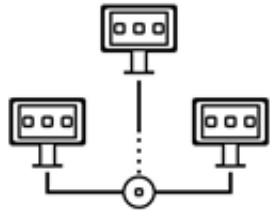
CCZG501

Introduction to
Parallel and Distributed Programming

CS1
Introduction

Dr. Saikishor Jangiti



Agenda

- Parallel and Distributed Systems (PDS)
- Libraries and APIs for Programming PDS
- Cloud and Big Data System use cases in need of programming a PDS

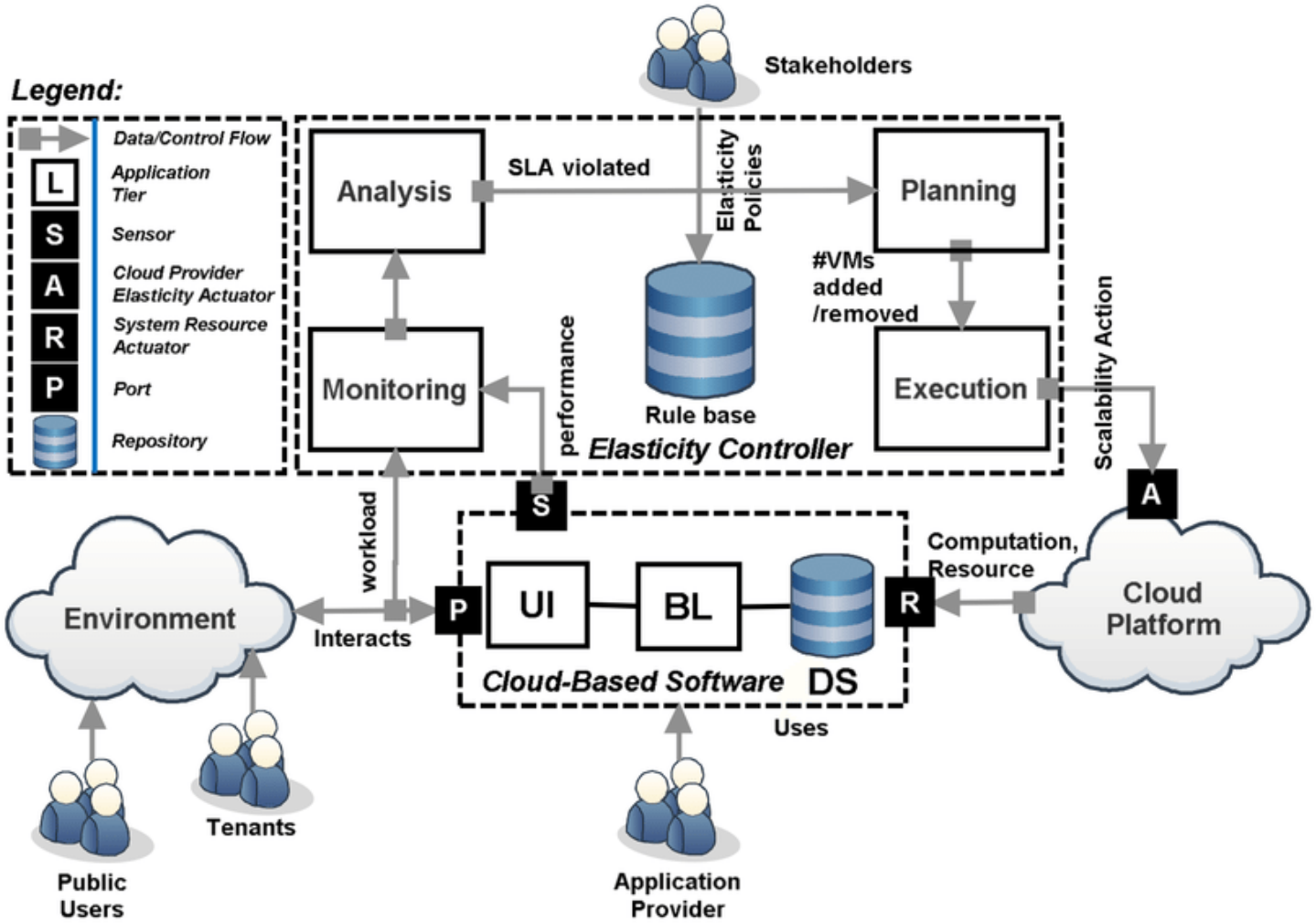
CLOUD COMPUTING

NIST, the US National Institute of Standards and Technology, defined cloud computing as

"a model for enabling ubiquitous, convenient, on-demand network access to a ***shared pool of configurable computing resources*** (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with ***minimal management effort or service provider interaction***."

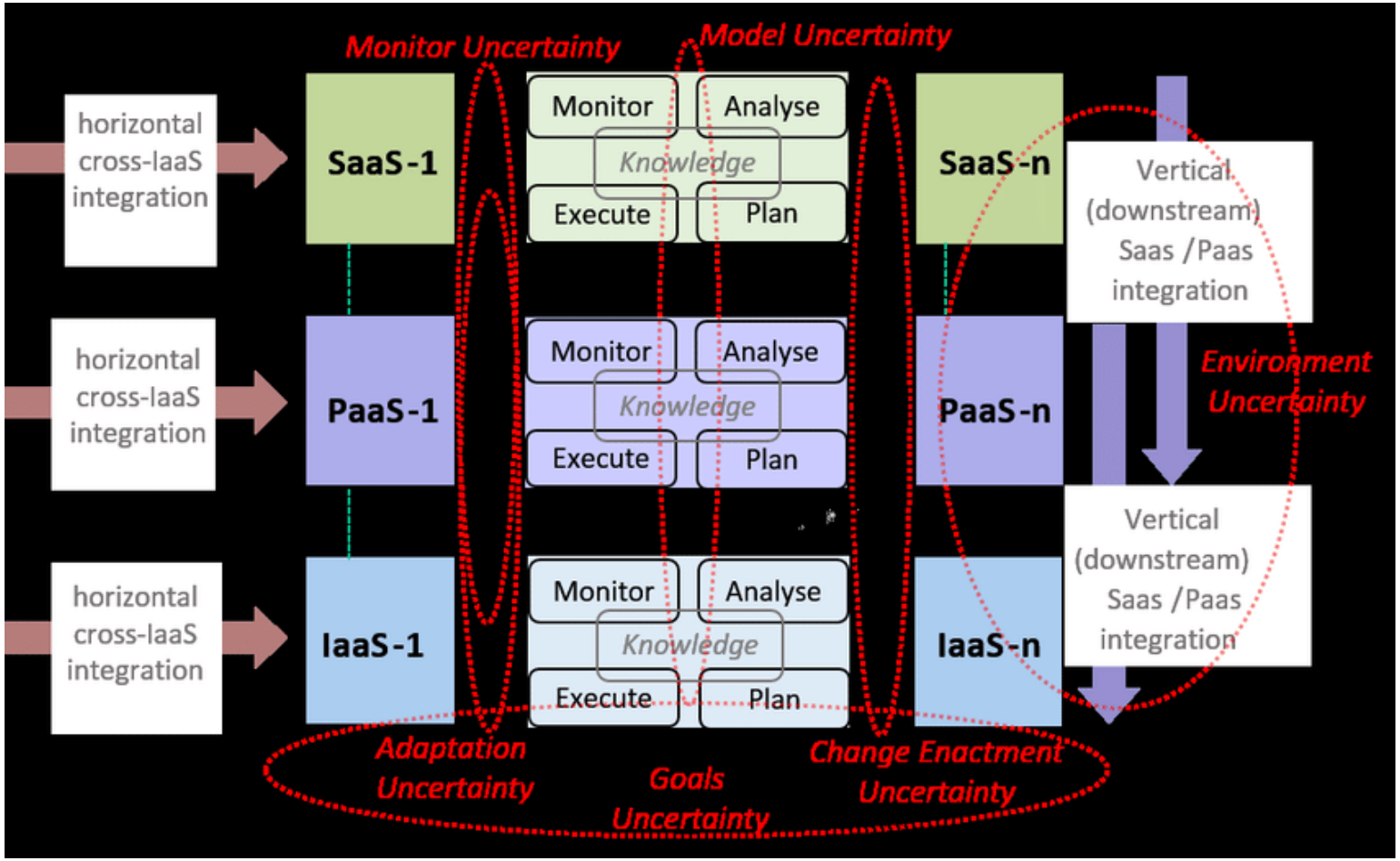
Hence, a Cloud has to self-manage its large pool of resources

CLOUD COMPUTING



Source : Pahl, Claus & Jamshidi, Pooyan & Zimmermann, Olaf. (2017). Architectural Principles for Cloud Software. ACM Transactions on Internet Technology. 18. 10.1145/3104028.

CLOUD COMPUTING



Source : Pahl, Claus & Jamshidi, Pooyan & Zimmermann, Olaf. (2017). Architectural Principles for Cloud Software. ACM Transactions on Internet Technology. 18. 10.1145/3104028.

Open Nebula received the following requirements from many leading supercomputing and research centers

- **Automatic and Elastic Management of the Computing Service**. The powerful CLI and APIs exposed by OpenNebula enable its easy integration with any of the most common job management systems (Torque, Open Grid Engine, Platform LSF…) to **automatically provision computing worker nodes** to meet dynamic demands. The advanced contextualization mechanisms enable the **automatic configuration** of the worker nodes.
- **Combination of Physical and Virtual Resources**. You can use OpenNebula for the management of the virtual worker nodes in your computer cluster and **keep physical resources for HPC performance-sensitive applications** that require high-bandwidth, low-latency interconnection networks.
- **Management of Several Physical Clusters with Different Configurations.** The multiple-zone functionality enables the management of multiple physical clusters with specific architecture and software/hardware execution environments to fulfill the needs from different workload profiles.
- **Support for Several VOs**. The new functionality to on-demand provision of Virtual Data Centers can be used to provide different VOs with isolated compartments of the cloud infrastructure.
- **Support for Heterogeneous Execution Environments**. The new repositories for VM appliances and templates can be used to provide users with pre-defined application environments. The fined-grain access control supports the creation and easy maintenance of the appliance repositories, that could be even private for different Virtual Data Centers (VOs).
- **Full Isolation of Execution for Performance-sensitive Applications**: The functionality for **automatic placement** of VMs and the configurable monitoring system enable the ability to define isolation levels for the computing services. The new multiple-zone support extends this functionality to easily manage fully isolated physical clusters.
- **Execution of Complete Computing Clusters**: Can deploy multi-tier services consisting of groups of inter-connected VMs and define their **auto-configuration** at boot time.
- **Cloudbursting to Meet Peak Demands**: The hybrid cloud functionality enables the deployment of architectures for **cloud-bursting** to address peak or fluctuating demands of HTC (High Throughput Computing) workloads.
- **Management of Persistent Scientific Data**: Can make disks persistent, save changes for subsequent new executions, and share the new disks with other users in your Virtual Data Center (VO).
- **Placement of VMs Near the Input Data**: OpenNebula's scheduler provides **automatic VM placement** for the definition of workload and **resource-aware allocation policies such as packing, striping, load-aware, or affinity-aware**.
- **Ensure that each Tenant Gets a Fair Share of Resources**: OpenNebula's resource quota management helps allocate, track and limit resource utilization.

Distributed system

"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable." - Leslie Lamport

*A distributed system is a collection of independent entities that cooperate to solve a problem that **cannot be individually solved***

*A distributed execution is the execution of processes across the distributed system to **collaboratively achieve a common goal***

The motivation

1. **Inherently distributed computations :** Money transfer in banking
2. **Resource sharing :** Printer, Network attached Storage, databases, files, distributed databases
3. **Access to geographically remote data and resources :** payroll data, supercomputers
4. **Enhanced reliability :** Geographically distributed resources are not likely to crash/malfunction at the same time
   Reliability entails several aspects:
   - availability, i.e., the resource should be accessible at all times;
   - integrity, i.e., the value/state of the resource should be correct, in the face of concurrent access from multiple processors
   - fault-tolerance, i.e., the ability to recover from system failures
5. **Increased performance/cost ratio**
6. **Scalability** adding more processors does not pose a direct bottleneck for the communication network.
7. **Modularity and incremental expandability** Heterogeneous processors may be easily added into the system

Characteristics of DS

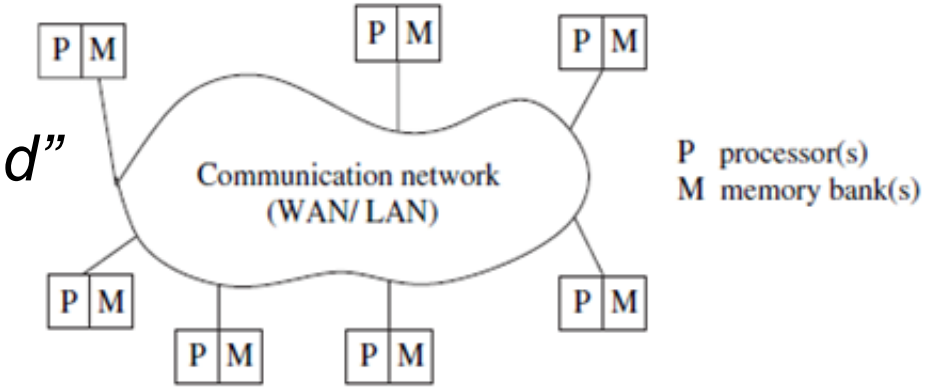
A distributed system can be characterized as a collection of mostly autonomous processors communicating over a communication network and having the following features:

No common physical clock

No shared memory - *requires message-passing for communication*

Geographical separation - *WAN / LAN*

Autonomy and heterogeneity - *“loosely coupled”*



Characteristics of parallel systems

The primary and most efficacious use of ***parallel systems is for obtaining a higher throughput by dividing the computational workload among the processors****.*

The ***tasks*** *that are most amenable to higher speedups on parallel systems are those that can be partitioned into subtasks very nicely,* ***involving much number-crunching and relatively little communication for synchronization.***

A parallel system may be broadly classified as belonging to one of three types:

