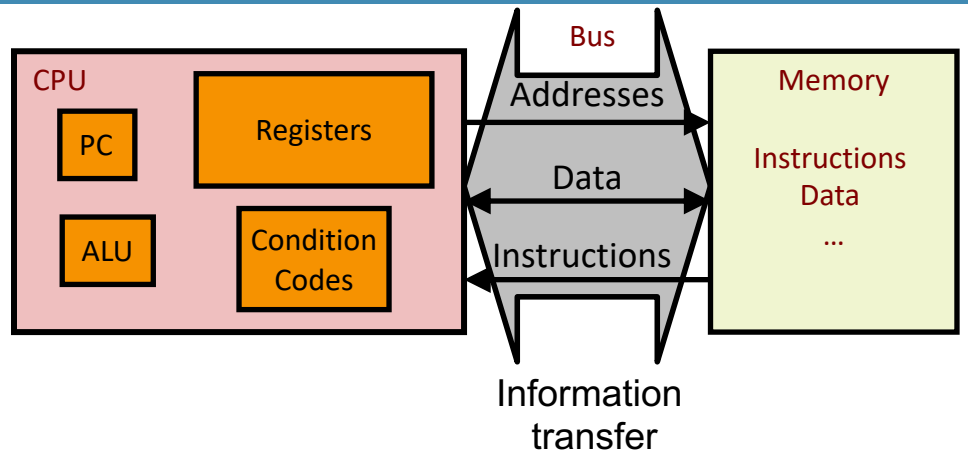
A processor's inner workings



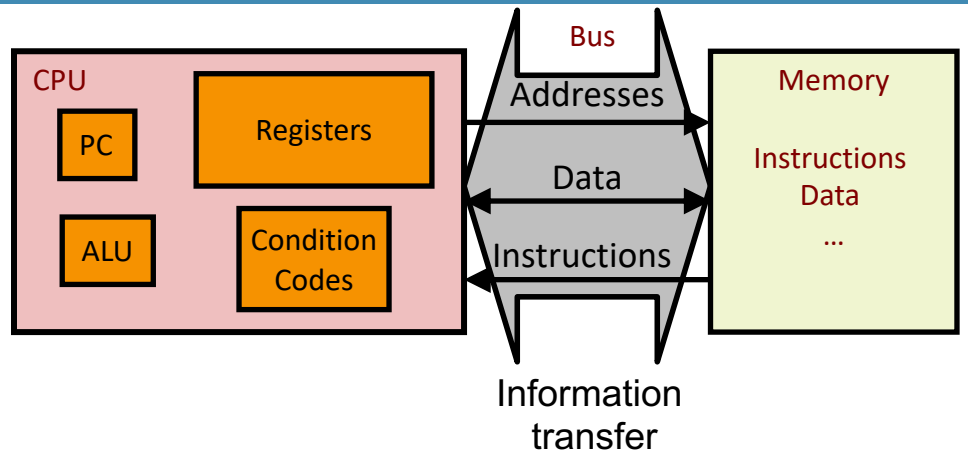
- CPU = executes the "application"
  - Manages the execution progress (PC)
  - Fetches needed instructions and data (addresses)
  - Executes (ALU) operations and manages results
- Memory = stores the executable code of the application and the data
  - Receives request + address, replies with data (a bit vector)
- Bus = facilitates information (=bits) movement

The CPU

- Computations are executed by the ALU
  - Integer, single/double precision arithmetic, …
  - Comparisons, logical operations, …
- ALU runs at its own “clock speed” / frequency
  - Defines how many cycles/s can be executed by the CPU
  - Each operation takes 1 or more cycles
- Higher performance CPUs
  - Make a faster/smarter ALU
    - More operations per cycle
  - Make faster CPUs
    - More cycles/s
  - Multiple cores
    - Even more operations per cycle!



