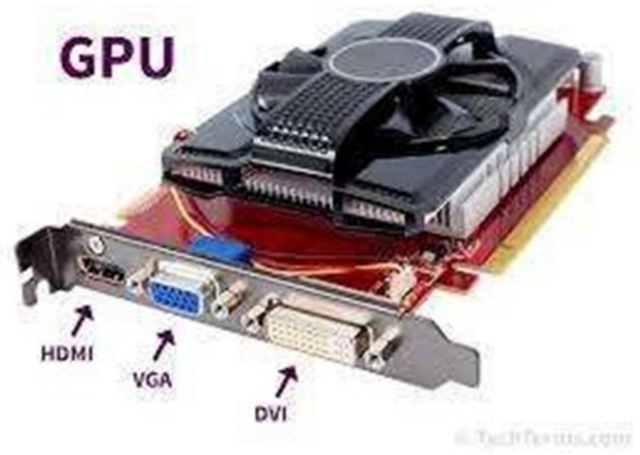
GRAPHICS PROCESSING UNIT AND GPU ARCHITECTURE

WHAT IS A GPU?

The graphics processing unit (GPU) in your device helps handle graphics-related work like graphics, effects, and videos. Learn about the different types of GPUs and find the one that meets your needs. Integrated GPUs are built into your PC's motherboard, allowing laptops to be thin, lightweight, and power-efficient.

- Dedicated graphics chip that handles all processing required for rendering 3D objects on the screen
- Typically placed on a video card, which contains its own memory and display interfaces (HDMI, DVI, VGA, etc)
- Primitive GPUs were developed in the 1980s, although the first “complete” GPUs began in the mid-1990s.



SYSTEMS LEVEL VIEW

- Video card connected to the motherboard through PCI-Express or AGP (Accelerated Graphics Port)
- Northbridge chip enables data transfer between the CPU and GPU
- Graphics memory on the video card contains the pixel RGB data for each frame

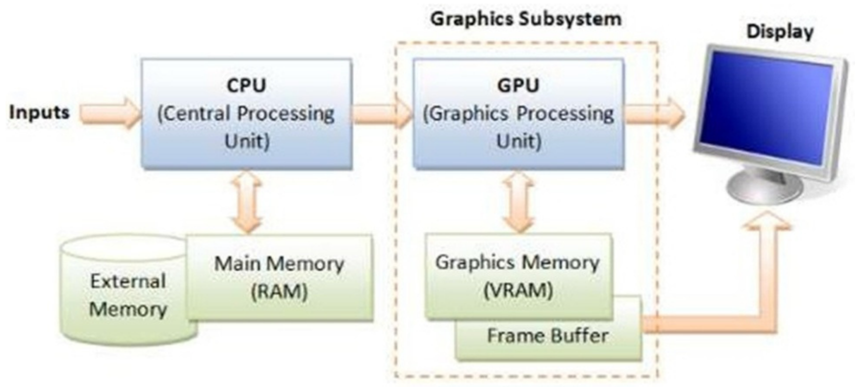
IS GPU SAME AS A GRAPHICS CARD?

A GPU, Graphics Card and Video Card are used interchangeably. To be exact, though, they mean different things. A GPU is the main chip on the Graphics Card. A Graphics Card is a fully functional piece of Hardware (including the GPU) with a PCB, VRAM, and other supporting hardware elements.

WHAT DOES A GPU DO?

The graphics processing unit, or GPU, has become one of the most important types of computing technology, both for personal and business computing. Designed for parallel processing, the GPU is used in a wide range of applications, including graphics and video rendering. Although they’re best known for their capabilities in gaming, GPUs are becoming more popular for use in creative production and artificial intelligence (AI).

GPUs were originally designed to accelerate the rendering of 3D graphics. Over time, they became more flexible and programmable, enhancing their capabilities. This allowed graphics programmers to create more interesting visual effects and realistic scenes with advanced lighting and shadowing techniques. Other developers also began to tap the power of GPUs to dramatically accelerate additional workloads in high-performance computing (HPC), deep learning, and more.



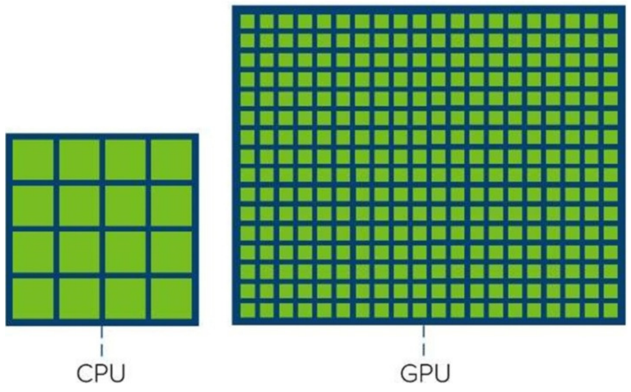
HOW IS A GPU DIFFERENT FROM A CPU?

THROUGHPUT MORE IMPORTANT THAN LATENCY

- High throughput needed for the huge amount of computations required for graphics
- Not concerned about latency because the human visual system operates on a much longer time scale
    - 16 ms maximum latency at 60 Hz refresh rate
    - Long pipelines with many stages; a single instruction may thousands of cycles to get through the pipeline.

