Why Parallel and Distributed Computing (PDC)?

To meet the demands of modern computing tasks that are **too large, too complex, or too time-sensitive** to be handled efficiently by a single processor or system.

- To achieve performance and speed, tasks are broken into smaller sub-tasks and executed simultaneously to help reduce the execution time significantly.
- To handle massive volumes of data that single machines cannot process or store. The data is divided into chucks and processed across multiple cores (parallel) or nodes (distributed).
- To ensure systems can grow with increasing user demands or data volumes. Horizontal vs. Vertical Scalability.
- Resource Sharing: CPU, storage, specialized hardware

Flynn's Classical Taxonomy to Classify Parallel Systems

SISD: Classic desktop running a single-threaded program (e.g., simple C program).
SIMD: Pi value calculation using Monti Carlo methods across number of processors.
MISD: Multiple image processing operations on the same image
MIMD: Web Server handling multiple types of requests across multiple processors

In **Shared Memory Architecture** all processors have direct (usually bus based) access to common physical memory. In a programming sense, it describes a model where parallel tasks all have the same "picture" of memory and can directly address and access the same logical memory locations regardless of where the physical memory actually exists.

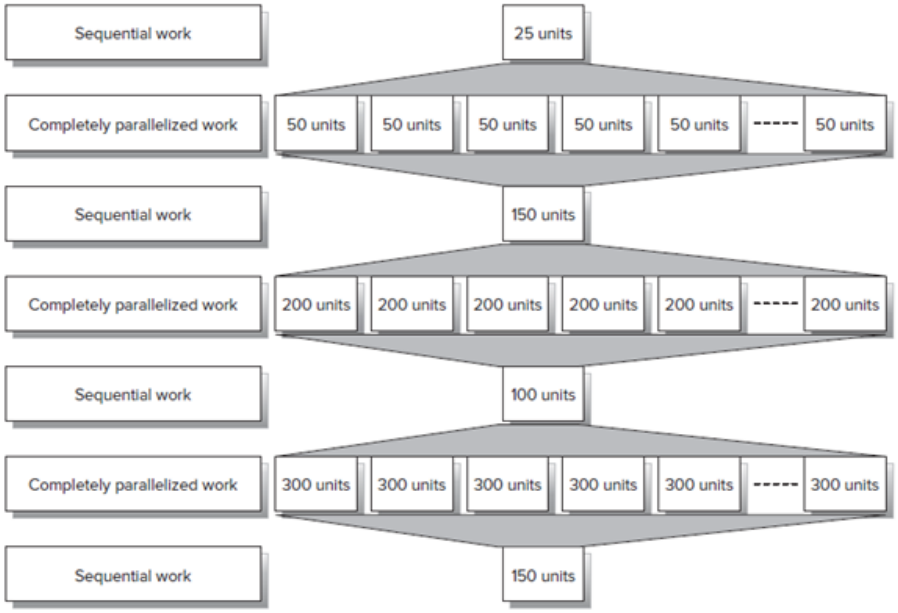
- Pros: Low communication overhead, easy programming
- Cons: Scalability issues, memory access bottlenecks

Distributed Memory Architecture has network-based memory access for physical memory that is not common. As a programming model, tasks can only logically "see" local machine memory and must use communications to access memory on other machines where other tasks are executing.

- Pros: High scalability, cost-effective
- Cons: Complex programming, data consistency challenges

Key differences between shared and distributed memory parallel architectures are their memory access and communication mechanisms. In shared memory, all processors access a common memory space, allowing for easy data sharing. In contrast, distributed memory systems have each processor with its own local memory, requiring explicit communication (like message passing) to share data.

PDC accelerates computing speeds providing multiplicity of data paths.



Total sequential work (in units) = 25 + 150 + 100 + 150 = 425 units of work
Total parallel units of work (in units) = 50 + 200 + 300 = 550 units of work
Critical Path Length= 25 + 50 + 150 + 200 + 100 + 300 + 150 = 975 units of work
Total work (in units) = 425 + (8 × 550) = 4,825 units of work
What happens if we have 16 threads? What happens if during parallel work, some threads take more time than others?

Designing Parallel Programs

- **Understand the Problem**: Can the problem be parallelized? Is it worth the effort? Identity hotspots and bottlenecks.
- **Partitioning**: Domain (data) vs. functional (task)
- **Communication**: Embarrassingly parallel. Synchronous vs. Asynchronous
- **Synchronization**: Barriers, locks/semaphore. Needed to handle dependencies (True, Anti, Output)
- **Load Balancing**: Dynamic Work assignment

To exploit **proper parallelism** from a problem**,** it requires **designing the proper communication and synchronization mechanisms** between the processes and sub-tasks, which is a hectic process.