The memory



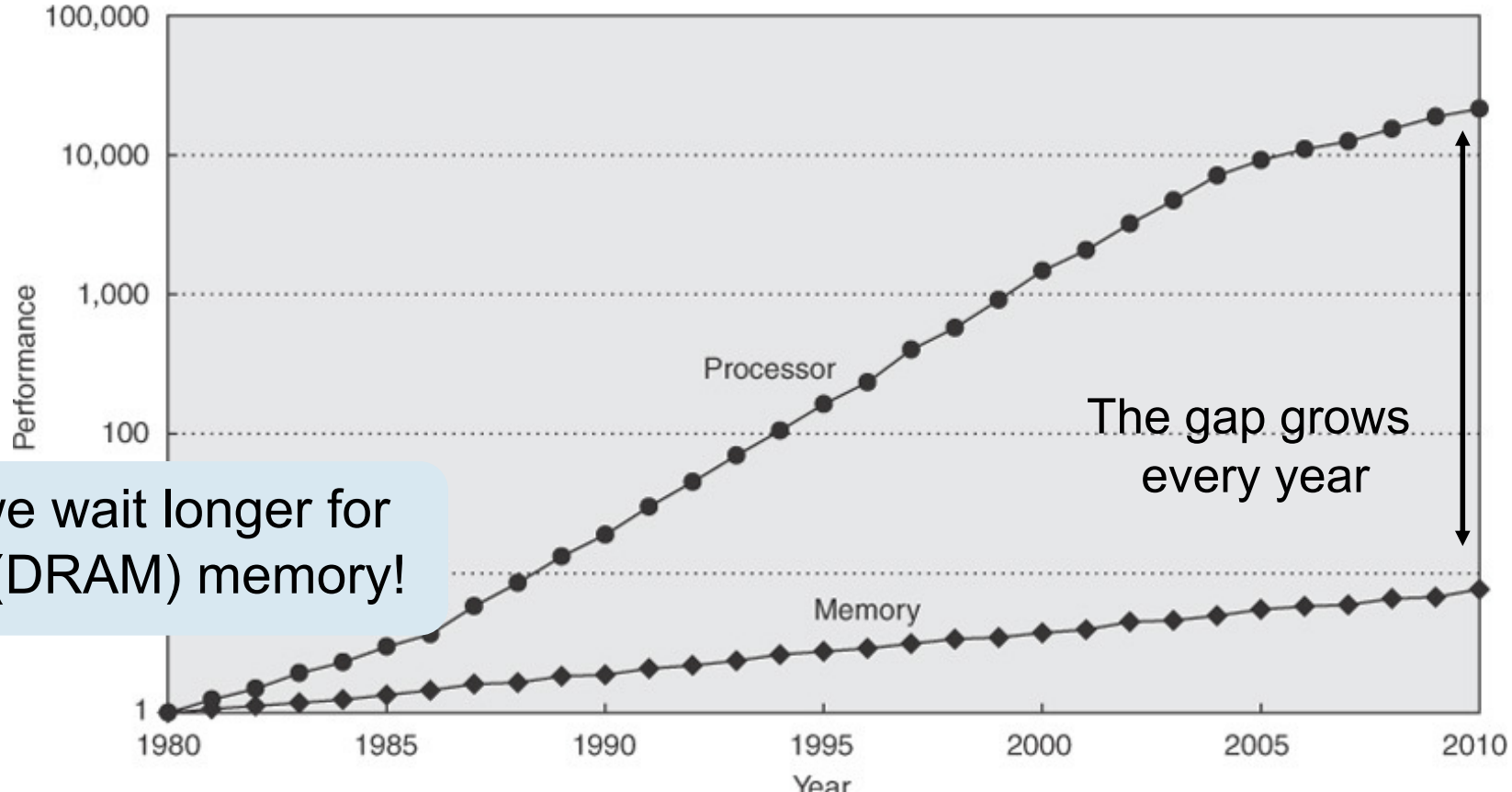
- Typically organized as linear spaces
  - Some word-size granularity

- Code and data are stored in memory
- Everything that lives in memory has an “address”
  - Visible at assembly level
  - Accessible via pointers/variable names/… from the program itself