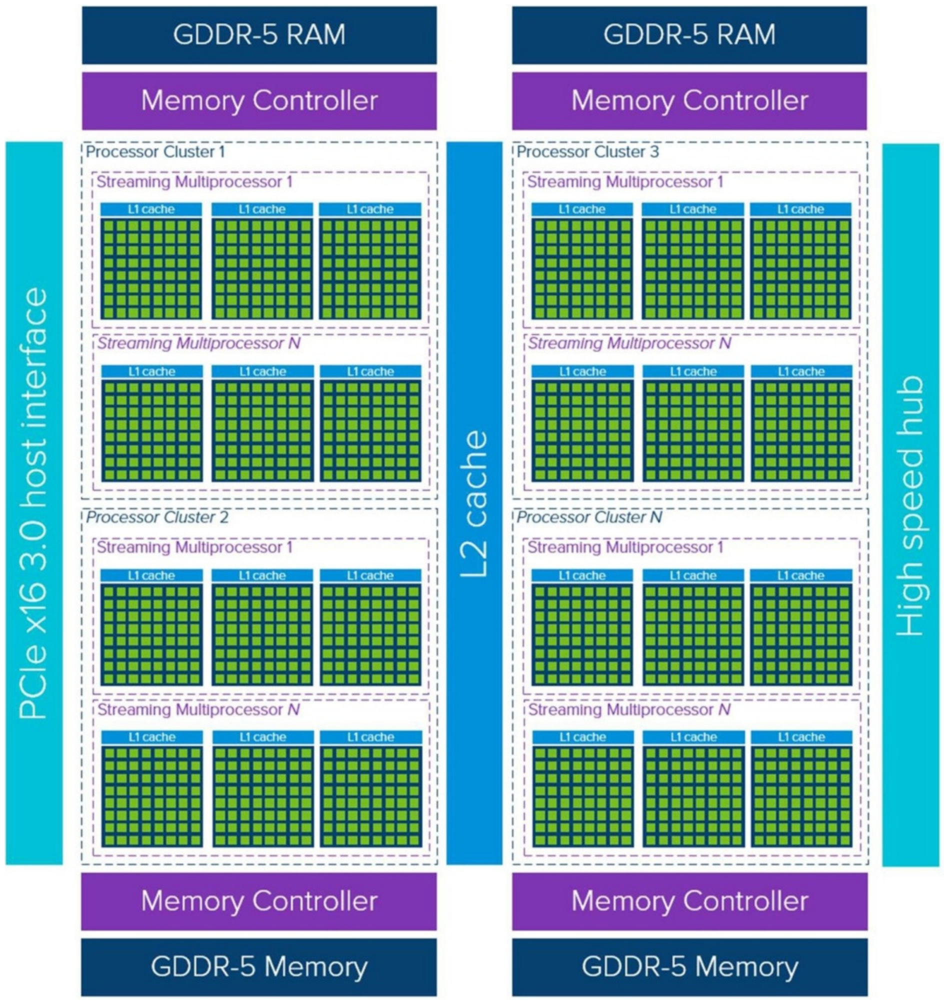
LATENCY VS THROUGHPUT

Let’s first take a look at the main differences between a Central Processing Unit (CPU) and a GPU. A common CPU is optimized to be as quick as possible to finish a task at an as low as possible latency, while keeping the ability to quickly switch between operations. Its nature is all about processing tasks in a serialized way. A GPU is all about throughput optimization, allowing pushing as many as possible tasks through is internals at once. It does so by being able to parallel process a task. The following exemplary diagram shows the ‘core’ count of a CPU and GPU. It emphasizes that the main contrast between both is that a GPU has a lot more cores to process a task.



EXPLORING THE GPU ARCHITECTURE

If we inspect the high-level architecture overview of a GPU (again, strongly dependent on make/model), it looks like the nature of a GPU is all about putting available cores to work and it's less focused on low latency cache memory access.



A single GPU device consists of multiple Processor Clusters (PC) that contains multiple Streaming Multiprocessors (SM). Each SM accommodates a layer-1 instruction cache layer with its associated cores. Typically, one SM uses a dedicated layer-1 cache and a shared layer-2 cache before pulling data from global GDDR-5 (or GDDR-6 in newer GPU models) memory. Its architecture is tolerant of memory latency.

Compared to a CPU, a GPU works with fewer and relatively small, memory cache layers. Reason beingis that a GPU has more transistors dedicated to computation meaning it cares less how long it takes the retrieve data from memory. The potential memory access 'latency' is masked as long as the GPU has enough computations at hand, keeping it busy.

Looking at the numbers of cores it quickly shows you the possibilities on parallelism that is it is capable of. When examining the 2019 NVIDIA flagship offering, the Tesla V100, one device contains 80 SM's, each containing 64 cores making a total of 5120 cores! Tasks aren't scheduled to individual cores, but to processor clusters and SM's. That's how it's able to process in parallel. Now combine this powerful hardware device with a programming framework so applications can fully utilize the

WHAT IS THE DIFFERENCE BETWEEN GPU AND CPU?

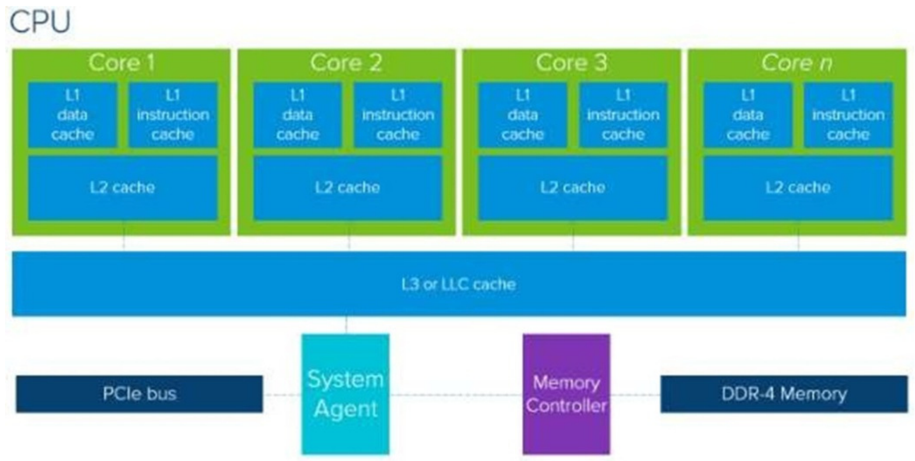
The major difference between GPU and CPU is that GPU has a highly parallel structure which makes it more effective than CPU if used on data that can be partitioned and processed in parallel. To be more specific, GPU is highly optimized to perform advanced calculations such as floating-point arithmetic, matrix arithmetic and so on.

The reason behind the difference of computation capability between CPU and GPU is that GPU is specialized for compute-intensive and highly parallel computations, which is exactly what you need to render graphics. The design of GPU is more of data processing than data caching and flow control. If a problem can be processed in parallel, it usually means two things: first, the same problem is executed for each element, which requires less sophisticated flow control; second, the dataset is massive and the problem has high arithmetic intensity, which reduces the need for low latency memory.

DIFFERENCES AND SIMILARITIES

However, it is not only about the number of cores. And when we speak of cores in an NVIDIA GPU, we refer to CUDA cores that consist of ALUs (Arithmetic Logic Units). The terminology may vary between vendors.

Looking at the overall architecture of a CPU and GPU, we can see a lot of similarities between the two. Both use the memory constructs of cache layers, memory controllers, and global memory. A high-level overview of modern CPU architectures indicates it is all about low latency memory access by using significant cache memory layers. Let's first take a look at a diagram that shows a generic, memory focused, modern CPU package (note: the precise layout strongly depends on vendor/model).