1. Multiprocessor system

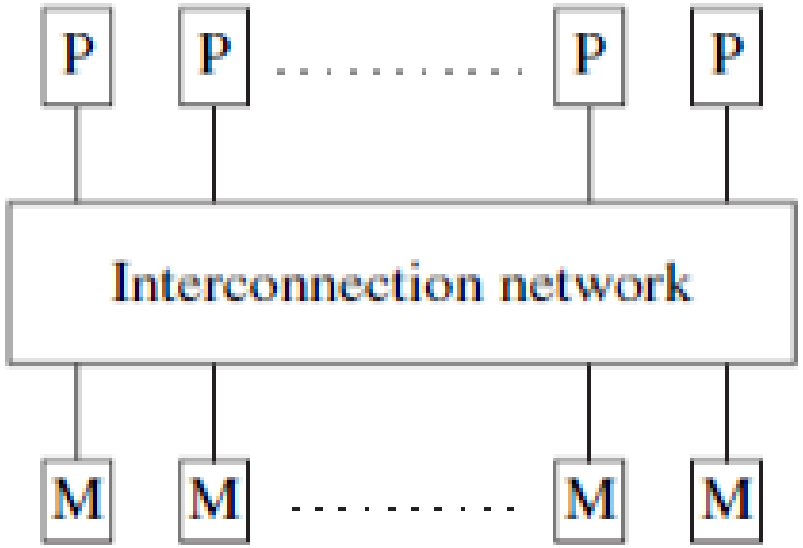
2. Multicomputer parallel systems

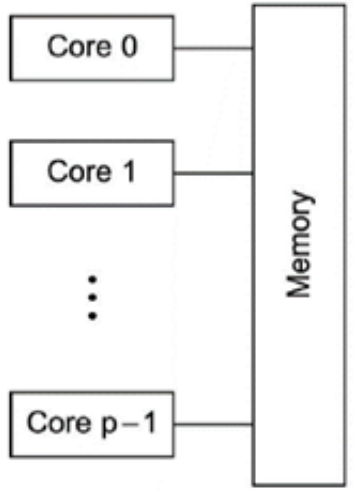
3. Array Processor

Multiprocessor system

Shared memory System

- Multiple processors have **direct access to shared memory** which forms a **common address space**.
- Do not have a common clock.
- Usually corresponds to a **uniform memory access** (UMA) architecture
  - **Same Access latency** to any memory location from any processor.
- All the processors usually run the **same operating system**
- Both the hardware and software are **very tightly coupled**.
- The processors are usually of the same type, and are housed within the same box/container with a shared memory.
- The interconnection network to access the memory may be a bus or a multistage switch for greater efficiency





Multicomputer parallel system
Distributed Memory System

Multiple processors **do not have direct access to shared memory**.

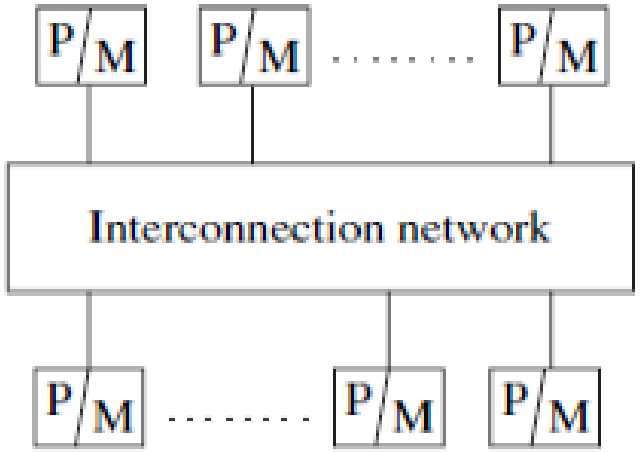
May or **may not form a common address space** in the memory

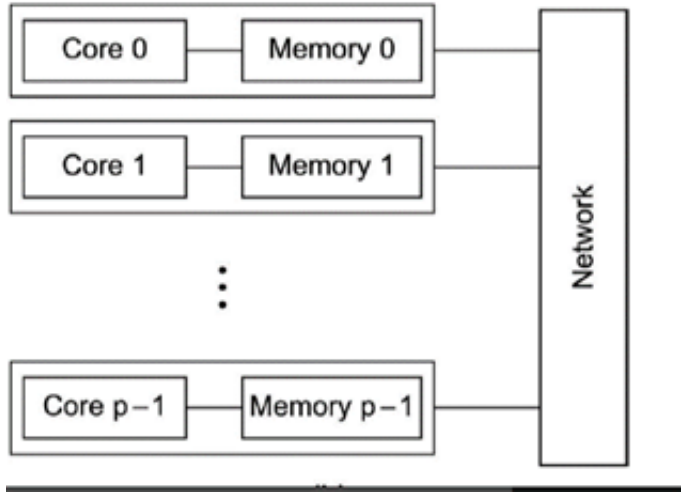
Do not have a common clock.

The processors are in close physical proximity and are usually very tightly coupled (homogenous hardware and software), and connected by an interconnection network.

The processors communicate either via common address space or via message-passing

A multicomputer system that has a common address space usually corresponds to a **non-uniform memory access (NUMA) architecture** in which the latency to access various shared memory locations from the different processors varies.





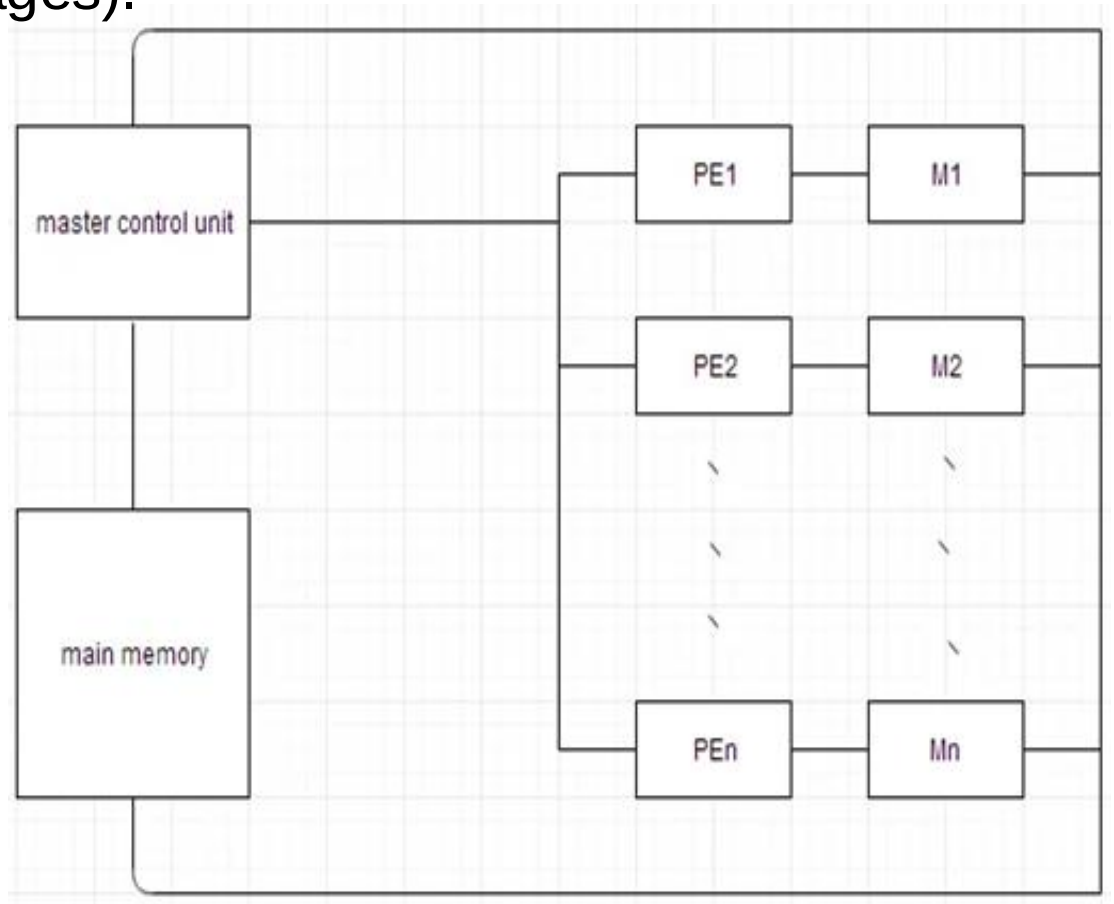
Array Processors

Belong to a class of parallel computers that are physically co-located, are **very tightly coupled**, and have a **common system clock** (but may not share memory and communicate by passing data using messages).

Perform tightly synchronized processing and data exchange in lock-step for applications such as DSP and image processing

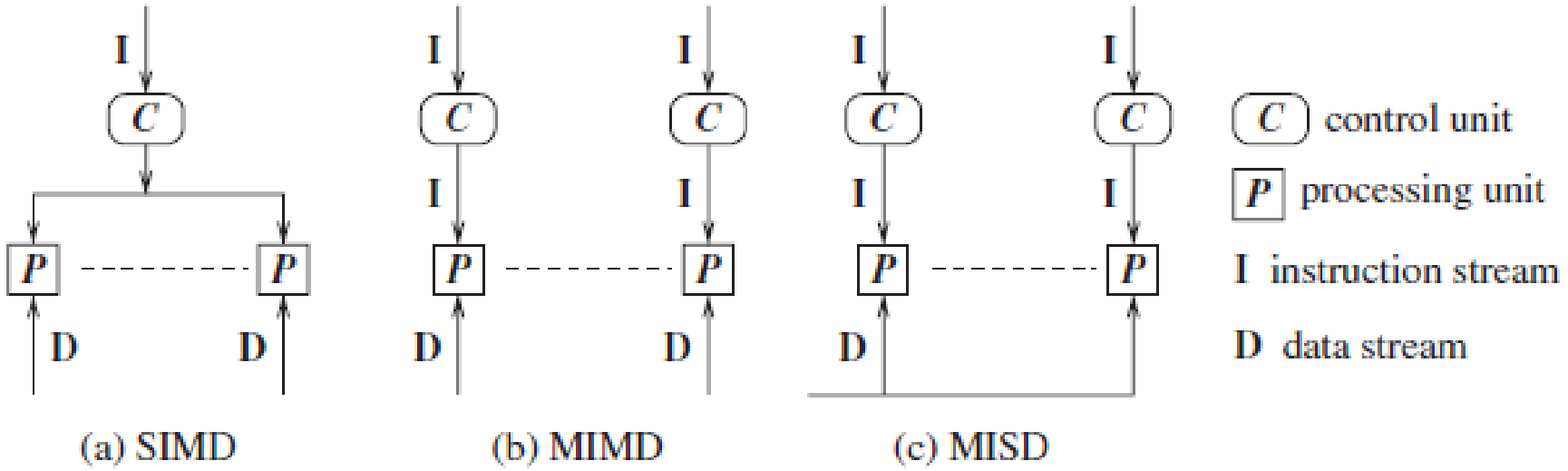
Applications usually involve a **large number of iterations on the data.**

Array processors, NUMA and message-passing multicomputer systems are less suitable when the degree of granularity of accessing shared data and communication is very fine.



Flynn’s taxonomy

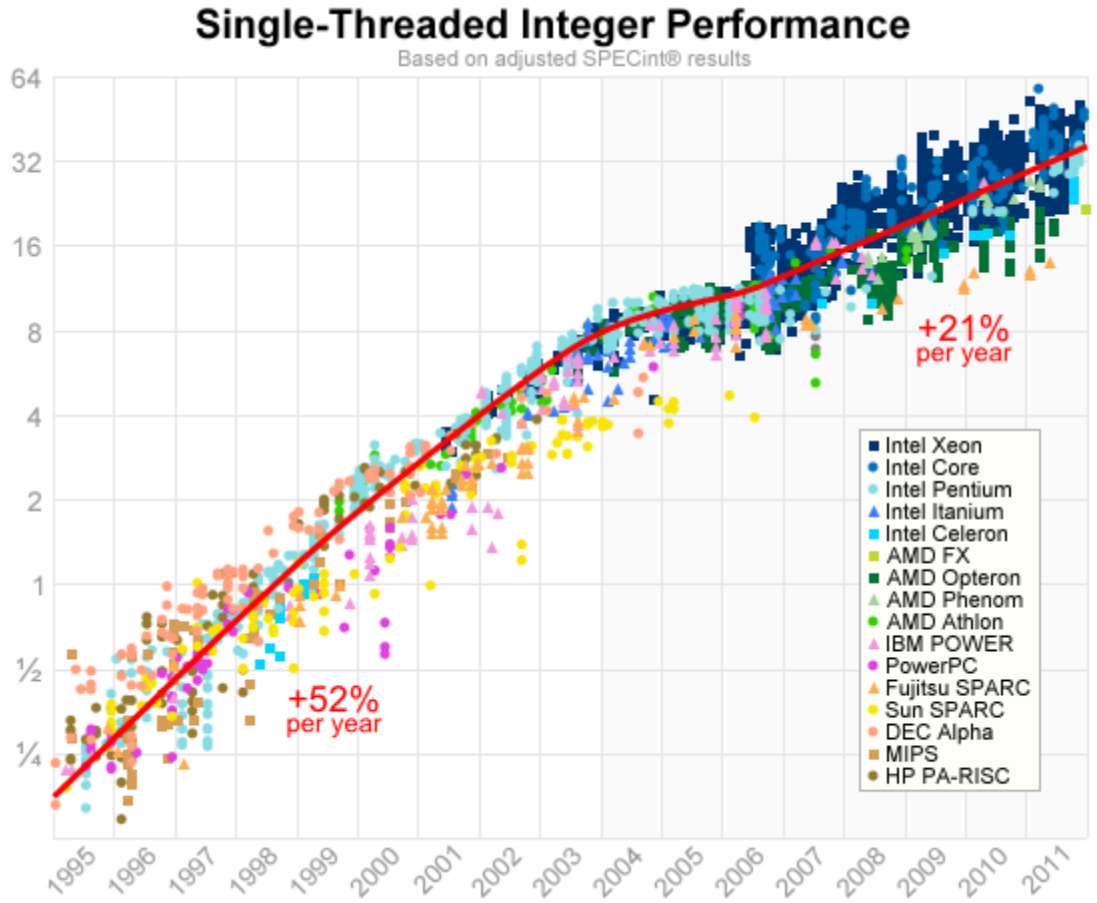
- Single instruction stream, single data stream (SISD)
- Single instruction stream, multiple data stream (SIMD)
- Multiple instruction stream, single data stream (MISD)
- Multiple instruction stream, multiple data stream (MIMD)