- Memory operations are slow!
  - Off-chip
  - Request read/write
  - Search and find

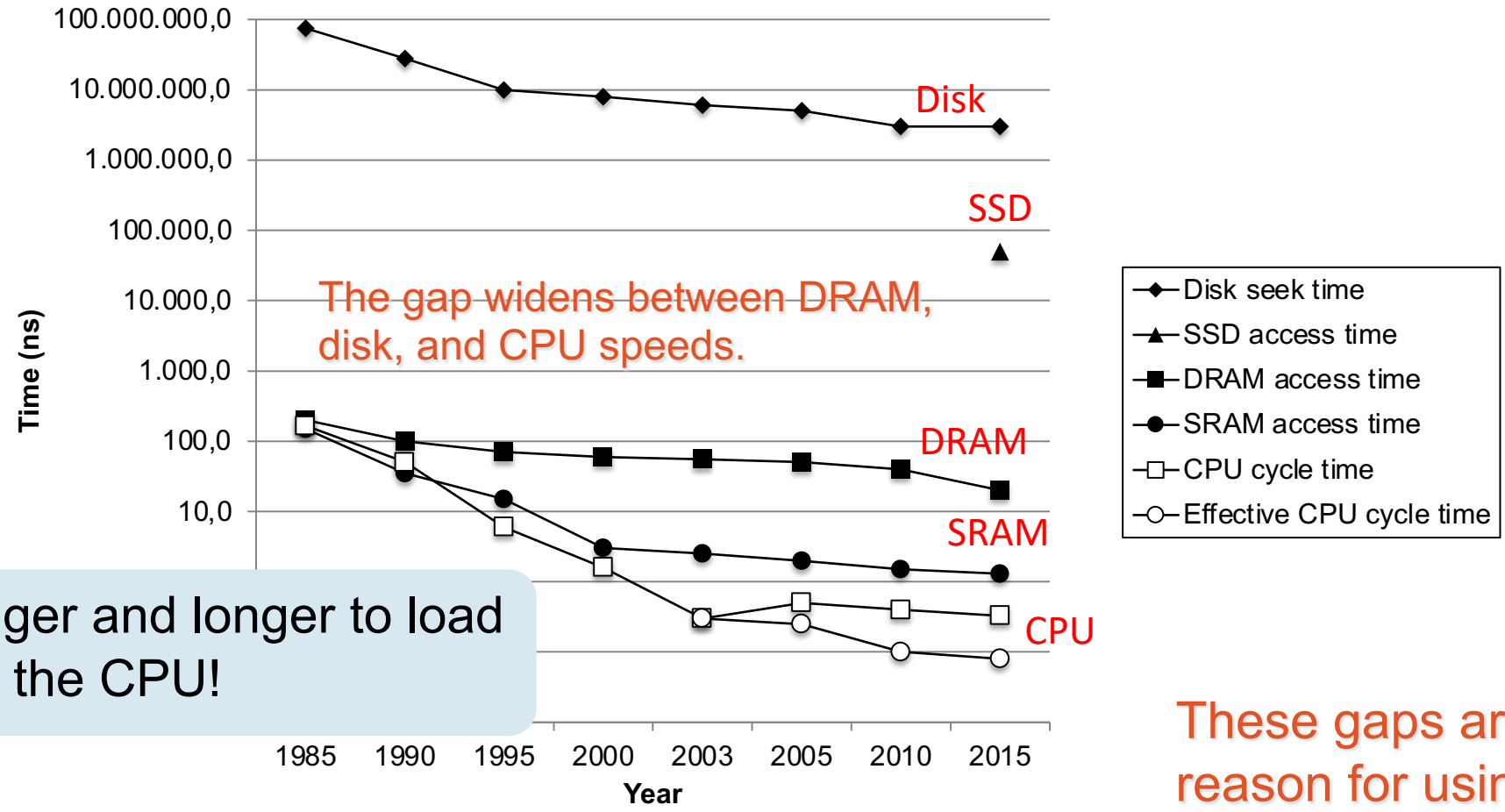
The CPU-Memory Gap

- Flat memory model
  - All accesses = same latency
  - Memory latency slower to improve than processor speed



… which means we wait longer for any access to the (DRAM) memory!