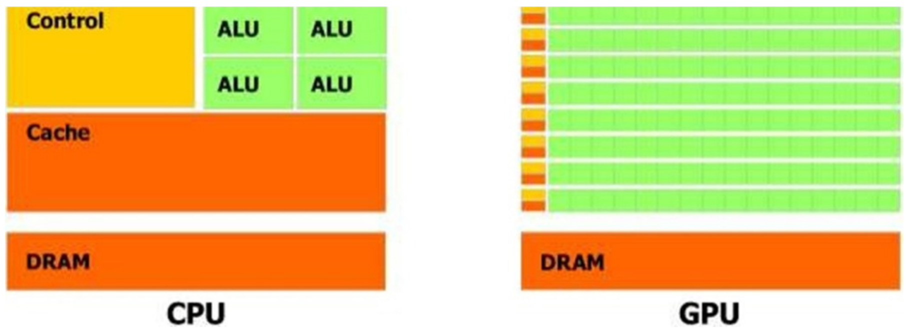
A single CPU package consists of cores that contain separate data and instruction layer-1 caches, supported by the layer-2 cache. The layer-3 cache, or last-level cache, is shared across multiple cores. If data is not residing in the cache layers, it will fetch the data from the global DDR-4 memory. The number of cores per CPU can go up to 28 or 32 that run up to 2.5 GHz or 3.8 GHz with Turbo mode, depending on make and model. Cache sizes range up to 2MB L2 cache per core.

HOW MUCH DO GPU ARCHITECTS MAKE?

"As of Nov 14, 2022, the average annual pay for a GPU Architect in the United States is $134,924 a year."

WHICH GPU IS BEST FOR PROGRAMMING?

NVIDIA GeForce RTX 3080 is the best overall GPU. NVIDIA GeForce RTX 3070 is best for someone on a budget. NVIDIA GeForce RTX 3090 is best for rendering in 3D. NVIDIA GeForce RTX 3080 is best for 4K gaming.



THIS FIGURE IS FROM THE NVIDIA CUDA PROGRAMMING GUIDE

The graph above shows the differences between CPU and GPU in their structure. Cache is designed for data caching; Control is designed for flow control; ALU (Arithmetic Logic Unit) is designed for data processing.
In the NVISION 08 Conference organized by the NVIDIA corporation, employers from NVIDIA used a rather interesting yet straight forward example illustrating the difference between CPU and GPU. You can watch the video by clicking the link below and hope it can give you a better idea about what the difference between GPU and CPU is.

CUDA PARALLEL COMPUTING PLATFORM

Our next step in understanding GPU architecture leads us to Nvidia's popular Compute Unified Device Architecture (CUDA) parallel computing platform. By providing an API that enables developers to optimize how GPU resources are used -- without the need for specialized graphics programming knowledge -- CUDA has gone a long way in making GPUs useful for general purpose computing.
Here, we'll take a look at key CUDA concepts as they relate to GPU architecture.

CUDA COMPUTE HIERARCHY

The processing resources in CUDA are designed to help optimize performance for GPU use cases. Three of the fundamental components of the hierarchy are threads, thread blocks, and kernel grids.

- **THREADS**

  A thread -- or CUDA core -- is a parallel processor that computes floating point math calculations in an Nvidia GPU. All the data processed by a GPU is processed via a CUDA core. Modern GPUs have hundreds or even thousands of CUDA cores. Each CUDA core has its own memory register that is not available to other threads.

  While the relationship between compute power and CUDA cores is not perfectly linear, generally speaking -- and assuming all else is equal -- the more CUDA cores a GPU has, the more compute power it has. However, there are a variety of exceptions to this general idea. For example, different

GPU micro architectures can impact performance and make a GPU with fewer CUDA cores more powerful.

- **THREAD BLOCKS**

  As the name implies, a thread block -- or CUDA block -- is a grouping of CUDA cores (threads) that can be executed together in series or parallel. The logical grouping of cores enables more efficient data mapping. Thread blocks share a memory on a per-block basis. Current CUDA architecture caps the number of threads per block at 1024. Every thread in a given CUDA block can access the same shared memory (more on the different types of memory below).

- **KERNEL GRIDS**

  The next layer of abstraction up from thread blocks is the kernel grid. Kernel grids are groupings of thread blocks on the same kernel. Grids can be used to perform larger computations in parallel (e.g. those that require more than 1024 threads), however since different thread blocks cannot use the same shared memory, the same synchronization that occurs at the block level does not occur at the grid level.

WHAT IS THE ADVANTAGE OF USING GPU FOR COMPUTATION?

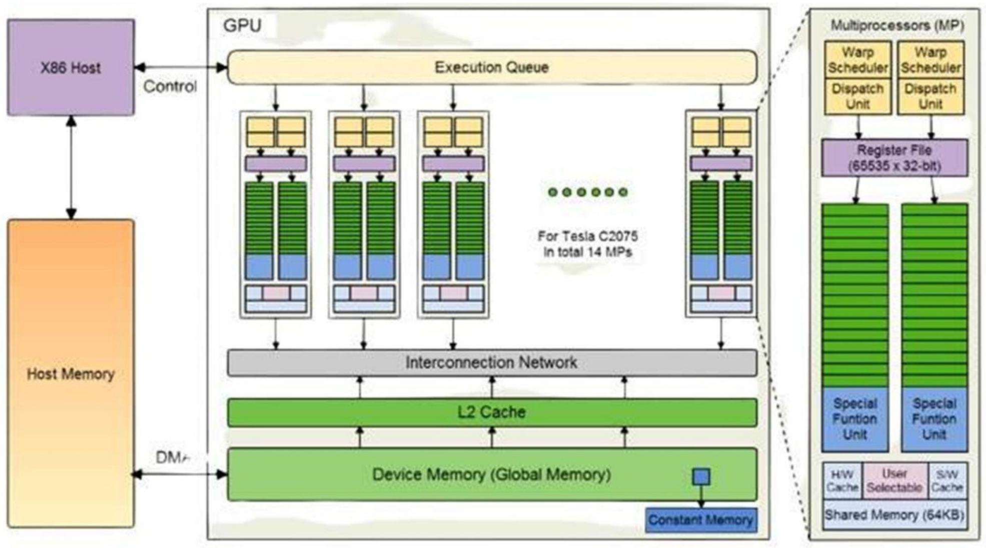
Nowadays, most scientific research requires massive data processing. What scientists usually do right now is to have all the data being processed on supercomputing clusters. Although most universities have constructed their own parallel computing clusters, researchers still need to compete for time to use the shared resources that not only cost millions to build and maintain but also consume hundreds of kilowatts of power.

Different from traditional supercomputers that are built with many CPU cores, supercomputers with a GPU structure can achieve the same level of performance with less cost and lower power consumption. Personal Supercomputer (PSC) based on NVIDIA's Tesla companion processors, was first introduced in 2008. The first-generation four-GPU Tesla personal supercomputer has 4 Teraflops of parallel supercomputing performance, more than enough for most small research.