(a) SIMD (b) MIMD (c) MISD

Why parallel computing

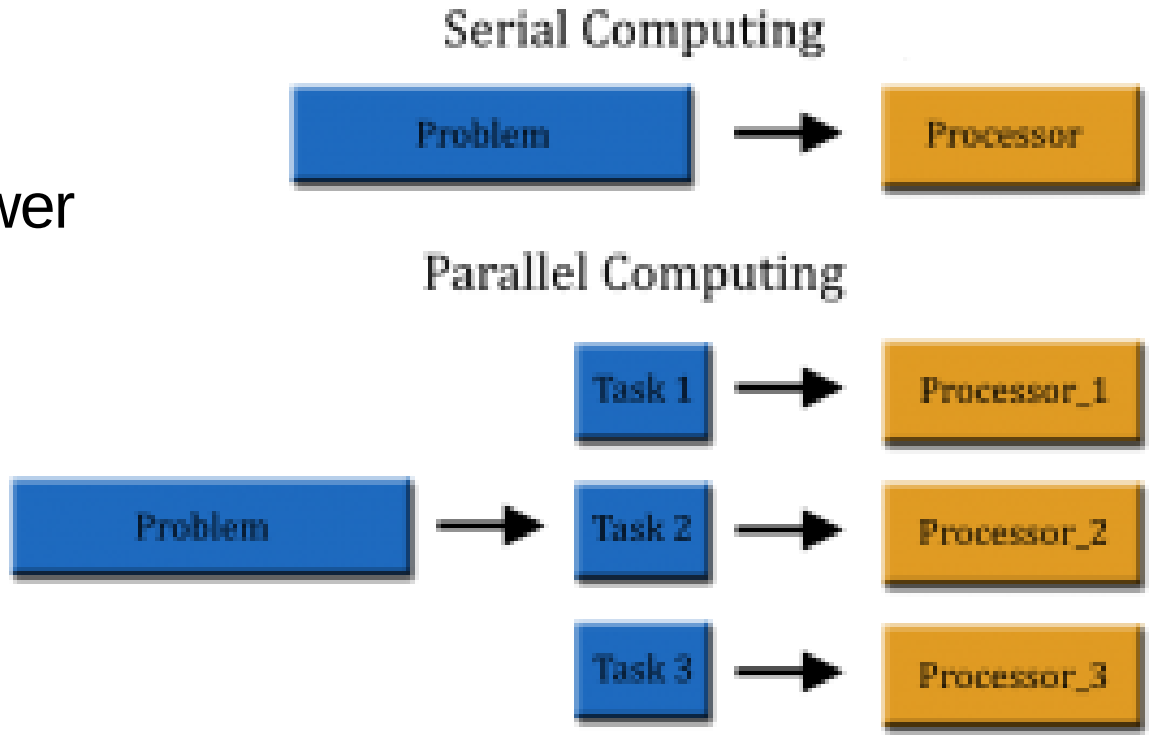
- From 1986 to 2003
  - The performance of microprocessors increased
  - on average, more than 50% per year
- Since 2003
  - single-processor performance improvement has slowed
- From 2015 to 2017
  - it increased at less than 4% per year



Why we need ever-increasing performance

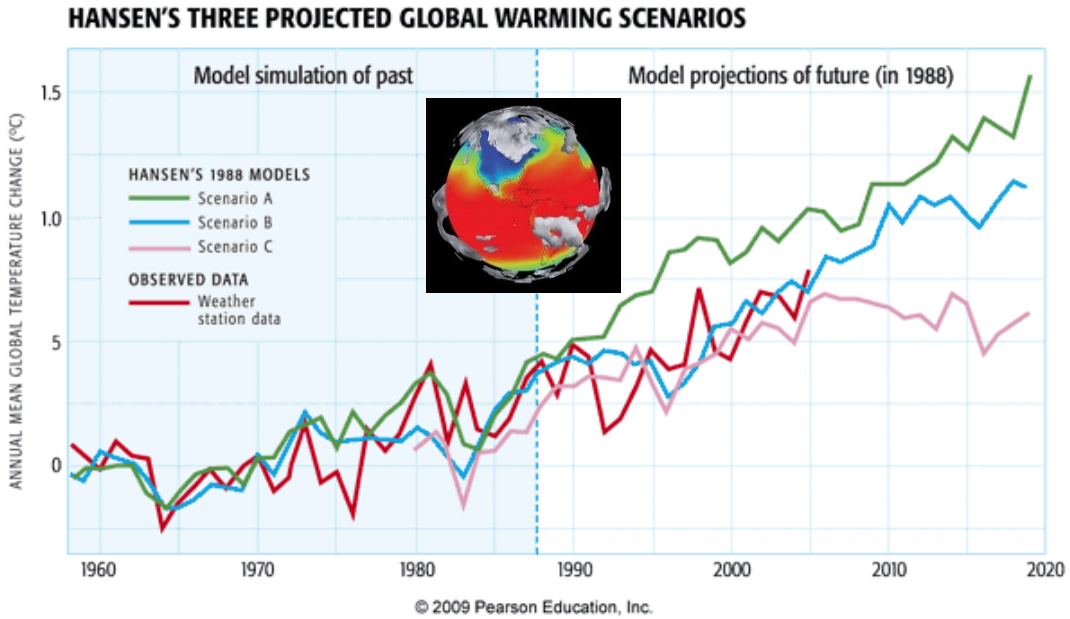
The vast increases in computational power

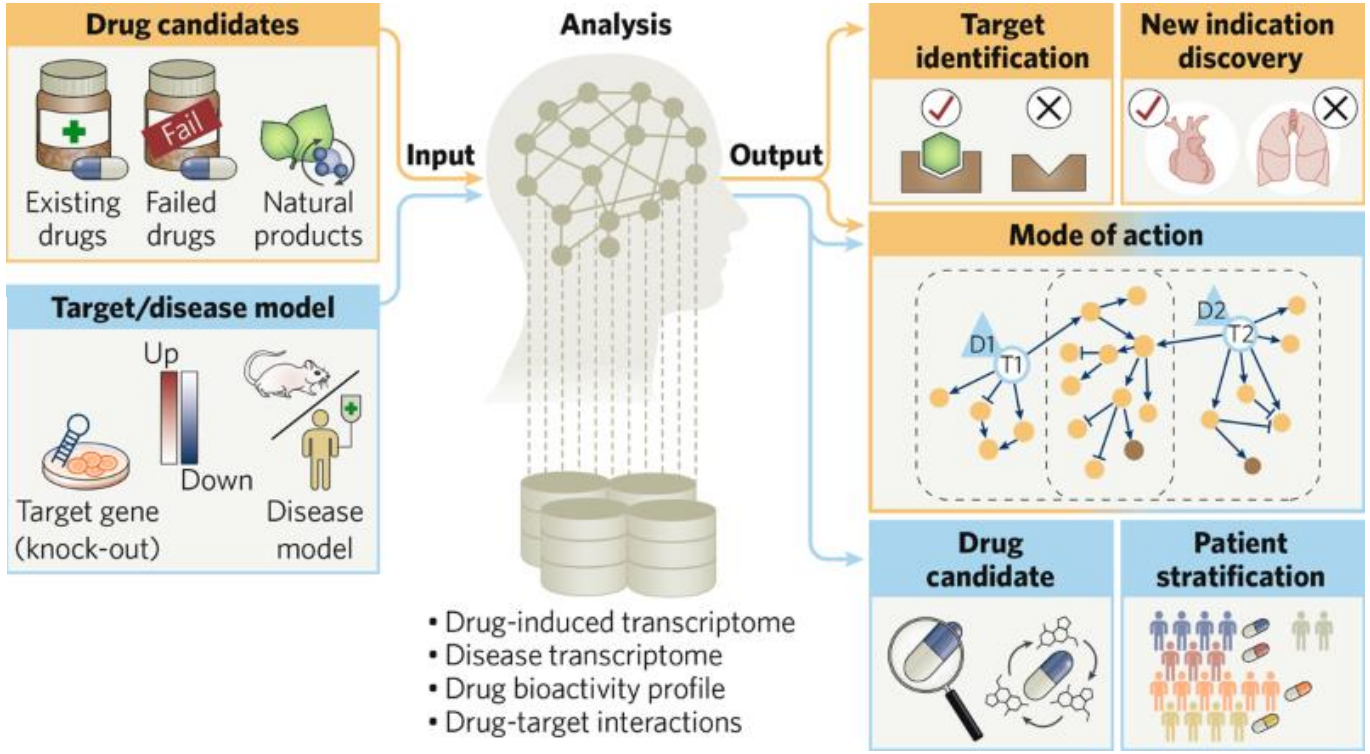
- Advances in diverse fields
-
  - Science
  - Internet
  - Entertainment.



Why we need ever-increasing performance

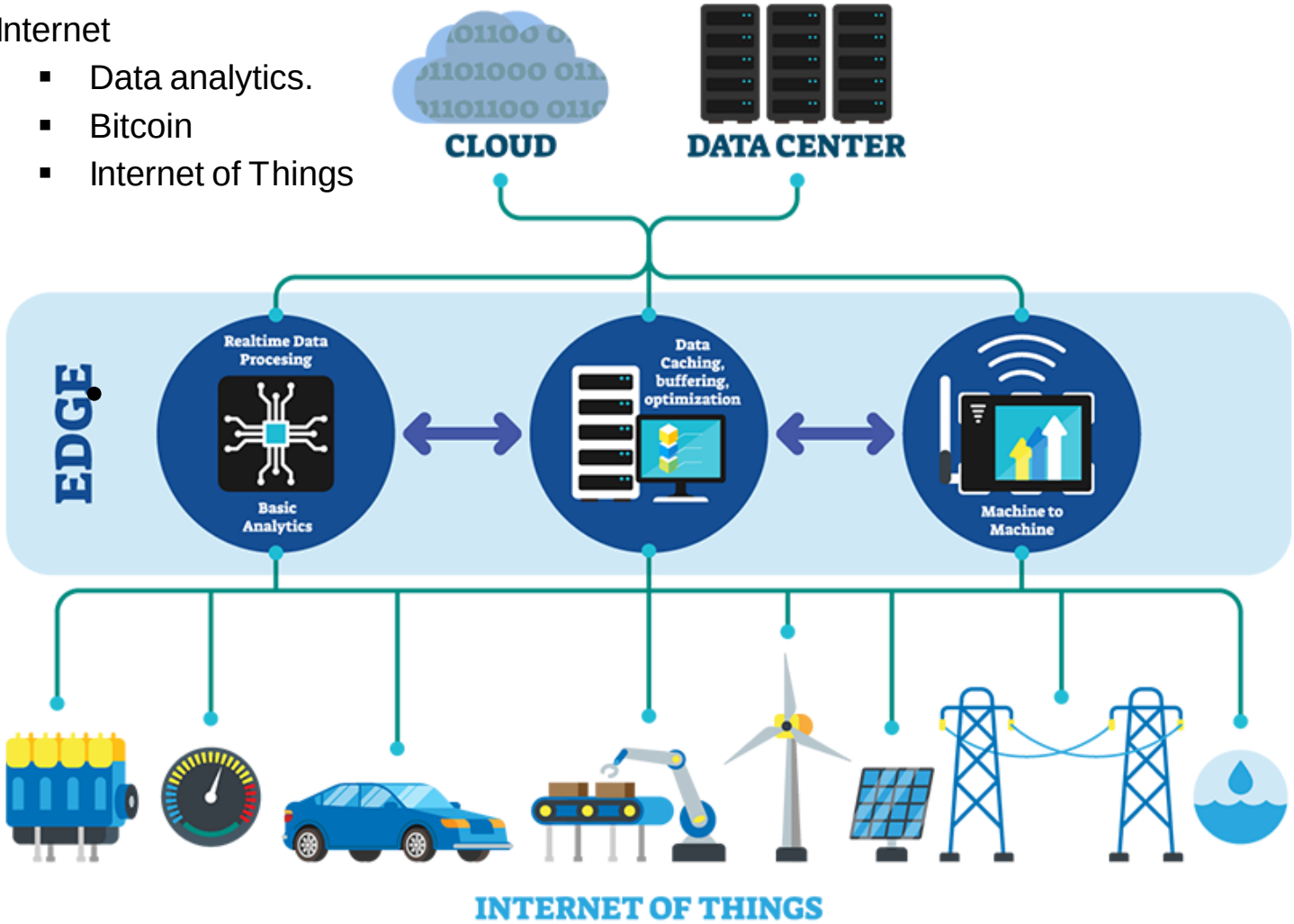
- Science
  - Climate modelling
  - Drug discovery