The CPU-Memories Gap

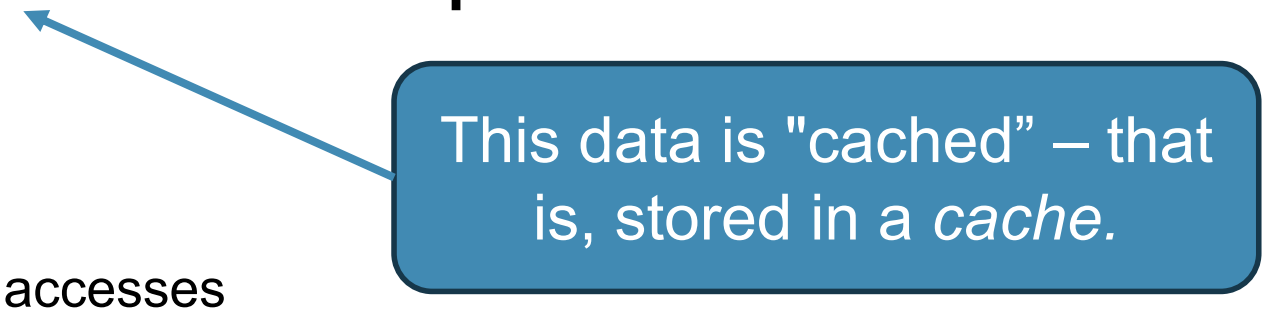


Data takes longer and longer to load to the CPU!

These gaps are the main reason for using a memory hierarchy.

Memory hierarchy

- A single memory for the entire system is not efficient!
- Several memory spaces
  - Large size, low cost, high latency – main memory
  - Small size, high cost, low latency – caches / registers
- Main idea: Bring **some of the data closer to the processor**
  - Smaller latency => faster access
  - Smaller capacity => not all data fits!
- Who can benefit?
  - Applications with **locality** in their data accesses
    - Spatial locality
    - Temporal locality