All it takes is 4 Tesla C1060 GPUs with 960 cores and two Intel Xeon processors. Moreover, a Personal supercomputer is also very energy efficient as it can even run off a 110-volt wall circuit. Although supercomputer servers with GPUs cannot match the performance of the top-ranking supercomputers that cost millions even billions, it is more than enough for researchers to perform daily research-related computations in subjects like bioscience, life science, physics, and geology.

WHAT ARE THE IMPORTANT PARTS OF A GPU?

Although modern GPUs are basically computers themselves, they still serve as a part of a computer system. A modern GPU is connected to the host through a high-speed I/O bus slot, usually a PCI-Express slot. Modern GPUs are extremely energy consuming. Some of the GPUs alone consume hundreds of watts of power, sometimes higher than all other parts of the computer system combined. Part of the reason that GPUs require such a power supply is that they have a much more complex structure and can perform much-sophisticated tasks than other parts of the computer systems. Owing to its high capacity, GPU needs its own memory, control chipset as well as many processors.



This figure is inspired by the figure found in Understanding the CUDA Data Parallel Threading Model: A Primer written by Michael Wolfe from The Portland Group

GPUs these days are usually equipped with several gigabytes of onboard memory for user configuration. GPUs designed for daily use like gaming and video renderings, such as NVIDIA’s Geforce series GPUs and ATI/AMD’s Radeon series GPUs, have onboard memory capacity ranging from several hundred megabytes to several gigabytes. Professional GPUs designed for high-definition image processing and General Purpose Computation, such as the Tesla Companion Processor we are using, usually have memory up to 5 or 6 gigabytes. Data are transferred between the GPU onboard memory and host memory through a method called DMA (Direct Memory Access). One thing worth mentioning is that CUDA C programming language supports direct access to the host memory from the GPU end under certain restrictions. As GPU is designed for compute-intensive operations, device memory usually supports high data bandwidth with no deeply cached memory hierarchy.

GPUs from NVIDIA have many processors, called streaming Processors (SP). Each streaming processor is capable of executing a sequential thread. For a GPU with Fermi architecture, like the one we are using, every 32 streaming processors is organized in a Streaming Multiprocessor (SM). A GPU can have one or more multiprocessors on board. For example, the Tesla C2075 GPU card we are using has 14 multiprocessors built in. Except for 32 streaming processors, each multiprocessor is also equipped with 2 warp schedulers, 2 special function units (4 in some GPUs), a set of 32-bit registers, and 64KB of configurable shared memory. A warp scheduler is responsible for threads control; SFU handles transcendental and double-precision operations. For a GPU with Kepler architecture, every 192 streaming processor is organized in a multiprocessor. There are also more warp schedulers and SFUs built in.

Shared memory, or L1 cache, is a small data cache that can be configured through software. Shared memory is also shared among all the streaming processors within one multiprocessor. Compared with onboard memory, shared memory is low-latency (usually **register** speed) and has high bandwidth. Each multiprocessor has 64KB of shared memory that can be configured by using special commands in the host code. Shared memory is distributed to the software-managed data cache and hardware data cache.

The user can choose to assign either 48KB to the software-managed data cache (SW) and 16KB to the hardware data cache (HW) or the other way around.

IS CUDA THE ONLY GPU PROGRAMMING LANGUAGE AVAILABLE?

When we are learning CUDA C programming language, it is important for you to know that the C programming language is not the only language that can be bound with CUDA structure. NVIDIA also made other programming languages available such as Fortran, Java and Python as binding languages with CUDA.

Furthermore, NVIDIA is not the only company manufacturing GPU cards, which means CUDA is not the only GPU programming MPI available. When NVIDIA were developing CUDA, AMD/ATI responded with ATI Stream, their GPGPU technology for AMD/ATI Radeon series GPUs. ATI Stream technology uses OpenCL as its binding language.

GPU ARCHITECTURE BASICS

Within a GPU device, there are multiple processor clusters (PC), which contain multiple streaming multiprocessors (SM). And, every SM accommodates a layer-1 instruction cache layer together with its associated cores. Usually, one SM adopts a dedicated layer-1 cache and a shared layer-2 cache before pulling data from the global GDDR-5 memory. Therefore, the GPU processor architecture is tolerable for memory latency.

- **GCA (GRAPHICS COMPUTE ARRAY)**
  Usually, a GCA, also known as a 3D engine, consists of pixel shades, vertex shades or unified shades, stream processors (CUDA cores), texture mapping units (TMUs), render output units (ROPs), L2 cache, geometry processors, and so on.

- **GMC (GRAPHICS MEMORY CONTROLLER)**
  The GMC, also known as memory chip controller (MCC) or memory controller unit (MCU), is a digital circuit that controls the data flow going to or from the computer's graphics memory. It can be a separate chip; it can also be integrated into another chip, such as being placed on the same die or as an integral part of a microprocessor. If the GMC exists as an integral part, it is called an IMC (Integrated Memory Controller).
  The memory GMC controls including VRAM, WRAM, MDRAM, DDR, GDDR and HBM.

- **VGA BIOS (VIDEO GRAPHICS ARRAY BASIC INPUT/OUTPUT SYSTEM)**
  VGA BIOS, also known as video BIOS, is the BIOS of a graphics card in a computer. It is a separate chip located on the graphics card, not part of the GPU.

- **BIF (BUS INTERFACE)**
  The bus interface (BI) is a computer bus for interfacing small peripheral devices such as flash memory with the processor. Usually, it includes SA, VLB, PCI, AGP and PCIe.

- **PMU (POWER MANAGEMENT UNIT)**
  The PMU is a microcontroller (microchip) that controls the power functions of digital platforms. It has many similar components to the average computer, such as CPU, memory, firmware, software, etc. The PMU is one of the few components that remain active even when the computer is completely turned off, powered by a backup battery.

- **VPU (VIDEO PROCESSING UNIT)**
  VPU is a specialized processor that takes video streams as input and can execute very complex processes on the input stream. It is usually used in machine learning applications and devices, and functions as an auxiliary component in those devices.
  VPU is a video codec responsible for video encoding and decoding. So, it is also called a video encoder and decoder. VPUs perform the compression or decompression of MPEG2, Theora, VP8, H.264, H.265, VP9, VC-1, etc.

- **DIF (DISPLAY INTERFACE)**
  Display interface, also called display controller, defines a serial bus and a communication protocol among the host, the source of the image data and the destination device. It includes RAMDACs, HDMI audio, DP audio, video underlay (VGA, DVI, HDMI, DisplayPort, S-Video, composite video, component video), PHY (LVDS, TIMDS) and EDID.
  In a nutshell, **GPU architecture** is simple than that of CPU. Graphics processing unit architecture has much more cores than a CPU to achieve parallel data processing with higher tolerate latency. Such kind of GPU is known as general-purpose GPU (GPGPU) used to accelerate computational workloads in modern high-performance computing (HPC).