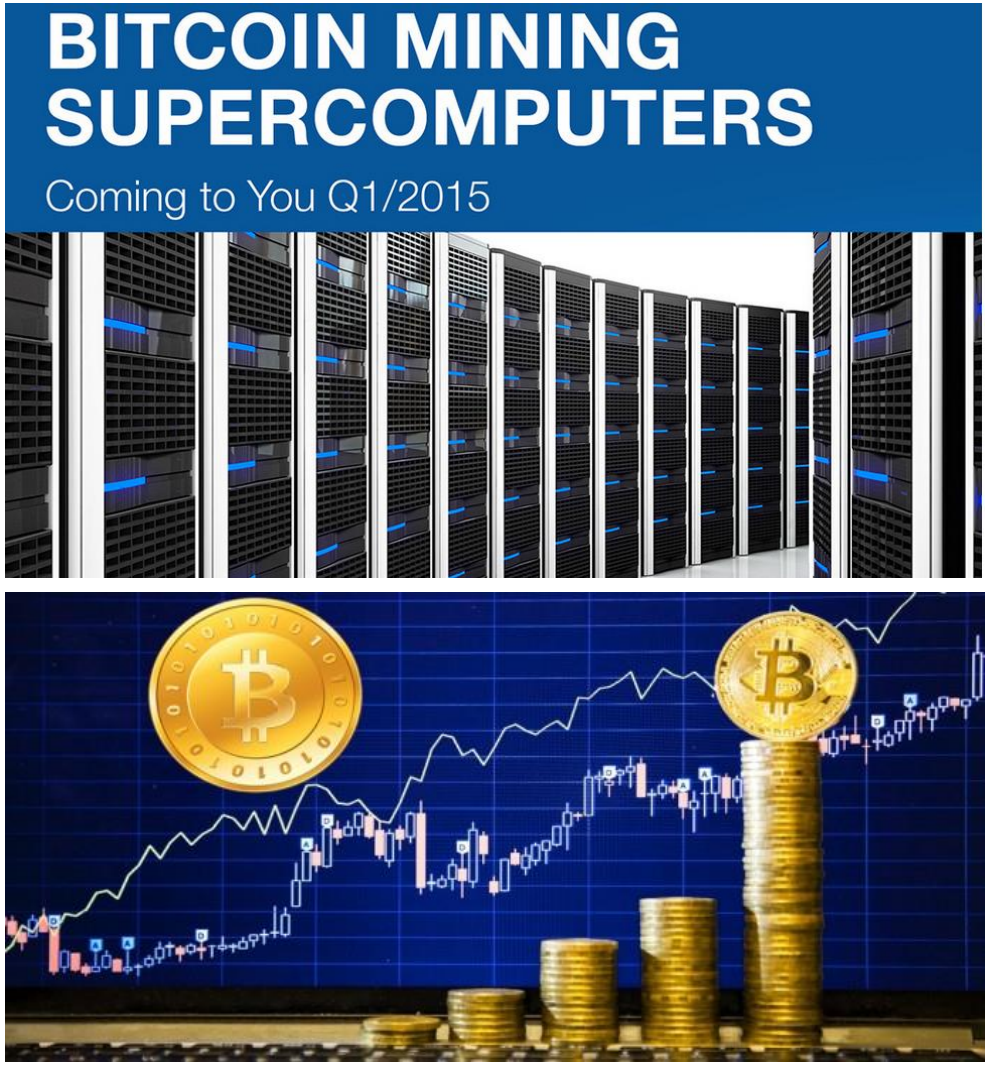




Why we need ever-increasing performance

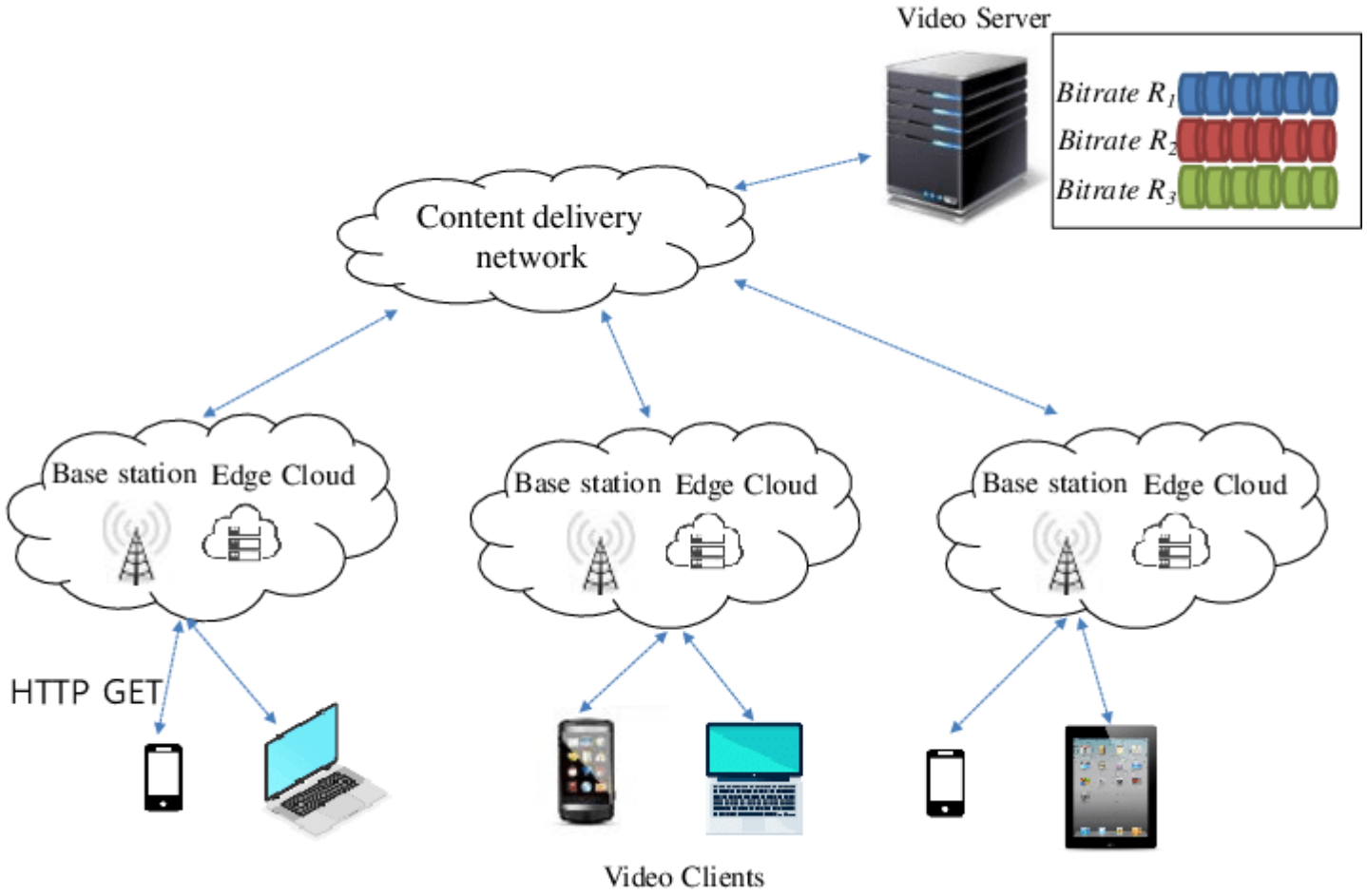
- Internet
  - Data analytics.
  - Bitcoin
  - Internet of Things





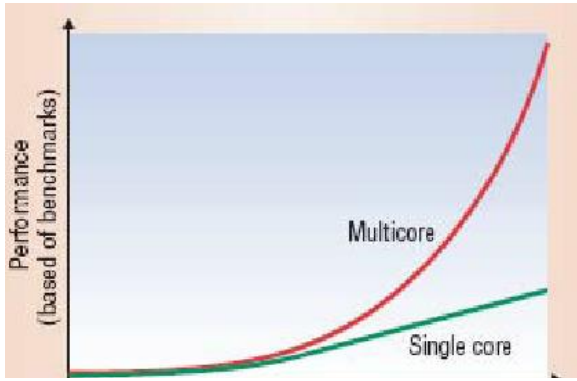
Why we need ever-increasing performance

- Entertainment.
  - Virtual reality
  - Video servers



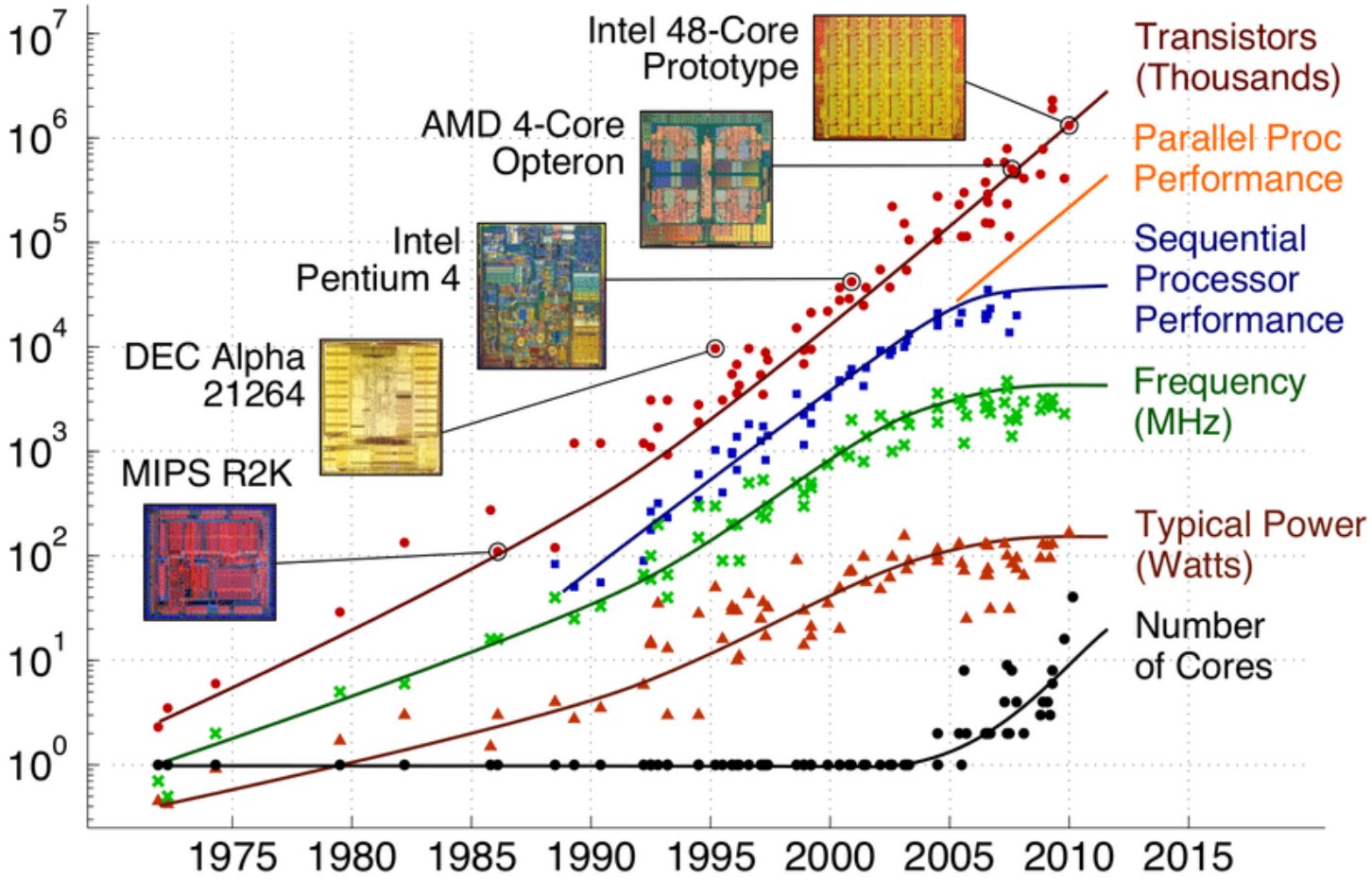
we need ever-increasing performance

Why we're building parallel systems



- Single-processor Integrated Circuit (IC) .
- The size of transistors decreased
- speed of transistors increased
- overall speed of the IC is increased

- Speed of transistors increased
- power consumption also increases
- Most of this power is dissipated as heat
- IC gets too hot, it becomes unreliable
- Impossible to continue to increase the speed of ICs

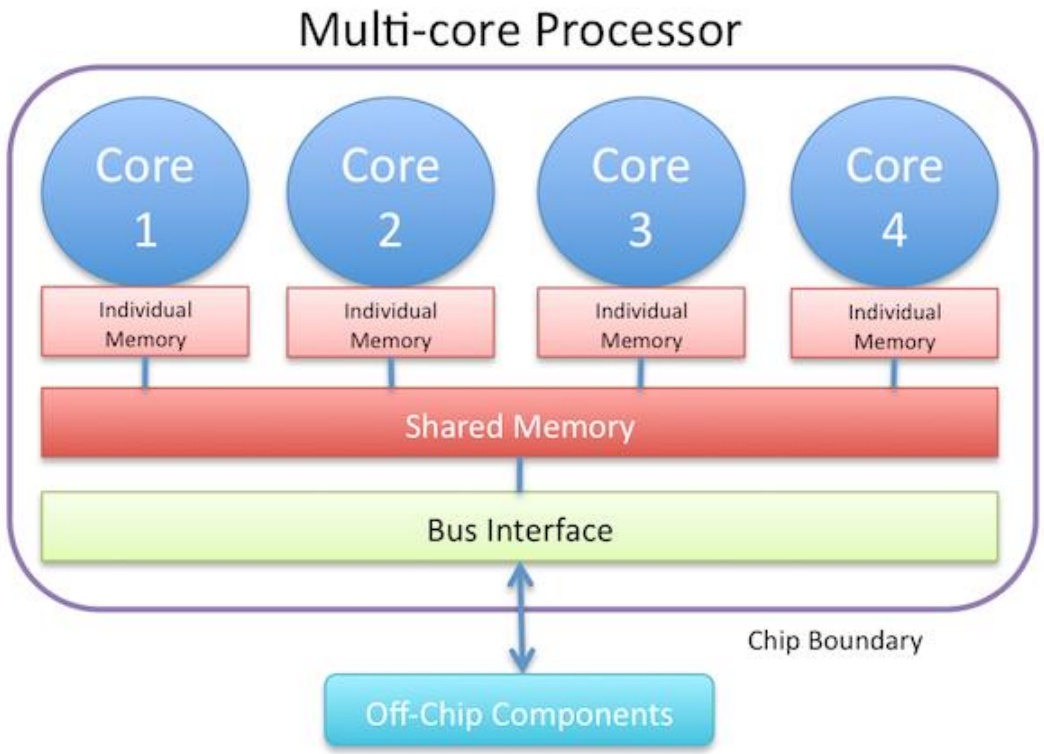


How then, can we continue to build ever more powerful computers?

The answer is parallelism.

- Rather than building
  - ever-faster
  - more complex
  - monolithic processors

The industry has decided to put
- Multiple
- relatively simple
- complete processors
- on a single chip



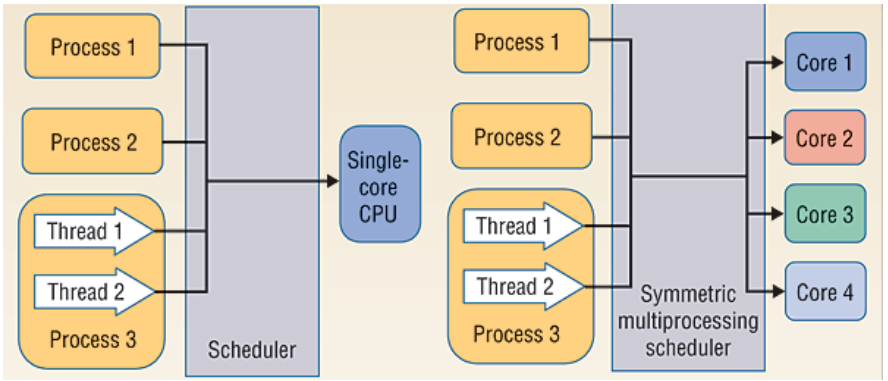
- Such ICs are called **multicore processors**
  - **core** has become synonymous with CPU
  - a conventional processor with one CPU is often called a **single-core** system

Why we need to write parallel programs

- Most programs that have been written for conventional, single-core systems
- **Cannot exploit the presence of multiple cores.**

We can run multiple instances of a program on a multicore system

- ***but this is often of little help***
- ***For example, being able to run multiple instances of our favorite game isn't really what we want***
- ***we want the program to run faster with more realistic graphics***



To do this

- we need to either rewrite our serial programs so that they're parallel
  - so that they can make use of multiple cores
- write translation programs, that is, programs that will automatically convert serial programs into parallel programs.
- The bad news is that researchers have had very limited success writing programs that convert serial programs in languages such as C, C++, and Java into parallel programs.