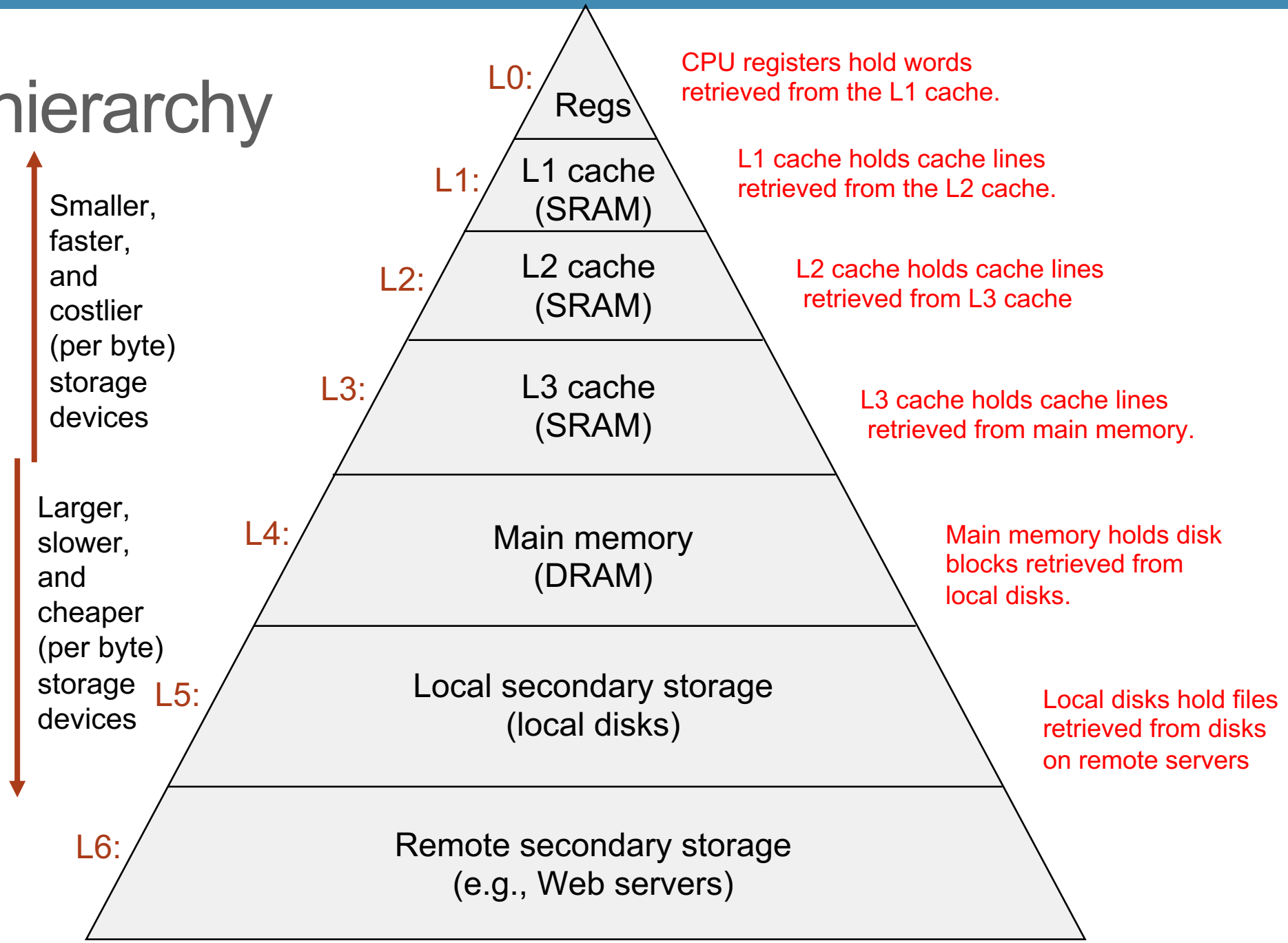
Memory hierarchy and caches

- **Cache:** A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.

- Memory hierarchy
  - Multiple layers of memory, from **small & fast** (lower levels) to **large & slow** (higher levels)
  - *For each k, the faster, smaller device at level k is a **cache** for the larger, slower device at level k+1.*

- How/why do memory hierarchies work?
  - **Locality** => data at level k is used more often than data at level k+1.
    - Level k+1 can be slower, and thus larger and cheaper.

Memory hierarchy



Caching in the Memory Hierarchy

Cache Type	What is Cached?	Where is it Cached?	Latency (cycles)	Managed By
Registers	4-8 bytes words	CPU core	0	Compiler
TLB	Address translations	On-Chip TLB	0	Hardware MMU
L1 cache	64-byte blocks	On-Chip L1	4	Hardware
L2 cache	64-byte blocks	On-Chip L2	10	Hardware
Virtual Memory	4-KB pages	Main memory	100	Hardware + OS
Buffer cache	Parts of files	Main memory	100	OS
Disk cache	Disk sectors	Disk controller	100,000	Disk firmware
Network buffer cache	Parts of files	Local disk	10,000,000	NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

Memory hierarchy

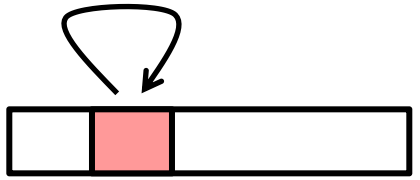
- Challenges
  - Size: no space for every memory address
  - Organization: what gets loaded & where ?
  - Policies: who's in, who's out, when, why?

- Performance
  - Hit = access found data in fast memory => low latency
  - Miss = data not in fast memory => high latency + penalty
  - **Metric**: hit ratio (H) = the fraction of accesses that hit => the higher the ratio, the better the performance!