HETEROGENEITY

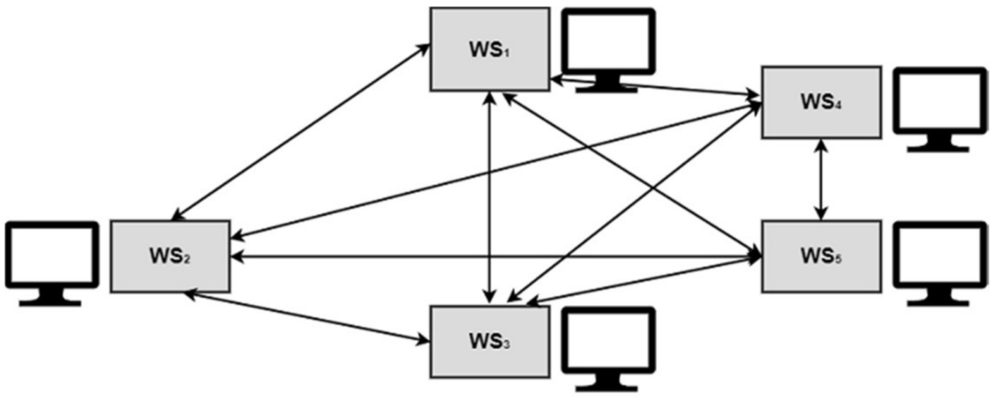
Heterogeneity is one of the most profound and challenging features of today's parallel and distributed computing systems. From the macro level, where networks of distributed computers, composed by diverse node architectures, are interconnected with potentially heterogeneous networks, to the micro level, where deeper memory hierarchies, heterogeneous multicores, and various accelerator architectures are increasingly common, the impact of heterogeneity on all computing tasks is increasing rapidly.

WHAT IS INTERCONNECTION TOPOLOGY?

Interconnection networks are composed of switching elements. Topology is the pattern to connect the individual switches to other elements, like processors, memories and other switches. A network allows exchange of data between processors in the parallel system.

In this topology, every node is connected using a separate physical link. Each computer network has a direct dedicated link, i.e., point to point connection with all other network computers. Each computer has its control and decision power for communication priorities.

It is a very reliable topology because any transmission line's failure only affects the transmission between the two connected computers. The communication or message is high-speed and has mostly a full-duplex mode, but at the same time, a large number of communication lines make the network costly as for nodes lines or links are required.



Fully Connected Network

LOAD BALANCING

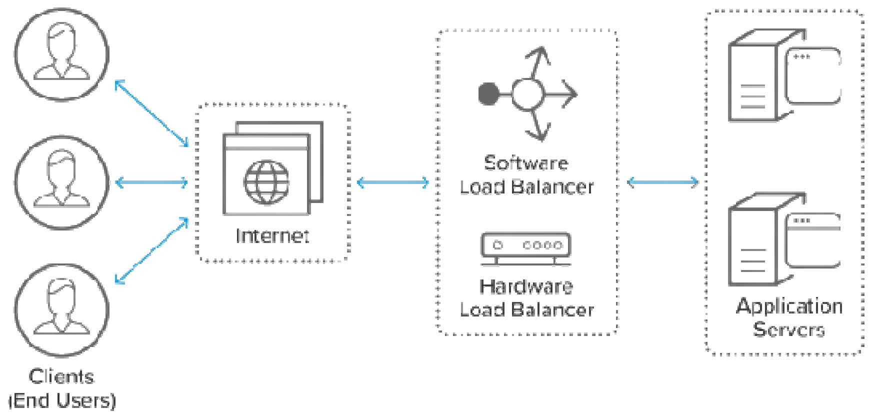
Load balancing refers to efficiently distributing incoming network traffic across a group of backend servers, also known as a server farm or server pool.

Modern high-traffic websites must serve hundreds of thousands, if not millions, of concurrent requests from users or clients and return the correct text, images, video, or application data, all in a fast and reliable manner. To cost-effectively scale to meet these high volumes, modern computing best practice generally requires adding more servers.

A load balancer acts as the "traffic cop" sitting in front of your servers and routing client requests across all servers capable of fulfilling those requests in a manner that maximizes speed and capacity utilization and ensures that no one server is overworked, which could degrade performance. If a single server goes down, the load balancer redirects traffic to the remaining online servers. When a new server is added to the server group, the load balancer automatically starts to send requests to it.

In this manner, a load balancer performs the following functions:

- Distributes client requests or network load efficiently across multiple servers
- Ensures high availability and reliability by sending requests only to servers that are online
- Provides the flexibility to add or subtract servers as demand dictates



LOAD BALANCING ALGORITHMS

Different load balancing algorithms provide different benefits; the choice of load balancing method depends on your needs:

- **Round Robin** – Requests are distributed across the group of servers sequentially.
- **Least Connections** – A new request is sent to the server with the fewest current connections to clients. The relative computing capacity of each server is factored into determining which one has the least connections.
- **Least Time** – Sends requests to the server selected by a formula that combines the fastest response time and fewest active connections. Exclusive to NGINX Plus.
- **Hash** – Distributes requests based on a key you define, such as the client IP address or the request URL. NGINX Plus can optionally apply a consistent hash to minimize redistribution of loads if the set of upstream servers changes.
- **IP Hash** – The IP address of the client is used to determine which server receives the request.
- Random with Two Choices – Picks two servers at random and sends the request to the one that is selected by then applying the Least Connections algorithm (or for NGINX Plus the Least Time algorithm, if so configured).

BENEFITS OF LOAD BALANCING

- Reduced downtime
- Scalable
- Redundancy
- Flexibility
- Efficiency

MEMORY CONSISTENCY MODEL

A memory consistency model is a contract between the hardware and software. The hardware promises to only reorder operations in ways allowed by the model, and in return, the software acknowledges that all such re-orderings are possible and that it needs to account for them.

TYPES OF CONSISTENCY

There are many different models of consistency, some of which are included in the list below:

- **Strict Consistency Model:** "The strict consistency model is the strongest form of memory coherence, having the most stringent consistency requirements. A shared-memory system is said to support the strict consistency model if the value returned by a read operation on a memory address is always the same as the value written by the most recent write operation to that address, irrespective of the locations of the processes performing the read and write operations. That is, all writes instantaneously become visible to all processes."

- **Sequential Consistency Model:** "The sequential consistency model was proposed by Lamport. A shared-memory system is said to support the sequential consistency model if all processes see the same order of all memory access operations on the shared memory. The exact order in which the memory access operations are interleaved does not matter. ... If one process sees one of the orderings of ... three operations and another process sees a different one, the memory is not a sequentially consistent memory."