- While we can write programs that recognize common constructs in serial programs
- automatically translate these constructs into efficient parallel constructs,
- the sequence of parallel constructs may be terribly inefficient.

- **multiplication of two matrices as a sequence of dot products**
- but parallelizing a matrix multiplication as a sequence of parallel dot products is likely to be fairly slow on many systems.
- An efficient parallel implementation of a serial program may not be obtained by finding efficient parallelization of each of its steps
- Rather, the best parallelization may be obtained by devising an entirely new algorithm.

suppose that we need to compute n values 1,4,3, 9,2,8, 5,1,1, 6,2,7 and add them together

- serial code:

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

- we have p cores and $p \leq n$. $p = 4$ & $n = 12$
  - ➢ Then each core can form 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;
}
```

Core	0	1	2	3
my_sum	8	19	7	15

“Master” core Global sum

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

If number of cores is large

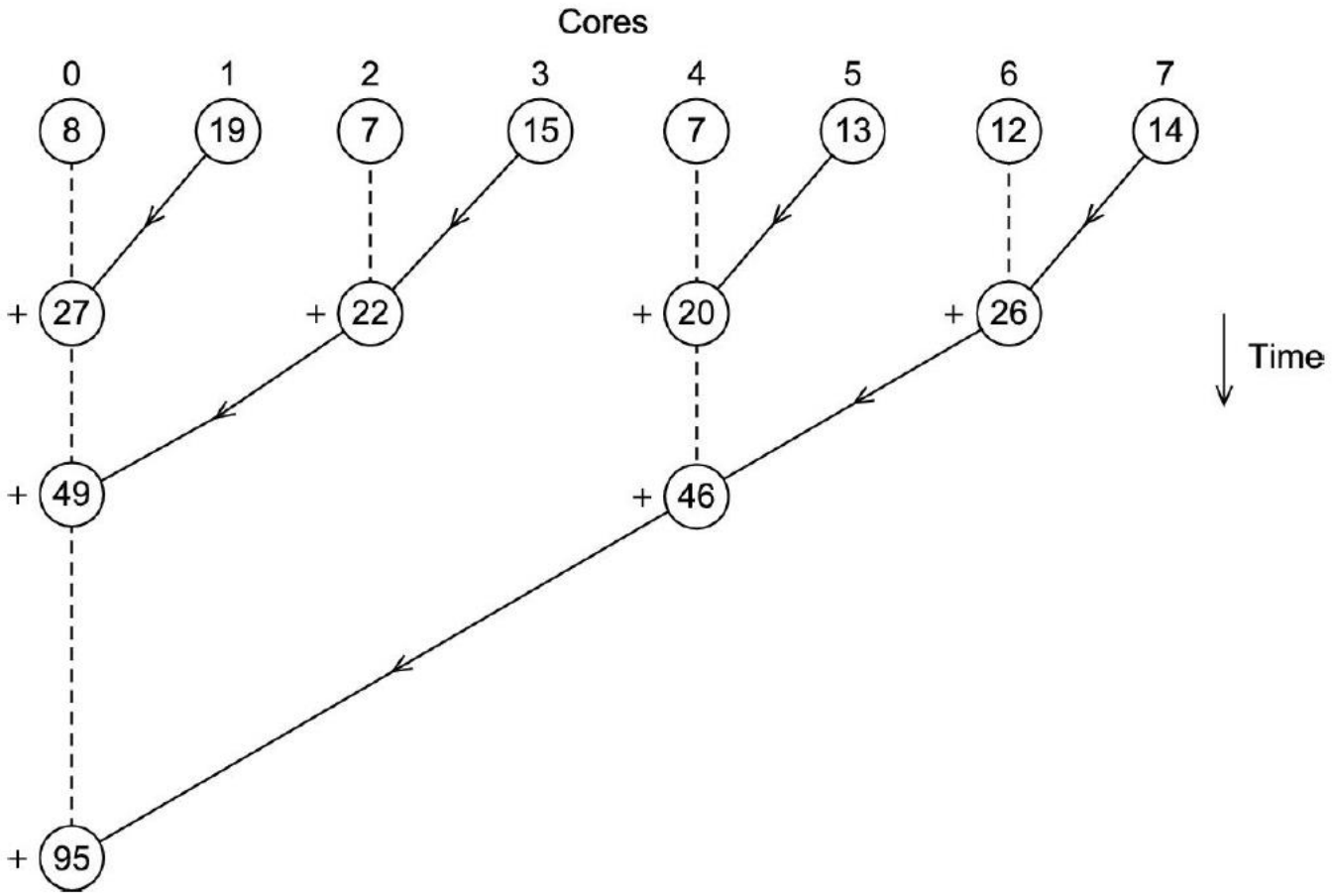


FIGURE 1.1 Multiple cores forming a global sum.

How do we write parallel programs?

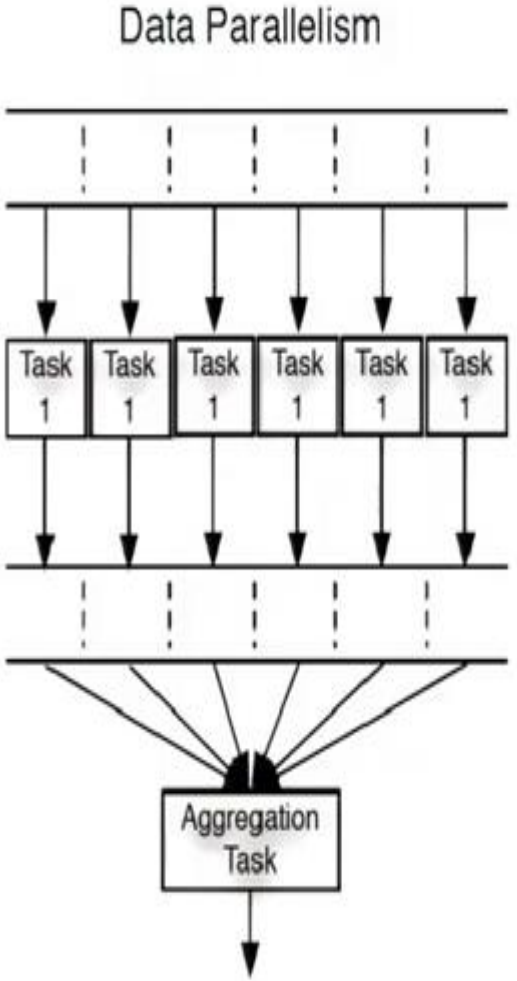
Data-parallelism

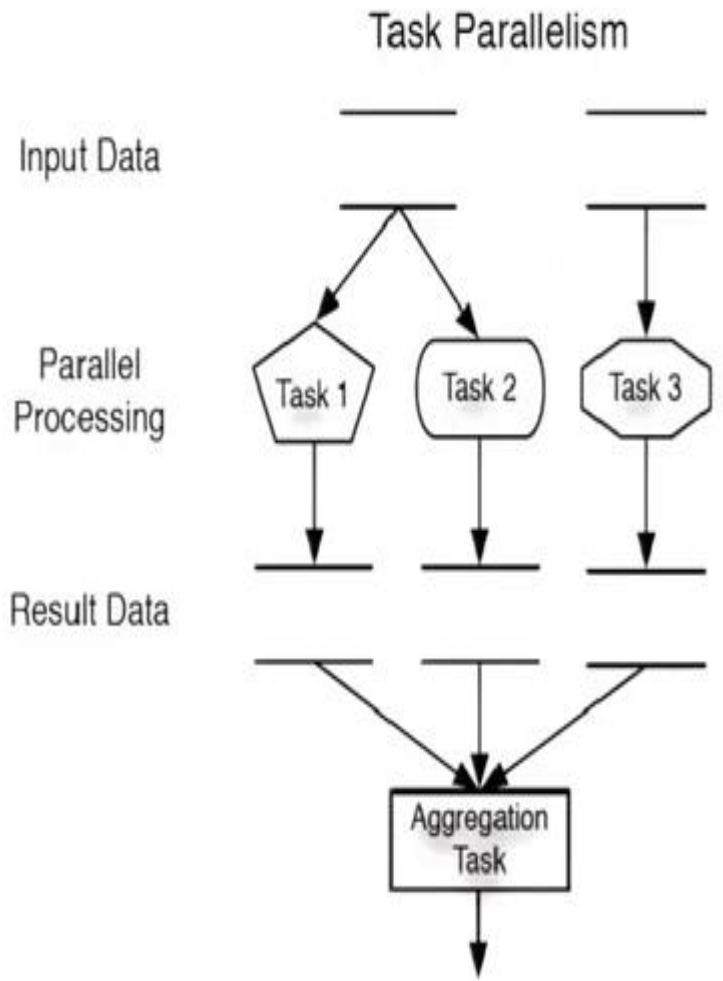
- partition the data used in solving the problem among the cores
- each core carries out more or less similar operations on its part of the data

Core	0	1	2	3
my_sum	8	19	7	15

Task-parallelism

- partition the various tasks carried out in solving the problem among the cores





Coordination among the cores

- **Communication**

➢ one or more cores send their current partial sums to another core.

- **Load balancing**.

➢ Load should be distributed equally

  - Suppose, one core has to compute most of the values
  - then the other cores will finish much sooner than the heavily loaded core
  - and their computational power will be wasted.

- **Synchronization**

```
Synchronize_cores();
```

➢ suppose that instead of computing the values to be added

  - the values are read from an array that is read in by the master core
  - the cores need to wait before starting execution of the code, master core should initialize