Quantifying the possible gains from parallelization

- Speedup = Serial/Parallel or Old Time/New Time, depending on the situation
- Amdahl's Law (Strong Scaling): Real vs. Ideal parallelism

$$\boldsymbol{Speedup} = \frac{\mathbf{1}}{\boldsymbol{S} + \frac{\boldsymbol{P}}{\boldsymbol{N}}}$$

What happens when N → ∞
Overestimates speedup as parallel overhead is ignored.

Parallel Overhead: Required execution time that is unique to parallel tasks, as opposed to that for doing useful work. Parallel overhead can include factors such as:

- Task start-up time
- Synchronizations
- Data communications
- Software overhead imposed by parallel languages, libraries, operating system, etc.
- Task termination time

- Gustafson-Barsis's Law (Weak Scaling): Overestimates speedup as parallel overhead is ignored.

$$\boldsymbol{speedup} \leq \boldsymbol{N} + (\mathbf{1} - \boldsymbol{N})\boldsymbol{S}$$

- Karp-Flatt metri: Calculates the serial fraction for a given parallel configuration and determines whether the principal barrier to speedup is due to inherently sequential code or parallelization overhead

$$e = \frac{{}^{1}/_{\boldsymbol{\Psi}(\boldsymbol{N})} - {}^{1}/_{N}}{1 - {}^{1}/_{N}}$$

If e is not increasing, sequential portion of code is decreasing efficiency as we move to larger machines. If e is increasing, parallel overhead also contributes to poor speedup and therefore efficiency will decrease increasingly rapidly as we move to larger machines.

- Isoefficiency metric: Evaluate the scalability of a parallel program executing on a parallel computer

$$W \geq \Theta(\boldsymbol{T_0}(\boldsymbol{N}))$$

Isoefficiency Function	Problem Size
$\mathrm{W} = \mathrm{O}(1)$ (constant)	Perfect scalability (ideal but unrealistic)
$\mathrm{W} = \mathrm{O}(\log N)$	Excellent scalability
$\mathrm{W} = \mathrm{O}(N)$	Good scalability
$\mathrm{W} = \mathrm{O}(N^2)$	Poor scalability
$\mathrm{W} = \mathrm{O}(e^N)$	Extremely poor scalability (unscalable)

Parallel Algorithms

You should know the working of, and how to dry run the following algorithms:

- Parallel Prefix Sum
- Odd-Even Sort
- Odd Even Merge Sort
- Parallel Shell Sort

Cache Coherence MESI Protocol

- Write Invalidate, Write Back
- Write Invalidate, Write Through
- Write Update, Write Back
- Write Update, Write Through

Distributed Systems

CAP Theorem: A distributed system can satisfy any two of these guarantees (Consistency, Availability, Partition Tolerance) at the same time but not all three.

The 6 V's of Big Data

- Volume: Amount of Data is increasing rapidly
- Velocity: Data is being generated fast and need to be processed fast
- Variety: Various types, formats, and structures of data
- Veracity: Accuracy, trustworthiness, and reliability of data
- Variability: Inconsistency in data flow or meaning.
- Value: The usefulness or business value that can be extracted from data.

Examples:

1. A telecom company collects call detail records, customer browsing data, and location information from millions of users every second. The data comes in different formats including JSON logs, audio, and structured tables. Based on the above scenario, identify at least **three V's of Big Data** that are most relevant and explain **why** they apply.
2. An e-commerce company analyzes customer reviews, product ratings, click behavior, and purchasing history to recommend products. However, they notice that many reviews contain slang, emojis, and misspellings. Which of the **6 V's of Big Data** are presenting challenges in this scenario, and how can the company address them?
3. A weather forecasting organization stores 50 years of global climate data. The size is in petabytes, and the data is constantly growing. Some of the sensors that collect the data are unreliable and report incorrect values. Which V's of Big Data are relevant here?