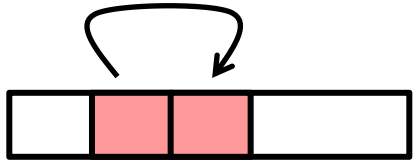
Locality

- Principle of Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently

- Temporal locality:
  - Recently referenced items are likely to be referenced again in the near future



- Spatial locality:
  - Items with nearby addresses tend to be referenced close together in time



Locality Example

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

- Data references
  - Reference array elements in succession (stride-1 reference pattern). — Spatial locality
  - Reference variable `sum` each iteration. — Temporal locality
- Instruction references
  - Reference instructions in sequence. — Spatial locality
  - Cycle through loop repeatedly. — Temporal locality

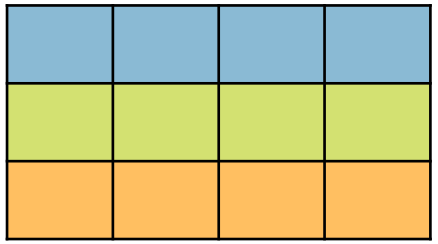
Qualitative estimates of locality

- Question: Does this function have good locality with respect to array `a`?

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

Matrix



Row-major in memory

Every read from the matrix fetches a cache line => assume 4 elements
Assume row-major order and N,M very large => reading a[0][0] will bring in blue elements, while reading a[1][0] will need red elements.
This is **poor locality** – not reusing the same or close-by elements.

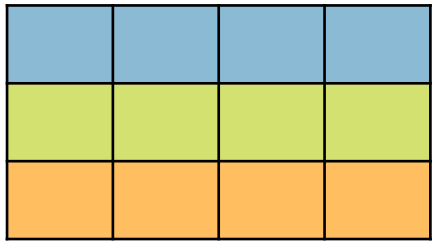
Qualitative estimates of locality

- Question: Does this function have good locality with respect to array `a`?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

Matrix



Row-major in memory

Every read from the matrix fetches a cache line => assume 4 elements
Assume row-major order and N,M very large => reading a[0][0] will bring in blue elements and reading a[0][1]..a[0][3] will need blue elements.
This is **great locality** – reusing the same or close-by elements.