```
if (I'm the master core)
    for (my_i = 0; my_i < n; my_i++)
        scanf("%lf", &x[my_i]);
```

Limits of Parallelism and Scaling

Amdahl's Law and Strong Scaling

Amdahl's law provides a theoretical limit to how much faster a program can run if it is parallelized

Amdahl's law provides a way to quantify the theoretical maximum **speedup in latency** (also called the **speedup factor**) that can occur with parallel execution.

Specifically, Amdahl's law describes the ratio of the original execution time with the improved execution time, assuming perfect parallelism and no overhead penalty.

P denotes the percent of a program that can be executed in parallel
N denotes the number of parallel execution units

Amdahl's law states that the theoretical maximum speedup would be:

$$S = \frac{1}{(1-p) + \frac{P}{N}}$$

$$S = \frac{T_{orig}}{T_{parallel}} = \frac{T_{orig}}{(1-p)T_{orig} + \frac{p}{N}T_{orig}} = \frac{T_{orig}}{((1-p) + \frac{p}{N})T_{orig}}$$

Gustafson's Law and Weak Scaling

In perfectly strong scaling, there is no overhead penalty for creating more threads or using more processors. A program run on a system with 100 processors will run in 1/100th of the time than it would on a single-processor system. In contrast, a more realistic and common property is **weak scaling**, which emphasizes accomplishing more work rather than running in less time. In weak scaling, the additional processors are used to tackle bigger and more complex problems, while holding the expected run time to be the same.

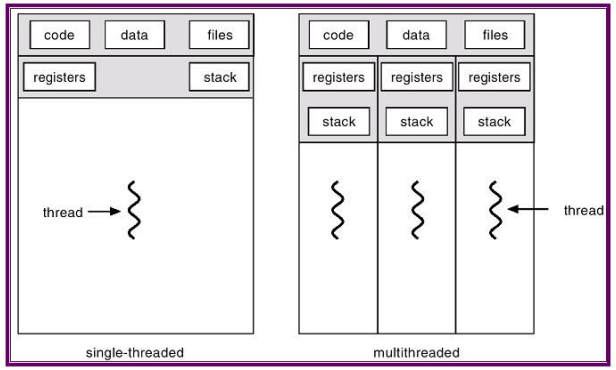
The amount of the improvement is denoted as s.

$$S = 1 - p + s * p$$

$$S = 0.2 + 5 * 0.8 = 0.2 + 4.0 = 4.2$$

learning to write programs that are explicitly parallel

- **C language** and four different **APIs** :
  - MPI — **Message-Passing Interface**
  - **POSIX threads** or **Pthreads**
  - OpenMP

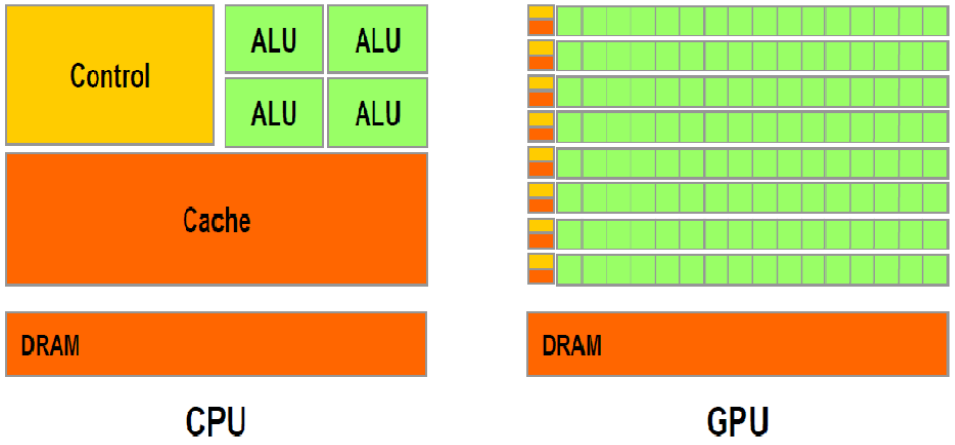


  - NVIDIA CUDA — **Compute Unified Device Architecture**

GPGPU **CPU+GPU**

The need to learn four APIs

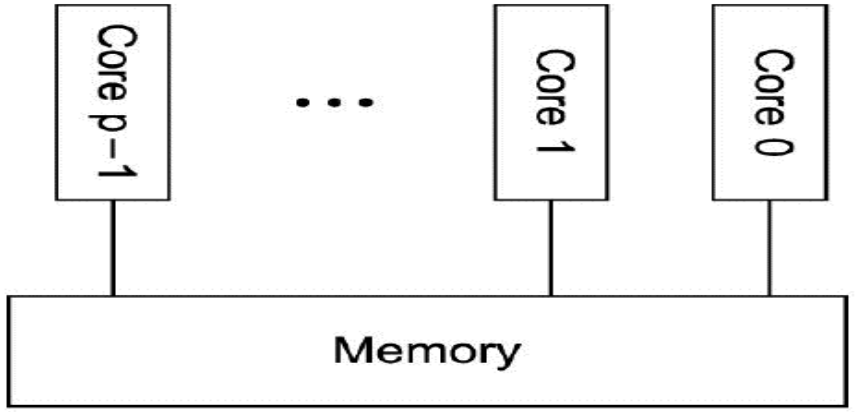
- MPI and Pthreads are libraries of type definitions, functions, and macros that can be used in C programs.
- OpenMP consists of a library and some modifications to the C compiler.
- CUDA consists of a library and modifications to the C++ compiler.
- MPI is an API for programming distributed memory MIMD systems
- Pthreads is an API for programming shared memory MIMD systems.
- OpenMP is an API for programming both shared memory MIMD and shared memory SIMD systems
- CUDA is an API for programming Nvidia GPUs



Shared memory and Distributed memory

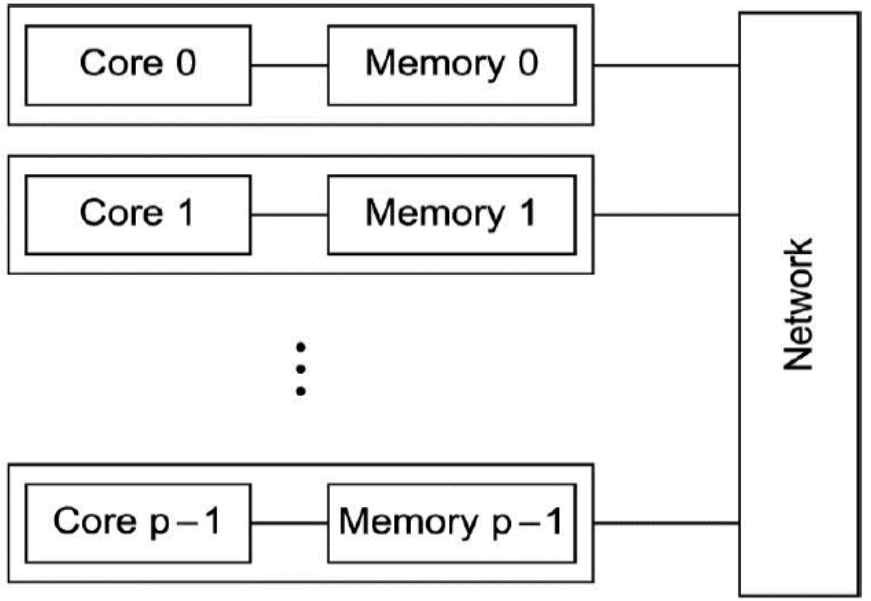
Shared-memory systems

- The cores can share access to the computer's memory
- in principle, each core can read and write each memory locatio
- we can coordinate the cores by having them examine and upda shared-memory locations



Distributed-memory systems

- each core has its own, private memory
- the cores can communicate explicitly by sending messages across a network



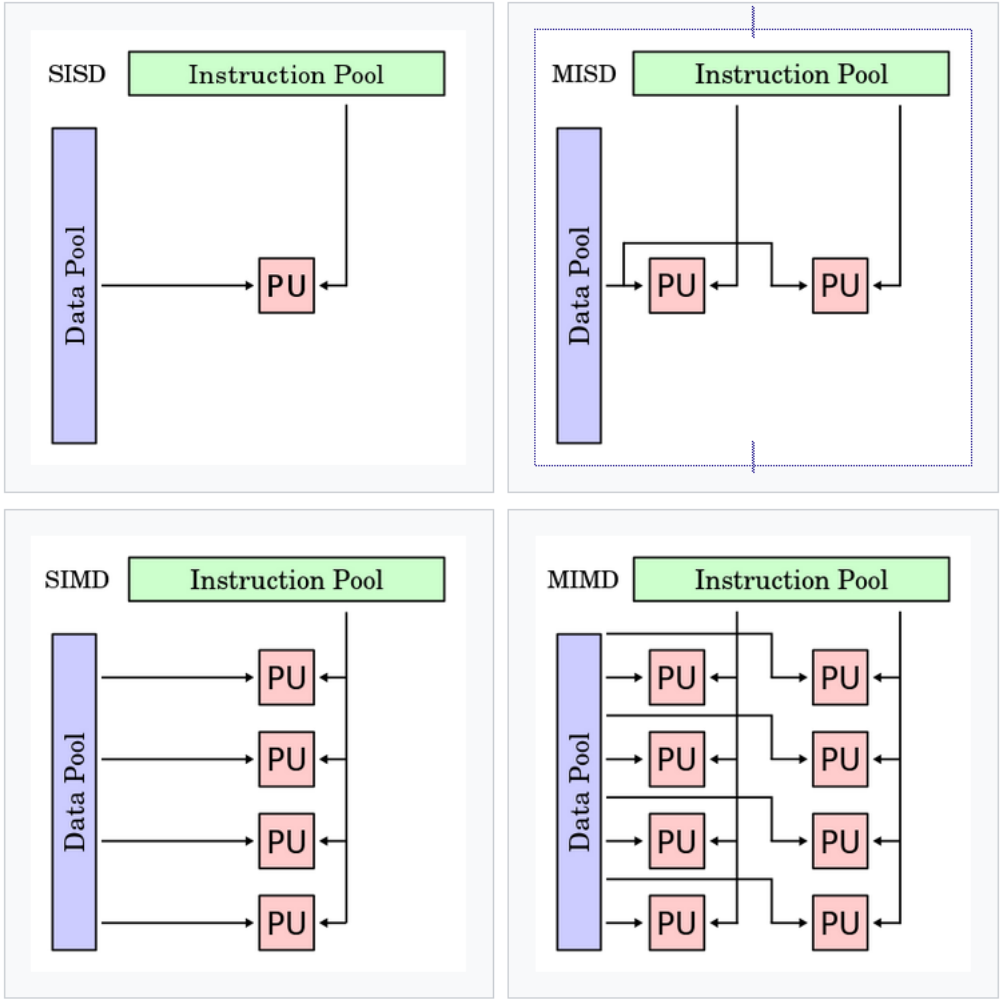
SIMD vs MIMD

Single-Instruction Multiple-Data or SIMD system

- Each core can access either its own private memory or memory that's shared among the cores
- In this type of system, all the cores carry out the same instruction on their own data

Multiple-Instruction Multiple-Data or MIMD system

- Each core can manage its own instruction stream and its own data stream
- The cores can be thought of as conventional processors, so they have their own control units, and they are capable of operating independently of each other



Concurrent, parallel, distributed computing

concurrent computing

- a program is one in which multiple tasks can be in progress at any instant

parallel computing

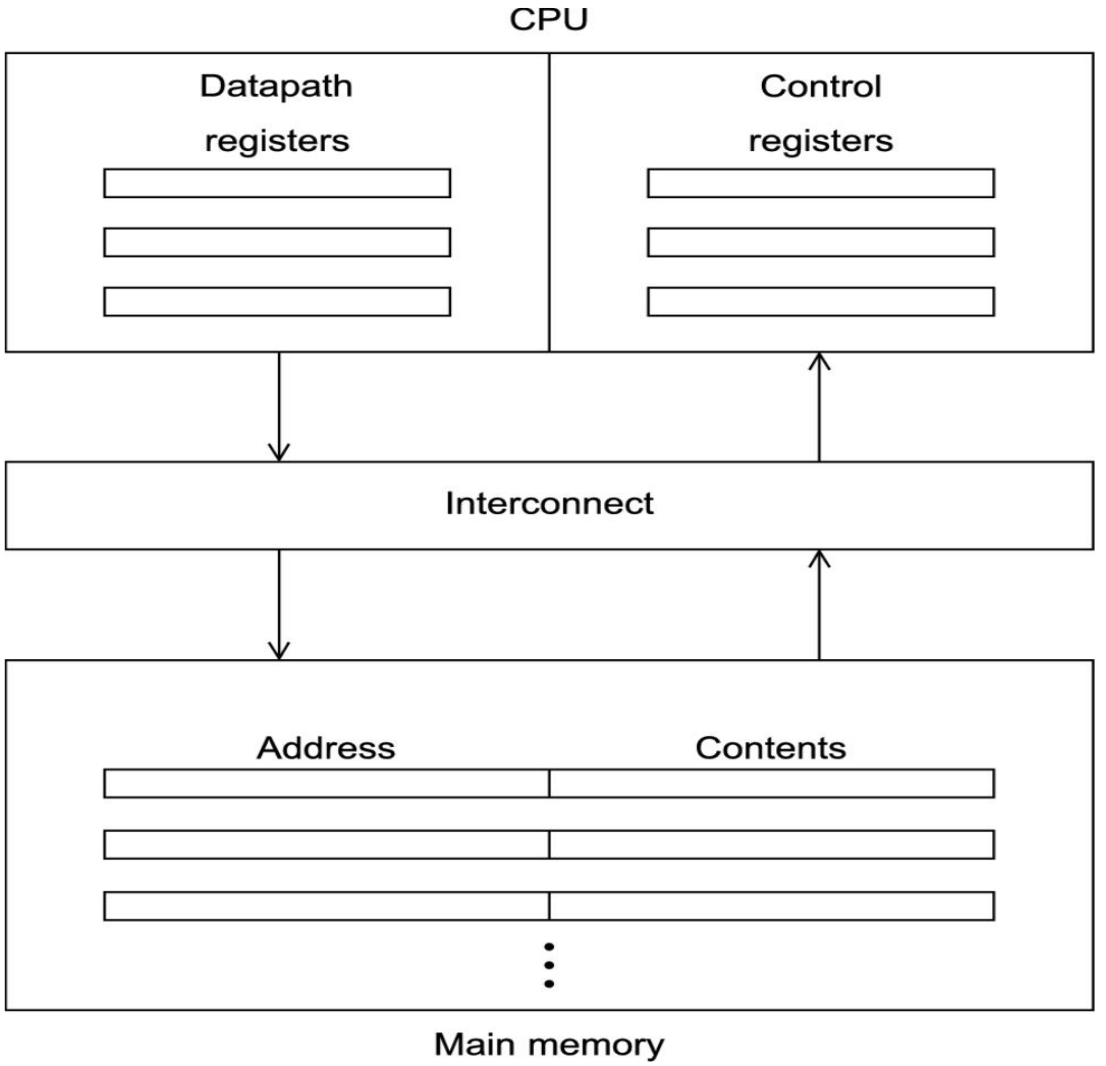
- a program is one in which multiple tasks cooperate closely to solve a problem

distributed computing

- a program may need to cooperate with other programs to solve a problem

The von Neumann architecture

- The separation of memory and CPU is often called the **Von Neumann bottleneck**, since the interconnect determines the rate at which instructions and data can be accessed.



Operating System, Processes, multitasking, and threads

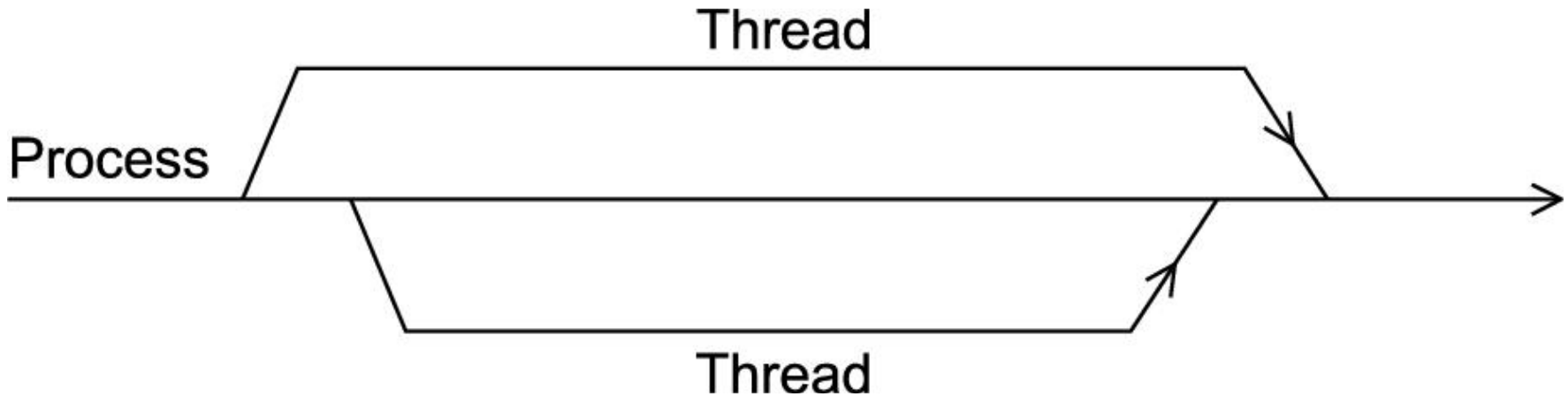
- The **operating system**, or OS, is a major piece of software
  - whose purpose is to manage hardware and software resources on a computer.
  - It determines which programs can run and when they can run.
  - It also controls the allocation of memory to running programs and access to peripheral devices, such as hard disks and network interface cards.
- When a user runs a program, the operating system creates a **process**
  - A process consists of several entities:
    - The executable machine language program.
    - A block of memory, which will include the executable code
    - a **call stack** that keeps track of active functions
- a **heap** that can be used for memory explicitly allocated by the user program, and some other memory locations.
- Descriptors of resources that the operating system has allocated to the process, for example, file descriptors.
  - Security information—for example, information specifying which hardware and software resources the process can access.
  - Information about the state of the process, such as
    - whether the process is ready to run or is waiting on some resource, the content of the registers, and information about the process's memory.

Multitasking, Threading

- Most modern operating systems are **multitasking**
  - simultaneous execution of multiple programs
  - even on a system with a single core
    - each process runs for a small interval of time **- time slice**
- In a multitasking OS, if a process needs to wait for a resource
  - It will **block**. It will stop executing and the OS can run another process.
  - However, many programs can continue to do useful work even though the part of the program that is currently executing must wait on a resource.
  - An airline reservation system that is blocked waiting for a seat map for one user could provide a list of available flights to another user.
- **Threading** provides a mechanism for programmers to divide their programs into more or less independent tasks, with the property that when one thread is blocked, another thread can be run.

Threads are “lighter weight” than processes

- Switching between threads is much faster than switching between processes
- If a process is the “master” thread of execution and threads are started and stopped by the process
  - when a thread is started, it **forks** off the process; when a thread terminates, it **joins** the process.



Improvements to the von Neumann model

- Caching, Virtual memory, and Low-level parallelism
  - Cache memory is a special memory that is effectively closer to the registers in the CPU
  - Multilevel caches - don't duplicate information
    - L1 miss and an L2 hit
- **Virtual memory**
  - main memory can function as a cache for secondary storage
  - **page table** is used to translate the virtual address into a physical address
  - the page table doesn't have a valid physical address for the page and the page is only stored on disk
    - then the attempted access is called a **page fault**

Instruction-level parallelism

- **Pipelining**
  - functional units are arranged in stages
- **Multiple issue**
  - multiple instructions can be simultaneously initiated
    - functional units are scheduled at compile time
      - multiple issue system is said to use **static** multiple issue
    - scheduled at run-time
      - **dynamic** multiple issue.
- A processor that supports dynamic multiple issue is **superscalar**

Hardware multithreading

- Attempts to provide parallelism through the simultaneous execution of different threads
  - a **coarser-grained** parallelism than ILP,—threads—are larger or coarser than the **finer-grained** units—individual instructions
- In **fine-grained** multithreading
  - the processor switches between threads after each instruction
  - skipping threads that are stalled
    - drawback that a thread that's ready to execute a long sequence of instructions may have to wait to execute every instruction
- **Coarse-grained** multithreading attempts to avoid this problem
  - by only switching threads that are stalled waiting for a time-consuming operation to complete (e.g., a load from main memory)
- **Simultaneous multithreading**
  - a variation on fine-grained multithreading. It attempts to exploit superscalar processors by allowing multiple threads to make use of the multiple functional units

Flynn's taxonomy - Classifications of parallel computers

- A classical von Neumann system
  - is a **single instruction stream, single data stream**, or SISD system
    - executes a single instruction at a time and computes a single data value at a time
- **Single instruction, multiple data**, or SIMD
  - systems are parallel systems
  - operate on multiple data streams
    - by applying the same instruction to multiple data items
- **Vector processors**
  - operate on arrays or vectors of data
    - while conventional CPUs operate on individual elements or scalars
- **Graphics processing units**
  - Typical GPUs can have dozens of cores, and these cores are also capable of executing independent instruction streams.
    - So GPUs are neither purely SIMD nor purely MIMD
    - GPUs are becoming increasingly popular for general, high-performance computing,
      - several languages have been developed that allow users to exploit their power.

Multiple instruction, multiple data, or MIMD, systems

- support multiple simultaneous instruction streams operating on multiple data streams.
  - MIMD systems typically consist of a collection of fully independent processing units or cores, each of which has its own control unit and its own datapath
- MIMD systems are usually **asynchronous**, that is, the processors can operate at their own pace

Distributed-memory systems

- The most widely available distributed-memory systems are called **clusters**.
  - A collection of commodity systems
    - the **nodes** of these systems, are usually shared-memory systems with one or more multicore processors
  - distinguish such systems from pure distributed-memory systems
  - they are sometimes called **hybrid systems**.

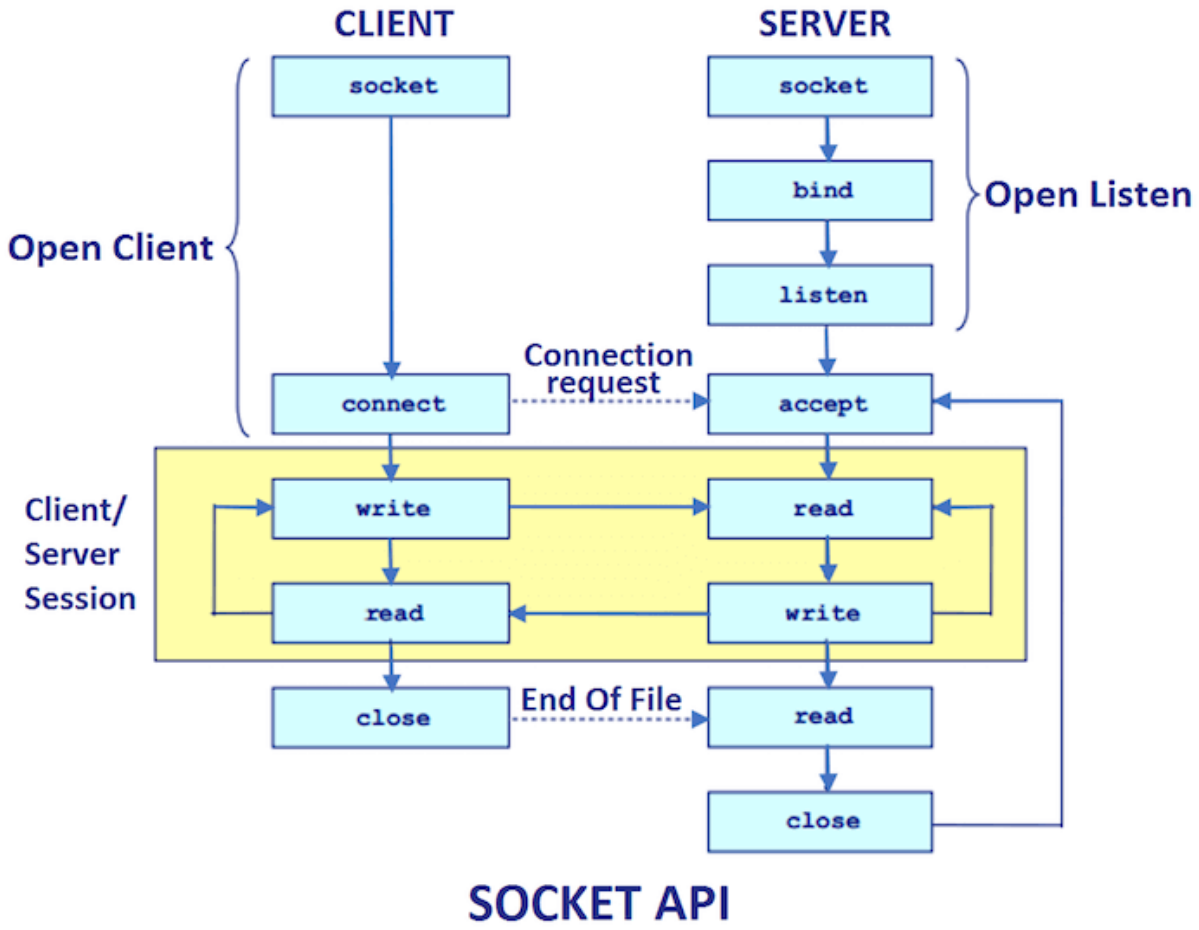
- The **grid** provides the infrastructure necessary to turn large networks of geographically distributed computers into a unified distributed-memory system.
  - In general, such a system is heterogeneous, that is, the individual nodes are built from different types of hardware.

Labs

- Lab 1: Heart-beat and membership
- Lab 2 : Parallel and distributed data processing
- Labs 3&4 :High performance using Multithreading
- Lab 5 : Scalability
- Lab 6: GPU programming
- Lab 7 : Event-driven Programming

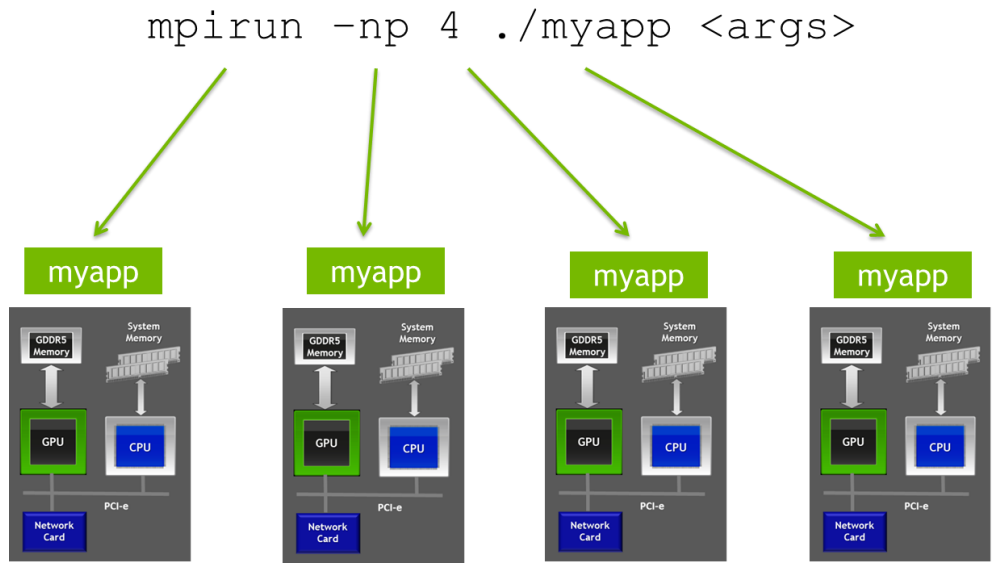
Socket Programming

- Client-server communication
- Discover node health
- Communication failures
- Secure socket layer, ssh-keygen
- In a cluster with dynamic membership, nodes can leave and rejoin or crash and rejoin



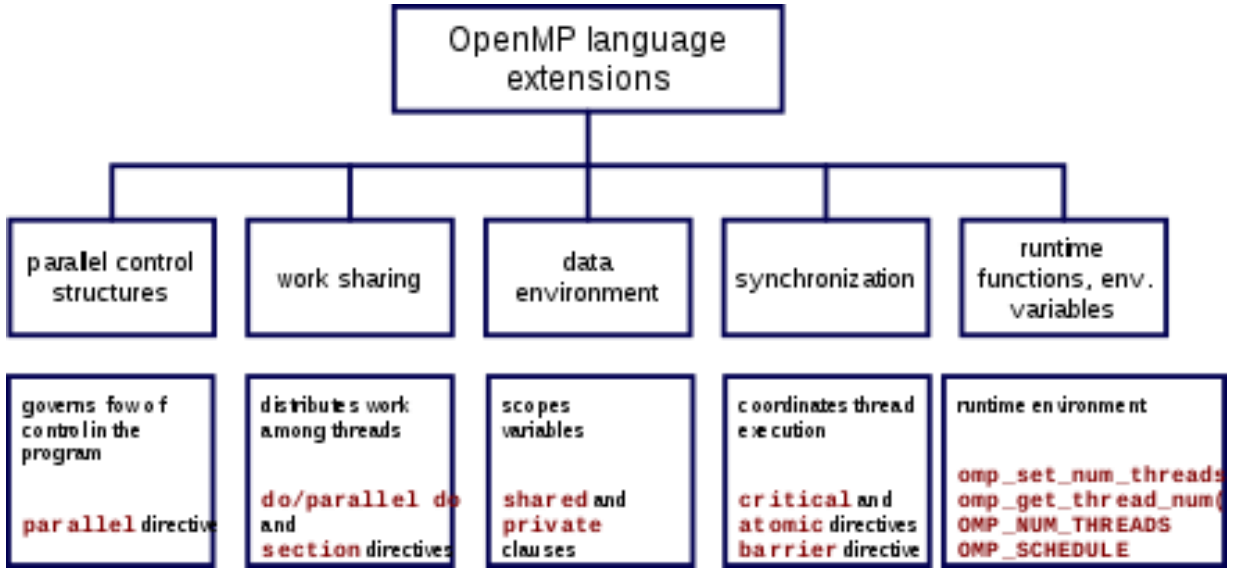
OpenMPI – Distributed Memory parallelism