Cloud Computing

- Cloud Computing Models: IaaS, PaaS, SaaS
- Cloud Deployment Models: Public, Private, Hybrid, Community

Example: Choosing the Optimal Cloud Service Model

A startup wants to deploy a mobile backend service and is evaluating three cloud models (IaaS, PaaS, SaaS) for the first month of usage. They expect the following resource consumption:

- 3 virtual machines (each with 4 vCPUs, 16 GB RAM)
- 800 GB storage
- 3 million API requests
- 10 hours of system administration and deployment time

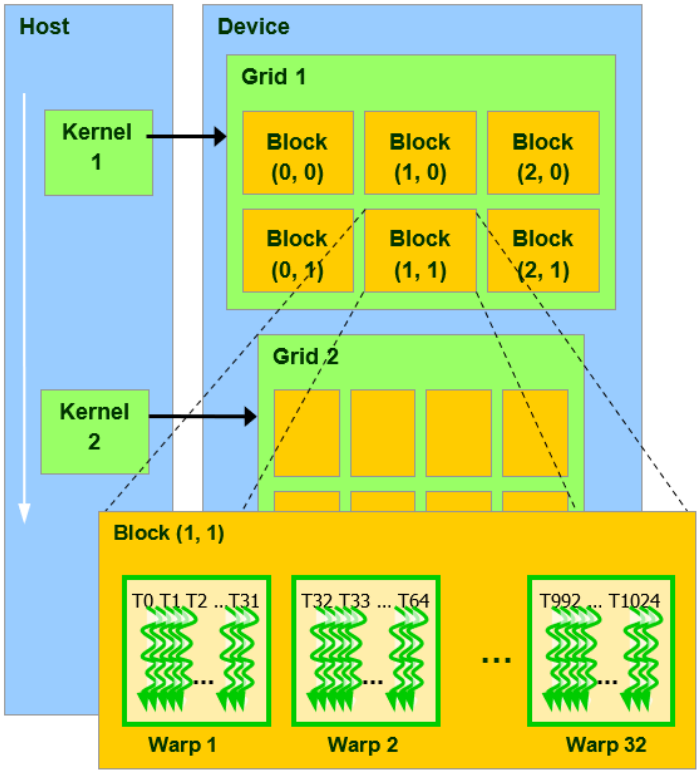
The cloud provider offers the following pricing structure:

Service Model	VM Cost	Storage Cost	API Requests	Admin Cost	Billing Model	Notes
IaaS	$60 per VM	$0.12/GB	$0.001 per 1k req	$35/hr	Pay-per-use	Full control of infrastructure
PaaS	$300 base (includes 3 VMs, 500 GB, 2M requests)	$0.08/GB extra	$0.002 per 1k extra req	$15/hr	Pay-per-use for overages	Managed environment
SaaS	$600 flat (up to 5 users, 5M requests, 1 TB storage)	-	-	-	Flat-rate	No infrastructure management

1. Calculate the total monthly cost under each model.
2. Identify the most cost-effective model for this usage.
3. If the startup wants root access to servers and full environment control, which model is most appropriate?

GPU Architecture and Programming

(a) In the context of GPU computing, illustrate and explain with the help of a diagram the difference between thread, block, grid, and warp. Also specify how the threads are organized within a thread block.



Thread is the smallest unit of execution on a GPU. Each thread executes the same program (kernel) but on different pieces of data (SIMD model). It has its own registers and local memory.

Block (Thread Block) is a group of threads that **execute together** and can **cooperate** via shared memory, thread synchronization (e.g., barriers). It typically has dimensions like **1D, 2D, or 3D** (e.g., blockDim.x, blockDim.y, blockDim.z). All threads in a block are scheduled to run on the same **Streaming Multiprocessor (SM)**.

Grid is a collection of **thread blocks** that execute the same kernel function. Blocks in a grid are **independent** and do not share memory. The grid can also be **1D, 2D, or 3D**.

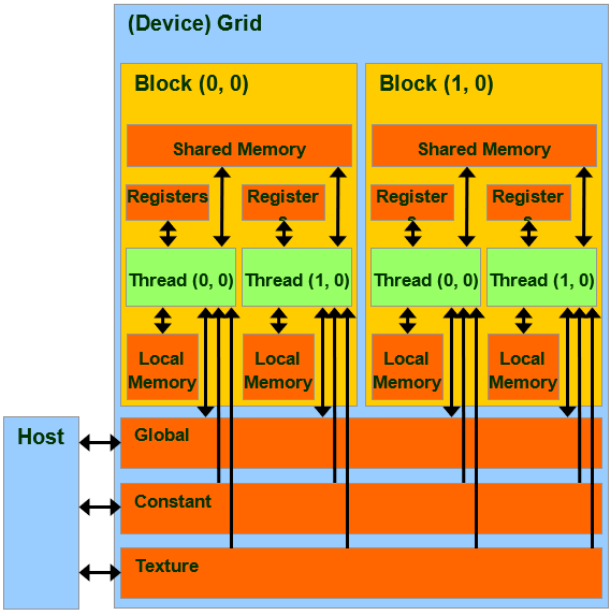
Warp is a group of **32 threads** within a thread block that execute **in lockstep** (SIMT - Single Instruction, Multiple Threads). It is the basic scheduling unit on the GPU hardware. Threads in the same warp execute the same instruction at the same time (unless divergent).