- **Casual Consistency Model:** "The causal consistency model ... relaxes the requirement of the sequential model for better concurrency. Unlike the sequential consistency model, in the causal consistency model, all processes see only those memory reference operations in the same (correct) order that are potentially causally related. Memory reference operations that are not potentially causally related may be seen by different processes in different orders."
- **FIFO Consistency Model:** For FIFO consistency, "Writes done by a single process are seen by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes." FIFO consistency is called PRAM consistency in the case of distributed shared memory systems."
- **Pipelined Random-Access Memory (PRAM) Consistency Model:** "The pipelined random-access memory (PRAM) consistency model ... provides weaker consistency semantics than the (first three) consistency models described so far. It only ensures that all write operations performed by a single process are seen by all other processes in the order in which they were performed as if all the write operations performed by a single process are in a pipeline. Write operations performed by different processes may be seen by different processes in different orders."
- **Weak Consistency Model:** "Synchronization accesses (accesses required to perform synchronization operations) are sequentially consistent. Before synchronization access can be performed, all previous regular data accesses must be completed. Before a regular data access can be performed, all previous synchronization accesses must be completed. This essentially leaves the problem of consistency up to the programmer. The memory will only be consistent immediately after a synchronization operation."

- **Release Consistency Model:** "Release consistency is essentially the same as weak consistency, but synchronization accesses must only be processor consistent with respect to each other. Synchronization operations are broken down into acquires and release operations. All pending acquires (e.g., a lock operation) must be done before a release (e.g., an unlock operation) is done. Local dependencies within the same processor must still be respected. "Release consistency is a further relaxation of weak consistency without a significant loss of coherence."

- ~~**Entry Consistency Model:** "Like variants of release consistency, it requires the programmer (or compiler) to use acquire and release at the start and end of each critical section, respectively. However, unlike release consistency, entry consistency requires each ordinary shared data item to be associated with some synchronization variable, such as a lock or barrier. If it is desired that elements of an array be accessed independently in parallel, then different array elements must be associated with different locks. When an acquire is done on a synchronization variable, only those data guarded by that synchronization variable are made consistent~~."

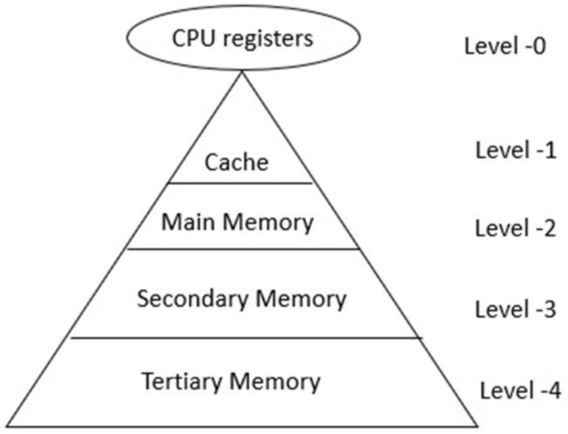
- **Processor Consistency Model:** "Writes issued by a processor are observed in the same order in which they were issued. However, the order in which writes from two processors occur, as observed by themselves or a third processor, need not be identical. That is, two simultaneous reads of the same location from different processors may yield different results."

- **General Consistency Model:** "A system supports general consistency if all the copies of a memory location eventually contain the same data when all the writes issued by every processor have completed."

MEMORY HIERARCHIES

The Computer memory hierarchy looks like a pyramid structure which is used to describe the differences among memory types. It separates the computer storage based on hierarchy.

Level 0: CPU registers
Level 1: Cache memory
Level 2: Main memory or primary memory
Level 3: Magnetic disks or secondary memory
Level 4: Optical disks or magnetic types or tertiary Memory



In Memory Hierarchy the cost of memory, capacity is inversely proportional to speed. Here the devices are arranged in a manner Fast to slow that is form register to Tertiary memory.

LET US DISCUSS EACH LEVEL IN DETAIL:

Level-0 – Registers

The registers are present inside the CPU. As they are present inside the CPU, they have least access time. Registers are most expensive and smallest in size generally in kilobytes. They are implemented by using Flip-Flops.

Level-1 – Cache

Cache memory is used to store the segments of a program that are frequently accessed by the processor. It is expensive and smaller in size generally in Megabytes and is implemented by using static RAM.

Level-2 – Primary or Main Memory

It directly communicates with the CPU and with auxiliary memory devices through an I/O processor. Main memory is less expensive than cache memory and larger in size generally in Gigabytes. This memory is implemented by using dynamic RAM.

Level-3 – Secondary storage

Secondary storage devices like Magnetic Disk are present at level 3. They are used as backup storage. They are cheaper than main memory and larger in size generally in a few TB.

Level-4 – Tertiary storage

Tertiary storage devices like magnetic tape are present at level 4. They are used to store removable files and are the cheapest and largest in size (1-20 TB).

Let us see the memory levels in terms of size, access time, Bandwidth.

LEVEL	REGISTER	CACHE	PRIMARY MEMORY	SECONDARY MEMORY
Bandwidth	4k to 32k MB/sec	800 to 5k MB/sec	400 to 2k MB/sec	4 to 32 MB/sec
Size	Less than 1KB	Less than 4MB	Less than 2 GB	Greater than 2 GB
Access time	2 to 5nsec	3 to 10 nsec	80 to 400 nsec	5ms
Managed by	Compiler	Hardware	Operating system	OS or user

WHY MEMORY HIERARCHY IS USED IN SYSTEMS?

Memory hierarchy is arranging different kinds of storage present on a computing device based on speed of access. At the very top, the highest performing storage is CPU registers which are the fastest to read and write to. Next is cache memory followed by conventional DRAM memory, followed by disk storage with different levels of performance including SSD, optical and magnetic Disk Drives.