```
#include <iostream>
#include <mpi.h>

int main(int argc, char** argv) {
  // Initialisation
MPI_Init(&argc,  &argv);
  // Reading size and rank
int size, rank;
MPI_Comm_size(MPI_COMM_WORLD,  &size);
MPI_Comm_rank(MPI_COMM_WORLD,  &rank);
  // Printing
  std::cout << "Hello  world, from process #" << rank<< std::endl;
    // Finalisation
MPI_Finalize();
return 0;
}
```



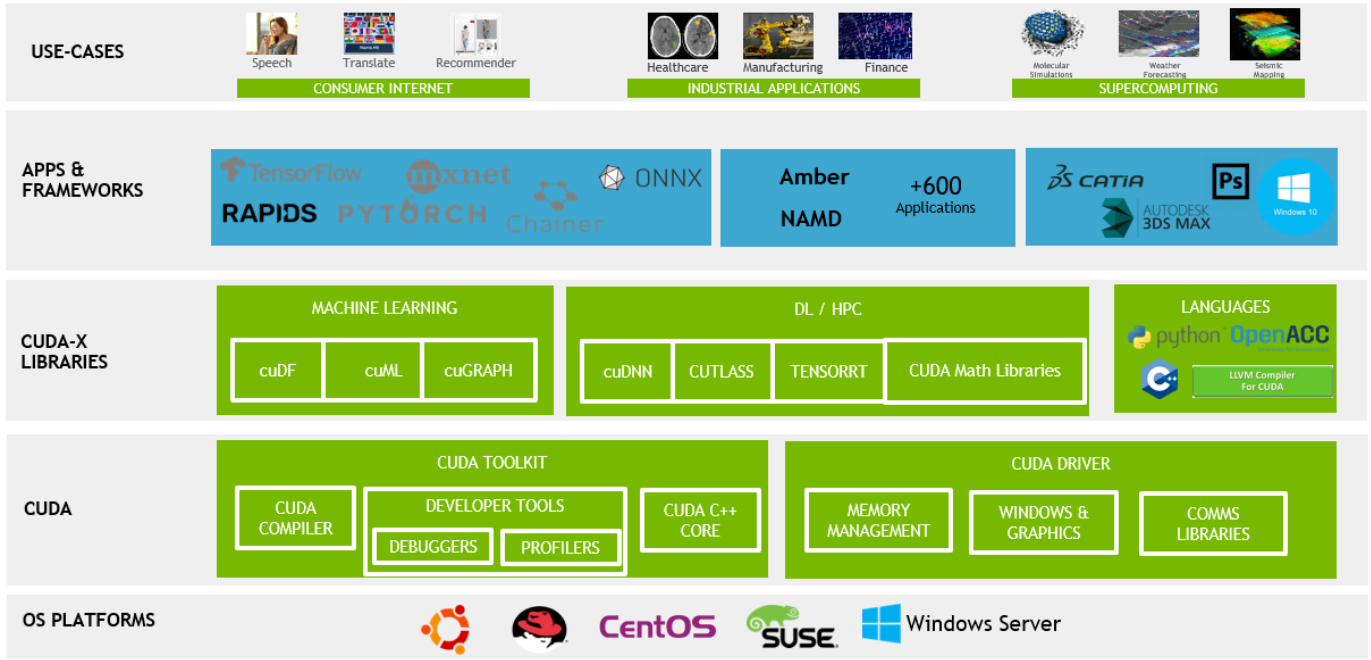
OpenMP – Shared Memory Parallelism