Threads are arranged in a **1D, 2D, or 3D structure** within the block. Each thread has a unique **thread index**: threadIdx.x, threadIdx.y, threadIdx.z. Thread blocks are similarly indexed: blockIdx.x, blockIdx.y, etc.

(b) GPUs are designed to execute thousands of threads in parallel, and one of the key techniques they use to hide memory latency and keep computation units busy is rapid context switching between warps. Briefly describe how GPUs enable fast warp-level switching and why this is effective for maintaining high throughput.

Solution: GPUs achieve memory latency hiding via massive fine-grained multithreading. When a warp (a group of 32 threads) encounters a long-latency memory operation, the GPU's warp scheduler simply switches to another warp that is ready to execute keeping the processing units busy. GPUs have *large registers* that store the architectural register states of all "live" warps scheduled on a GPU core. This enables the hardware to context switch from one warp to another, without having to save (old thread's) and restore (new thread') register states from memory through a software OS routine. The register values for all threads remain in the register file, with no expensive OS context switching. This latency hiding is critical to the GPU's performance model because global memory access can take hundreds of cycles and have simpler control logic with no branch prediction.

(c) Explain the memory hierarchy in GPUs and describe the characteristics of each memory type. How are these memory types accessed and used by threads, blocks, and grids during GPU program execution? Illustrate your explanation with the help of a clear and well-labeled diagram.



Solution: Access Pattern and Usage

Threads:

- Each thread has its own registers and local memory.
- Threads access global memory when working on large datasets.
- Use constant memory when reading shared constants.

Blocks:

- All threads in a block share shared memory.
- Shared memory is used for fast cooperation and communication among threads (e.g., for reduction, tiling).

Grids:

- All blocks access global memory.
- No direct inter-block communication: communication happens via global memory.

Memory Type	Scope	Latency	Bandwidth	Size	Accessed by	Usage
Registers	Single Thread	Very low	Very high	Few KB/thread	Individual thread	Store thread-local variables
Shared Memory	Thread Block	Low	High	~48 KB/block	All threads in a block	Inter-thread communication and caching
Global Memory	All threads	High	Moderate	GBs	All threads (all blocks)	Main memory for data exchange across grid
Constant Memory	All threads	Low (cached)	High for broadcasts	64 KB	Read-only by all threads	Store constants; fast when all threads access same value
Texture / Surface Memory	All threads	Variable	Cached access	Varies	All threads	Optimized for 2D/3D spatial locality
Local Memory	Single Thread	High	Moderate	Allocated from global	Individual thread	Overflow from registers; private to thread

(d) Briefly explain what happens when branch instructions are executed in GPU code. In particular, how is performance impacted?

Solution: GPUs execute the same instructions on multiple data elements as a set of threads, in lockstep. When the threads in a warp take different paths at a branch, both paths must be explored. The branch condition is evaluated as a predicate and then individual lanes execute no-ops on the not-taken branch path. Specifically, threads that do not take a branch path basically stall while waiting for threads that do take that path to finish it. Since the threads that execute one path of the branch must wait for those that execute the other path of the branch, overall performance is reduced. The performance impact is that the utilization of the GPU lanes is reduced, as only a subset of lanes execute as each branch path is run.

(e) How many warps will there be in the block for a given 16x16 block of threads?

Solution: *16^2 / 32 = 256 / 32 = 8 Warps*

(f) How many of the blocks in (e) can be scheduled on the streaming multiprocessor if it can take up 1536 threads?

Solution: 1536 / 256 = 6 *blocks*

(g) Assuming a maximum of 8 blocks can be scheduled to the streaming multiprocessor in (f), which would more fully occupy the streaming multiprocessor in (f), 8x8 blocks of threads, 16x16 blocks of threads, or 24x24 blocks of threads?

Solution:

8 x 8: 1536 / 64 = 24, 64 * 8 = 512, 1024 of the 1536 threads not utilized.
16 x 16: 1536 / 256 = 6, 256 * 6 = 1536, fully utilized – the best.
24 x 24: 1536 / 576 = 2, 576 * 2 = 384 of the 1536 threads not utilized.