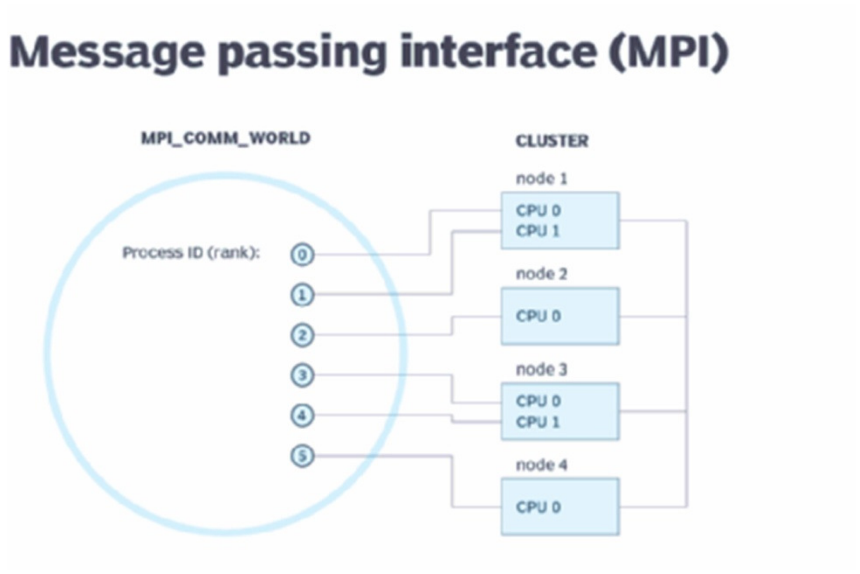
To bridge the processor memory performance gap, hardware designers are increasingly relying on memory at the top of the memory hierarchy to close / reduce the performance gap. This is done through increasingly larger cache hierarchies (which can be accessed by processors much faster), reducing the dependency on main memory which is slower.

MESSAGE PASSING INTERFACE (MPI)

MPI stands for **Message Passing Interface**. The message passing interface (MPI) is a standardized means of exchanging messages between multiple computers running a parallel program across distributed memory.

In parallel computing, multiple computers – or even multiple processor cores within the same computer – are called nodes. Each node in the parallel arrangement typically works on a portion of the overall computing problem. The challenge then is to synchronize the actions of each parallel node, exchange data between nodes, and provide command and control over the entire parallel cluster. The message passing interface defines a standard suite of functions for these tasks. The term *message passing* itself typically refers to the sending of a message to an object, parallel process, subroutine, function or thread, which is then used to start another process.

MPI defines useful syntax for routines and libraries in programming languages including Fortran, C, C++ and Java.



BENEFITS OF THE MESSAGE PASSING INTERFACE

The message passing interface provides the following benefits:

1. **Standardization.** MPI has replaced other message passing libraries, becoming a generally accepted industry standard.
2. **Developed by a broad committee.** Although MPI may not be an official standard, it's still a general standard created by a committee of vendors, implementers and users.
3. **Portability.** MPI has been implemented for many distributed memory architectures, meaning users don't need to modify source code when porting applications over to different platforms that are supported by the MPI standard.
4. **Speed.** Implementation is typically optimized for the hardware the MPI runs on. Vendor implementations may also be optimized for native hardware features.
5. **Functionality.** MPI is designed for high performance on massively parallel machines and clusters. The basic MPI-1 implementation has more than 100 defined routines.

WHAT IS SIMD?

SIMD is a form of parallel computer architecture that is categorized under Flynn's classification given by Michael Flynn. In the SIMD architecture, a single instruction is applied to several data streams. SIMD consists of a single control signal that is used to call several isolated processing units. Therefore, all the processing units accept the same instruction from the control unit and use it on separate elements of data.
The SIMD organization uses shared memory unit which is divided into different modules. As a result, the memory unit can interact with all the processing units simultaneously. Since the SIMD architecture uses a single copy of instruction on multiple data streams, it requires less memory. Also, SIMD requires a single instruction decoder which reduces the overall cost of the system.
SIMD architectures are particularly effective for tasks that can be easily parallelized, such as image processing, video encoding and decoding, etc.

WHAT IS MIMD?

MIMD is an abbreviation for Multiple Instruction and Multiple Data Stream. The MIMD architecture consists of multiple instructions and data streams. Therefore, MIMD architecture requires multiple processing units. For this reason, MIMD systems are considered to have most complex organization, but they are highly efficient.
The MIMD architecture uses several instructions over different data streams simultaneously. This provides high concurrency. The MIMD system can work with shared and distributed memory model efficiently.
The MIMD architecture does not need any additional control unit which reduces the cost of the system. It also provides efficient execution of the conditional statements such as if/else statements. This is because the processing unit is independent.
MIMD architectures are more flexible and are better suited for tasks that require more complex and varied computation, such as general–purpose computing and AI applications.

DIFFERENCE BETWEEN SIMD AND MIMD

Sr.	SIMD	MIMD
1.	SIMD stands for Single Instruction Multiple Data.	MIMD stands for Multiple Instruction Multiple Data.
2.	It requires less memory.	It requires more memory.
3.	It is inexpensive in comparison to MIMD.	It is expensive in comparison to SIMD.
4.	It has a single decoder.	It contains multiple decoders.
5.	It uses latent (tacit) synchronization.	It uses accurate (explicit) synchronization.
6.	It is a synchronous programming technique.	It is an asynchronous programming technique.
7.	It is simple in comparison to MIMD.	It is complex in comparison to SIMD.
8.	It is not as efficient as MIMD in terms ofperformance. / It is efficient in comparison to SIMD.	

CONCLUSION

SIMD allows synchronous processing where a single instruction is carried out on multiple data streams at the same time, whereas the MIMD architecture follows the asynchronous mechanism where multiple instructions operate on multiple data streams. The SIMD architecture is less complex as compared to the MIMD architecture.

In general, both SIMD and MIMD architectures can be useful for improving the performance of certain types of computational tasks. SIMD architectures provide more parallelism but they are less flexible, while MIMD architectures provide more flexibility with less parallelism.

WHAT IS MULTITHREADING?

Multithreading is a model of program execution that allows for multiple threads to be created within a process, executing independently but concurrently sharing process resources. Depending on the hardware, threads can run fully parallel if they are distributed to their own CPU core.

WHAT IS MULTITHREADING USED FOR?

The main reason for incorporating threads into an application is to improve its performance. Performance can be expressed in multiple ways:

- A web server will utilize multiple threads to simultaneous process requests for data at the same time.
- An image analysis algorithm will spawn multiple threads at a time and segment an image into quadrants to apply filtering to the image.
- A ray-tracing application will launch multiple threads to compute the visual effects while the main GUI thread draws the final results.

Multithreading also leads to minimization and more efficient use of computing resources. Application responsiveness is improved as requests from one thread do not block requests from other threads.

Additionally, multithreading is less resource-intensive than running multiple processes at the same time. There is much more overhead, time consumption, and management involved in creating processes as compared to creating and managing threads.

WHAT IS AN EXAMPLE OF MULTITHREADING?

Most of the applications that you use on a daily basis have multiple threads running behind the scenes. Consider your internet browser. At any given time, you may have numerous tabs open, each one displaying various types of content. Multiple threads of execution are used to load content, display animations, play a video, and so on.