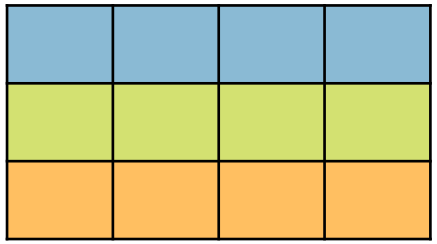
Qualitative estimates of locality

- Question: Does this function have good locality with respect to array `a`?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M/2; i++)
        for (j = 0; j < N/4; j++)
            sum += a[i*2][j*4];
    return sum;
}
```

Matrix

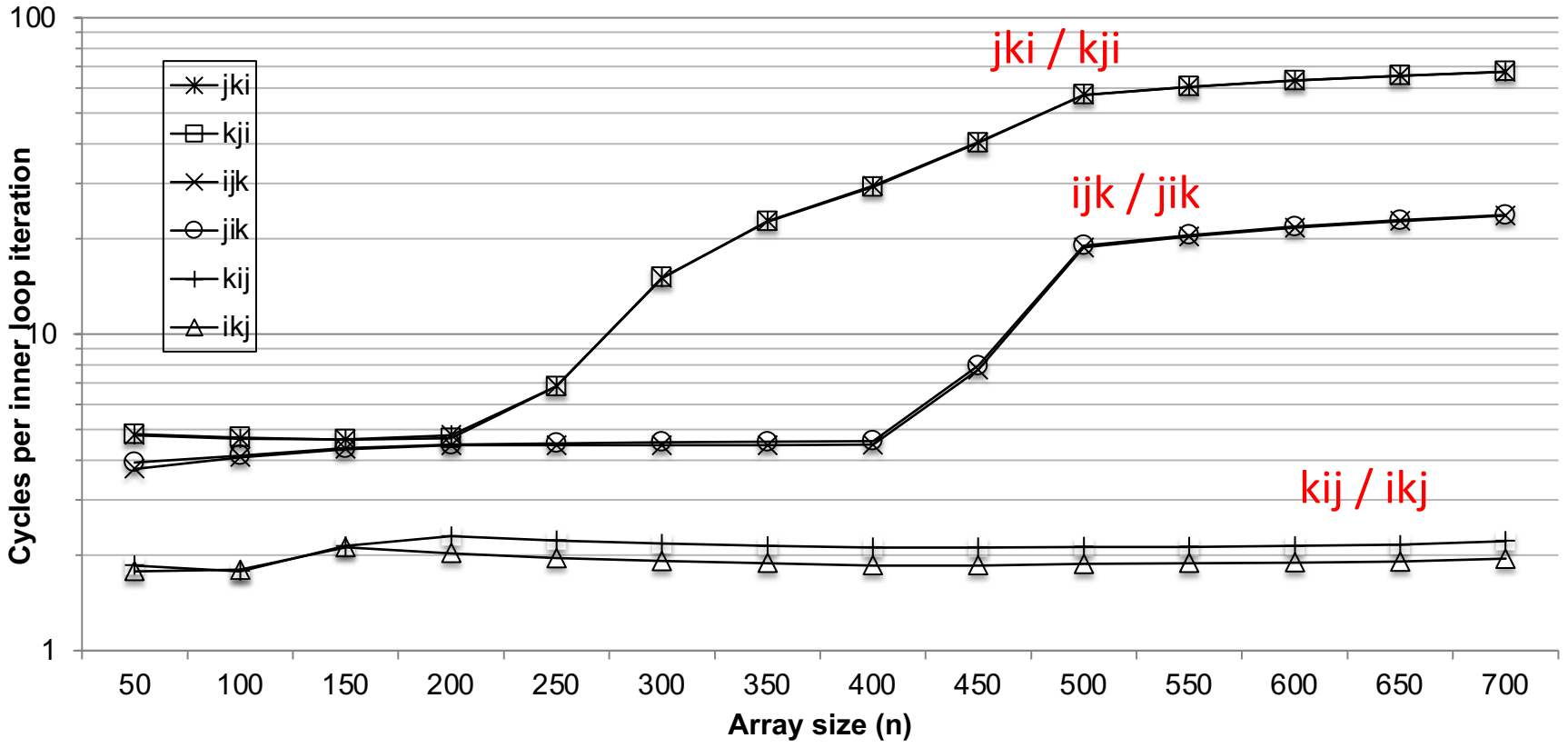


Row-major in memory

Every read from the matrix fetches a cache line => assume 4 elements
Assume row-major order and N,M very large => reading a[0][0] will bring in blue elements, while reading such scattered data from a further will need different colors. This is **non-perfect locality** – depends on sizes …

Matrix Multiplication

Good vs bad locality / caching …




```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
   sum = 0.0;
   for (k=0; k<n; k++)
     sum += a[i][k] * b[k][j];
   c[i][j] = sum;
 }
}
```
ijk / jik

```
for (k=0; k<n; k++) {
 for (i=0; i<n; i++) {
  r = a[i][k];
  for (j=0; j<n; j++)
   c[i][j] += r * b[k][j];
 }
}
```
kij / ikj

```
for (j=0; j<n; j++) {
 for (k=0; k<n; k++) {
   r = b[k][j];
   for (i=0; i<n; i++)
    c[i][j] += a[i][k] * r;
 }
}
```
jki / kji

Main challenges

- Compute and memory performance grow at different speeds
  - Caching is the current way
  - Technology will eventually improve latency and bandwidth

- For high performance
  - Take care of the data size
  - Organize data in memory to allow for high performance = memory layout
  - Make use of caching = memory access patterns

Memory operations are the main bottleneck in most HPC today!
Check your **data memory layout** and **access patterns** to improve **locality!!**

In summary: Computing systems basics …

- … are essential for the building HPC systems
- … and for programming them

- Be literate in these topics ☺
  - Caching
  - Processing
  - Data representation
  - Instructions

- … else you will have trouble programming these machines efficiently.