```
1       int x = 5;
2   #   pragma omp parallel num_threads(thread_count) \
3           private(x)
4       {
5           int my_rank = omp_get_thread_num();
6           printf("Thread %d > before initialization, x = %d\n",
7                  my_rank, x);
8           x = 2*my_rank + 2;
9           printf("Thread %d > after initialization, x = %d\n",
10                 my_rank, x);
11      }
12      printf("After parallel block, x = %d\n", x);
```



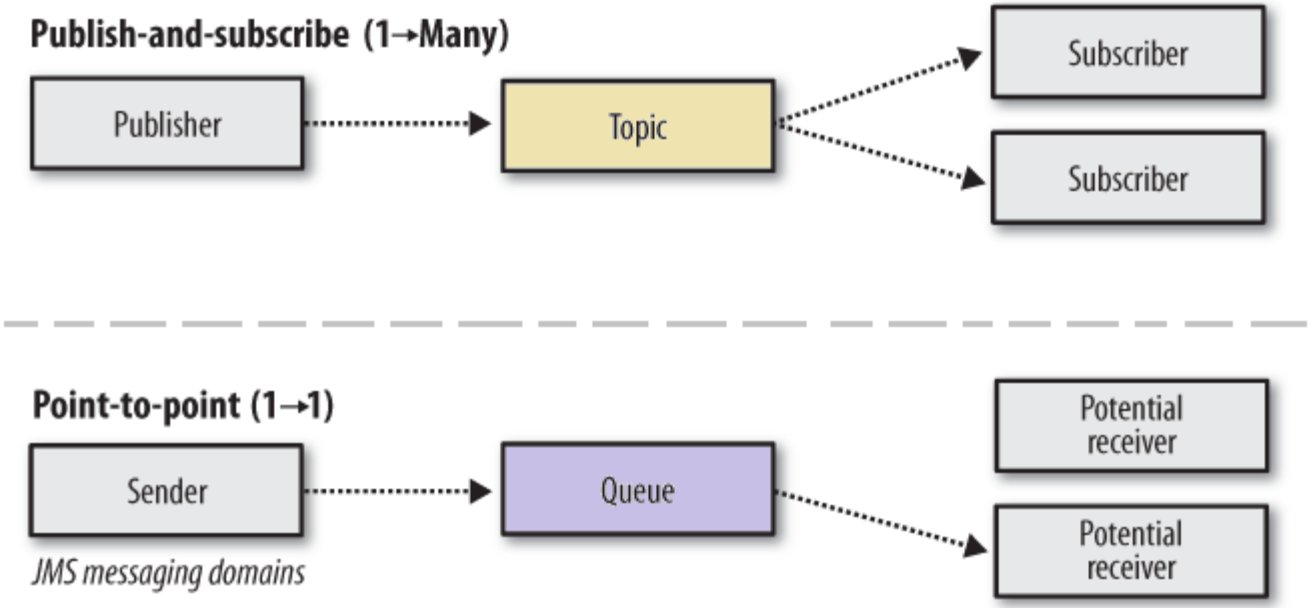
GPU & CUDA – Massive Parallelism

```c
#include <stdio.h>
#include <cuda.h>   /* Header file for CUDA */

/* Device code:  runs on GPU */
__global__ void Hello(void) {

    printf("Hello from thread %d!\n", threadIdx.x);
}  /* Hello */


/* Host code:  Runs on CPU */
int main(int argc, char* argv[]) {
    int thread_count;      /* Number of threads to run on GPU */

    thread_count = strtol(argv[1], NULL, 10);
                        /* Get thread_count from command line */

    Hello <<<1, thread_count>>>();
                        /* Start thread_count threads on GPU, */

    cudaDeviceSynchronize();        /* Wait for GPU to finish */

    return 0;
}  /* main */
```



Event driven programming

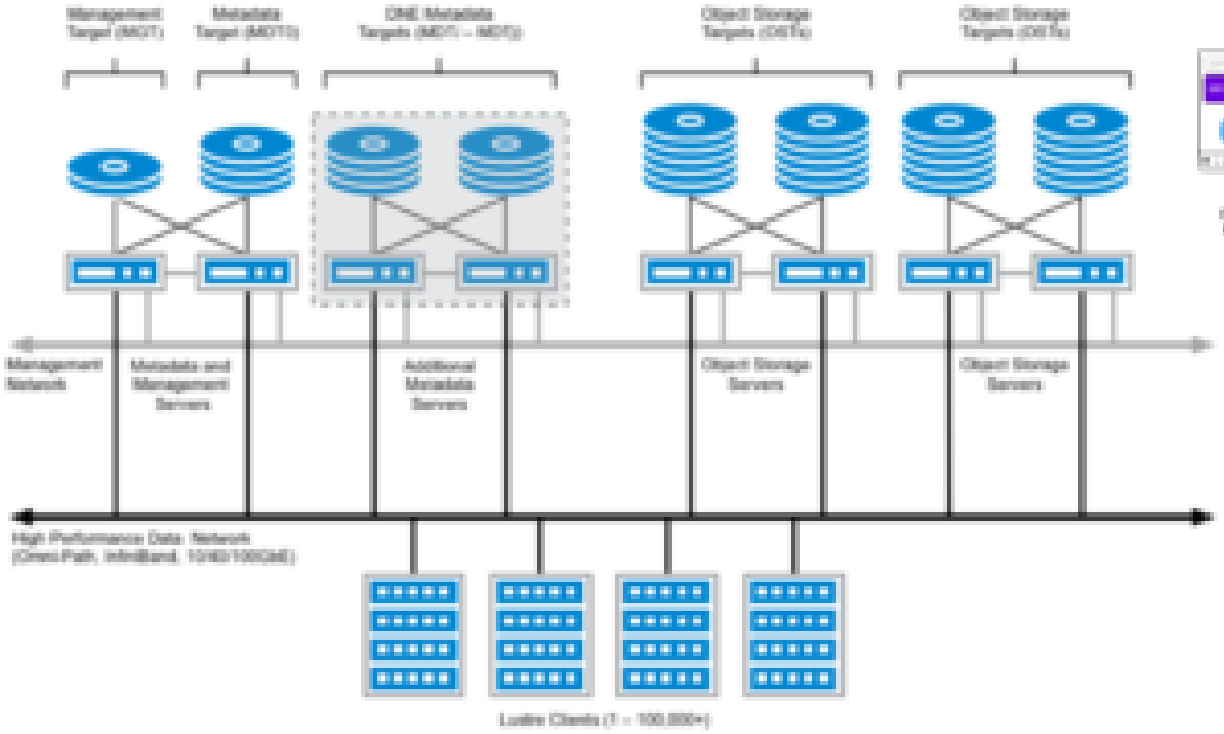
- Queues
- Publish / subscribe
- Event-driven means, that the process will monitor its file descriptors (and sockets), and act only when some event occurs on some descriptor (events are: data received, error, became writeable

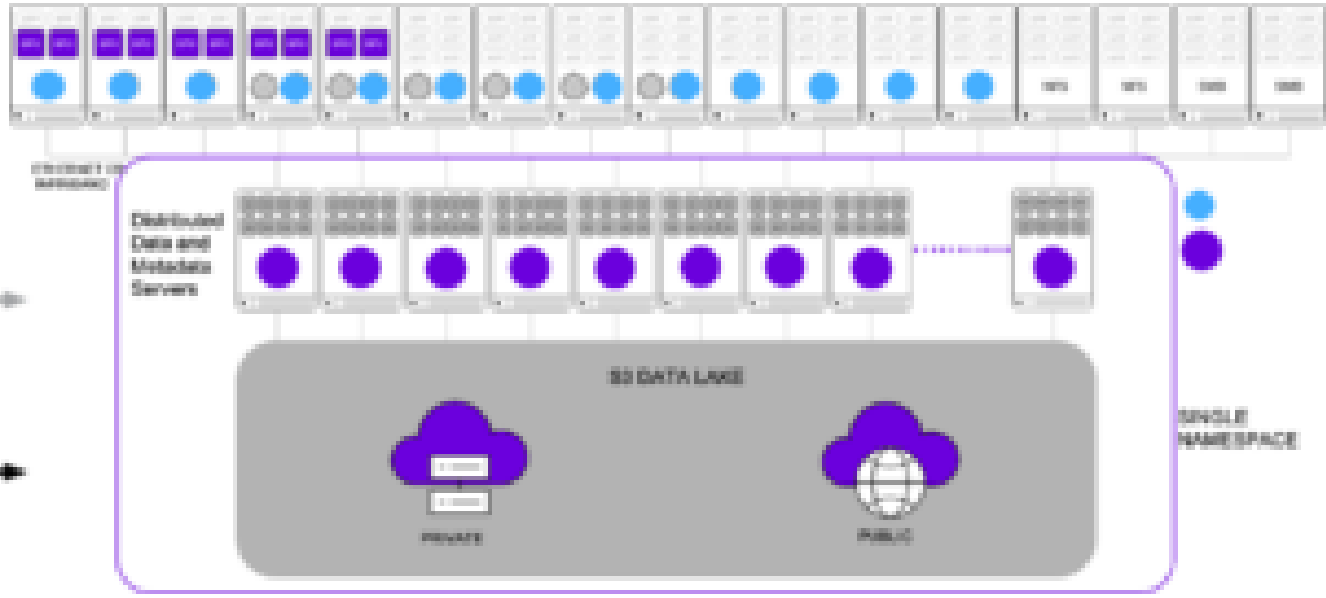


Use Cases of Big Data, High-Performance Data Analytics **and HPC**

- **Fraud Detection**
  - PayPal saved over $700 million in fraudulent transactions that they would not have detected previously.
  - On a daily basis, they deal with 10+ million logins, 13 million transactions and 300 variables that are calculated per event to find a potentially fraudulent transaction
- **Personalized Medicines and Drug Discovery**
  - Reduced time to find the right proteins, eventually saving a lot of lives.
- **Smart Energy Grids**
  - Forecasting the energy requirements
- **Manufacturing Simulation Analysis**
  - simulation analysis can save companies a lot of money when developing new products

HPC Storage


Management Target (MGT)
Metadata Target (MDT0)
ONE Metadata Targets (MDT1 – MDTn)
Object Storage Targets (OST0)
Object Storage Targets (OSTn)
Management Network
Metadata and Management Servers
Additional Metadata Servers
Object Storage Servers
Object Storage Servers
High Performance Data Network
Lustre Clients (1 – 100,000+)

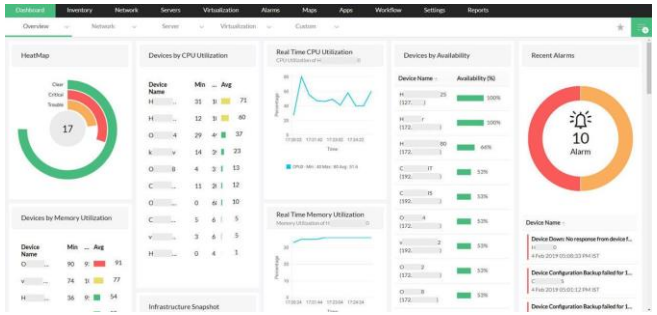

Distributed Data and Metadata Servers
S3 DATA LAKE
PRIVATE
PUBLIC
SINGLE NAMESPACE

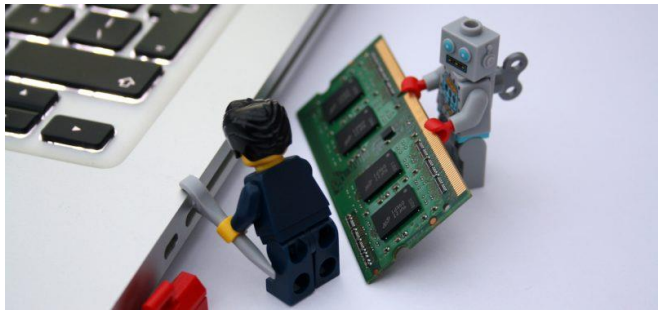
PDS as cloud services

1. Google Firebase
2. Google Cloud DNS
3. Amazon Elastic MapReduce
4. Amazon managed Blockchain
5. Building recommendation systems on Google Cloud: Matrix Factorization in **BigQuery** Machine Learning
6. Google's globally distributed edge points of presence for Content Delivery Network
7. Azure high-performance computing (HPC)
8. Google Cloud GPUs

PDS & HPC for AI in the Cloud

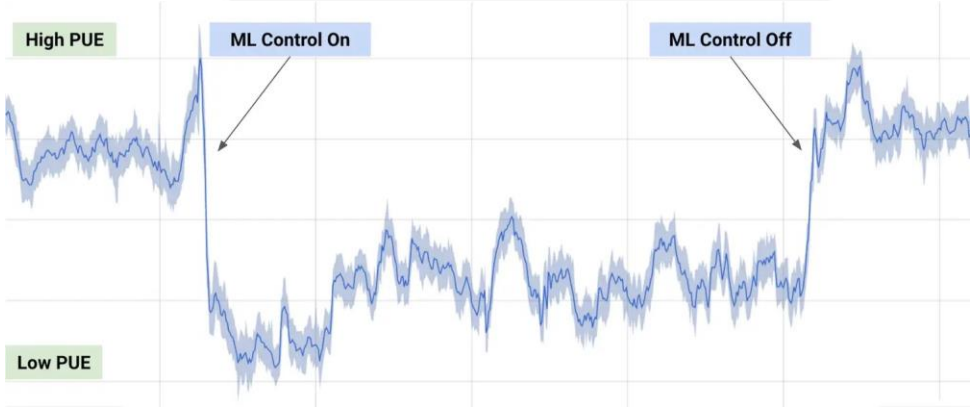
- The link between AI and HPC is symbiotic as HPC powers AI workloads, but AI can identify improvements in HPC data centers.
- AI, for example, can optimize heating and cooling systems, reducing electricity costs and improving efficiency. AI systems can also monitor the health of servers, storage, and networking gear, check to see that systems remain correctly configured, and predict when equipment is about to fail.
- Furthermore, AI can be used for security purposes, screening and analyzing incoming and outgoing data, detecting malware, and implementing behavioral analytics to protect data.





Google's DeepMind trains AI to cut its energy bills by 40%

The AI firm used machine learning to reduce electricity use in its data centres



Thank You!

- Parallel
- And
- Distributed
- Systems

- Performance
- Availability
- Durability
- Scalability