COMMON ISSUES IN MULTITHREADED APPLICATIONS

For all the advantages of using multiple threads, they add complexity and can create tough bugs to solve. There are some common scenarios where you may encounter challenges with debugging multithreaded applications. These include:

- Investigating data access issues where two threads are reading and modifying the same data. Without the proper use of locking mechanisms, data inconsistency and dead-lock situations can arise.
- Thread starvation and resource contention issues arise if many threads are trying to access a shared resource.
- Display issues can surface if threads are not coordinated correctly when displaying data.

PARALLEL ALGORITHM

An algorithm is a sequence of steps that take inputs from the user and after some computation, produces an output. A parallel algorithm is an algorithm that can execute several instructions simultaneously on different processing devices and then combine all the individual outputs to produce the final result.

Depending on the architecture of computers, we have two types of algorithms

- **Sequential Algorithm**: An algorithm in which some consecutive steps of instructions are executed in a chronological order to solve a problem.
- **Parallel Algorithm:** The problem is divided into sub-problems and is executed in parallel to get individual outputs. Later on, these individual outputs are combined together to get the final desired output.

It is not easy to divide a large problem into sub-problems. Sub-problems may have data dependency among them. Therefore, the processors have to communicate with each other to solve the problem.

It has been found that the time needed by the processors in communicating with each other is more than the actual processing time. So, while designing a parallel algorithm, proper CPU utilization should be considered to get an efficient algorithm.

To design an algorithm properly, we must have a clear idea of the basic model of computation in a parallel computer.

MODEL OF COMPUTATION

Both sequential and parallel computers operate on a set (stream) of instructions called algorithms. These set of instructions (algorithm) instruct the computer about what it has to do in each step.

Depending on the instruction stream and data stream, computers can be classified into four categories:

- Single Instruction stream, Single Data stream (SISD) computers
- Single Instruction stream, Multiple Data stream (SIMD) computers
- Multiple Instruction stream, Single Data stream (MISD) computers
- Multiple Instruction stream, Multiple Data stream (MIMD) computers

WHAT IS PERFORMANCE ANALYSIS?

Performance Analysis is the process of studying or evaluating the performance of a particular scenario in comparison of the objective which was to be achieved. Performance analysis can be do in finance on the basis of ROI, profits etc. In HR, performance analysis can help to review an employee's contribution towards a project or assignment, which he/she was allotted.

The performance analysis step consists of 3 basic steps:

1) **Data Collection:** It is a process by which data related to performance of a program is collected. They are generally collected in a file, it may be presented to a real user in a real time. The basic data collection techniques are:
   - **Profiles:** It records the time spent in different parts of the program. This process in very important for highlighting performance problems. They are gathered automatically.
   - **Counters:** It records frequencies or cumulative number of events. It may require programmer intervention.
   - **Event:** It records each occurrence of various specified events. It thus produces a large number of data. It can be produced automatically or with programmer intervention.

2) **Data Transformation:** It is applied often to reduce the volume of data. For example, a profile recording the minutes spent in each sub routine job on each processor might be transformed to determine minutes spent in each subroutine on each processor and the standard deviation from this mean.

3) **Data Visualization:** Although data reduction techniques can rescue the volume of data, it is often necessary to explore raw data. This process can benefit much more from the use of data visualization techniques.
   When a particular tool is selected for a particular task, the following issues are considered:

   - **Accuracy:** Performance data that we get using the sampling technique are less accurate than data we get using counters or timers.
   - **Simplicity:** The best tools are that collect data automatically without much programmer intervention.
   - **Flexibility:** A flexible tool can be extended to collect additional information or to provide multiple views of the same data.
   - **Intrusiveness:** We need to take into account the overheads when analyzing data.
   - **Abstraction:** A good performance tool allows that data to be judged at a level of abstraction which is suitable for the programming model of parallel programs.

PARALLEL PROGRAMMING MODELS

The von Neumann machine model assumes a processor able to execute sequences of instructions. An instruction can specify, in addition to various arithmetic operations, the address of a datum to be read or written in memory and/or the address of the next instruction to be executed. While it is possible to program a computer in terms of this basic model by writing machine language, this method is for most purposes prohibitively complex, because we must keep track of millions of memory locations and organize the execution of thousands of machine instructions.

Hence, modular design techniques are applied, whereby complex programs are constructed from simple components, and components are structured in terms of higher-level abstractions such as data structures, iterative loops, and procedures. Abstractions such as procedures make the exploitation of modularity easier by allowing objects to be manipulated without concern for their internal structure. So do high-level languages such as FORTRAN, Pascal, C, and Ada, which allow designs expressed in terms of these abstractions to be translated automatically into executable code.

- **Data parallel**

  Data parallelism is a programming model in which a program is executed concurrently on multiple processors, working on different parts of the data set. The goal is to increase the overall processing speed by distributing the work across multiple processors. A data-parallel model focuses on performing operations on a data set, typically a regularly structured array. A set of tasks will operate on this data, but independently on disjoint partitions.

  There are several ways to implement data parallelism in a program. One common approach is to use a parallel programming library or framework, such as Open-MP or Intel TBB, which provides a set of APIs for specifying parallel regions of code and data partitioning. Another approach is to use a high-level programming model such as Map-Reduce, which abstracts away the details of parallel execution and data partitioning, allowing the programmer to specify a map function and a reduce function that are applied to the data set in parallel.

Data parallelism can be an effective way to scale the performance of a program on parallel hardware, such as a multi-core CPU or a GPU. However, it is important to carefully design the datapartitioning and communication patterns to ensure good load balance and avoid bottlenecks.

- **Task Parallel**

  A task-parallel model focuses on processes, or threads of execution. These processes will often be behaviorally distinct, which emphasizes the need for communication. Task parallelism is a natural way to express message-passing communication.

- **Process-Centric**

  Process-Centric is a programming paradigm that separates the concerns of data structures and the concurrent processes that act upon them. The data structures in this case are typically persistent, complex, and large scale - the subject of general purpose applications, as opposed to specialized processing of specialized data sets seen in high productivity applications (HPC).

  The model allows the creation of large scale applications that partially share common data sets. Programs are functionally decomposed into parallel processes that create and act upon logically shared data.

- **Shared/Distributed Memory**

  In a **shared memory programming model**, multiple processors or cores in a computer have access to a common memory space, allowing them to share data and communicate with each other through shared variables. This makes it easy to write parallel programs, as the programmer does not have to worry about explicit data transfer or communication between processors.

  However, shared memory systems can suffer from performance bottlenecks due to contention for shared resources, and they may require explicit synchronization mechanisms to ensure correct execution.

  In a **distributed memory programming mode**l, each processor or core has its own private memory space and must explicitly send and receive data to communicate with other processors. This can be more complex to program than shared memory, as the programmer must explicitly handle data transfer and communication between processors.

  However, distributed memory systems can scale better to large numbers of processors, as there is no contention for shared resources and the communication overhead can be more easily managed.