**DSC-204A: Scalable Data Systems, Fall 2025**

# 9: Parallelism

*Lecturer: Hao Zhang* *Scribe: Yu Han, Yifei Chen, Yuchen Xiong, Jingwen Zhang*

## 1 Parallelism

Parallel computing addresses the central issue that **workloads take too long for a single processor** by splitting the workload across multiple processors or workers, following the classic "divide and conquer" principle. In data systems, there are several key parallelism paradigms, including **task parallelism**, **data parallelism**, and **hybrid parallelism**, each corresponding to different ways of organizing **data** and **functions**:

- **Task parallelism** uses *shared or replicated data* while *partitioning the function*. In other words, all workers access the same dataset (or a copy of it), but different workers execute different tasks or parts of the computation.
- **Data parallelism** uses *shared or replicated functions* while *partitioning the data*. This means each worker runs the same task or model but operates on different subsets of the data.
- **Hybrid parallelism** combines both data and function partitioning, allowing different workers to handle different data and different functions at the same time for better scalability and flexibility.

### 1.1 Terminology

In system architecture, different domains use different terminologies: **SIMD** (Single Instruction Multiple Data), **MIMD** (Multiple Instruction Multiple Data), and **SIMT** (Single Instruction Multiple Threads) describe *single-node multi-core execution*, while **distributed systems** focus on *multi-node, multi-core clusters*.

In machine learning, **data parallelism** distributes data across workers running the same model, whereas **model parallelism** partitions the model itself into different components that are processed in parallel. Depending on how the model is partitioned, model parallelism can be further categorized into:

- **Inter-operator parallelism**: Different operators (e.g., layers or modules in a neural network) are assigned to different devices or workers.
- **Intra-operator parallelism**: The computation within a single operator is further partitioned and parallelized across multiple devices or workers.

This distinction allows model parallelism to scale to larger models and make better use of hardware resources.

# 2 Task parallelism

In **task parallelism**, different tasks are assigned to different workers, which can be **cores**, **threads**, or entire **nodes**.

**Advantages:**

- **Conceptual simplicity**: The idea of assigning separate tasks to different workers is straightforward and easy to understand.
- **Low software complexity**: Since workers can operate relatively independently, the overall system design and implementation can be simpler compared to other parallelism strategies.

**Disadvantages:**

- **Difficult to implement**: Tasks must often be *topologically sorted* to respect dependency constraints in the task graph, and efficiently scheduling tasks across heterogeneous workers can be non-trivial.
- **Potential idle time**: Idle time may occur when tasks have uneven execution times or when the number of available tasks is smaller than the number of workers, leading to underutilization of resources.

## 2.1 Degree of parallelism

A central concept in task parallelism is the **degree of parallelism**, which represents the maximum number of tasks that can be executed concurrently at any given point. Increasing the number of workers beyond this degree does not lead to additional performance gains because there are no extra tasks that can be run in parallel. This makes efficient scheduling and workload balancing critical to achieving good parallel performance. **Basic idea.** Split up *tasks* across workers. If tasks read the same dataset, make *copies* of the dataset to each worker (aka *replication*) to avoid read-time contention and remote I/O.

**Setup.** Given three workers (W1, W2, W3) and a task DAG:

$$T1 \to T4 \to T6, \qquad T2 \to T5 \to T6, \qquad T3 \to T6,$$

all reading the same dataset $D$.

**Step-by-step schedule (as in the slide).**

1. **Replicate data:** Copy the whole dataset $D$ to all workers.
2. **Parallel wave 1:** Run $T1$ on W1, $T2$ on W2, $T3$ on W3 (three tasks in parallel).
3. **Parallel wave 2 (respecting dependencies):**
   - After $T1$ finishes, run $T4$ on W1.
   - After $T2$ finishes, run $T5$ on W2.
   - After $T3$ finishes, W3 becomes **idle** (no ready successors for $T3$ except $T6$, which also depends on $T4$ and $T5$).
4. **Final wave:** After $T4$ and $T5$ both finish, run $T6$ (e.g., on W1). During this time, W2 (and W3) are **idle**.

**Why idleness occurs.** Even with three workers, the *degree of parallelism* is not constant across time:

- In the first wave, three independent sources ($T1, T2, T3$) permit full utilization.
- In later waves, the DAG limits concurrency (only $T4$ and $T5$ are ready; $T6$ requires both to finish), so at least one worker must be idle.

This illustrates two common realities of task parallelism: (i) replication enables local reads to keep early waves fast; (ii) dependency structure, not just the number of workers, ultimately bounds achievable parallelism and can create idle periods. **Quantifying the Benefit of Parallelism: Speedup**

## 2.2 Speedup

The **speedup** of a parallel system is defined as:

$$\text{Speedup} = \frac{\text{Completion time with 1 worker}}{\text{Completion time with } n \text{ workers}}.$$

Ideally, speedup increases linearly with the number of workers $n$ (e.g., doubling the number of workers halves the runtime). However, in practice, the achievable speedup is constrained by several key factors:

- The **degree of parallelism**, which limits how many tasks can actually run concurrently at each stage.
- The **task dependency graph structure**, which may force some tasks to wait for others to finish.
- **Intermediate data sizes** and **communication overhead**, which can reduce effective parallel efficiency.

These constraints are well captured by **Amdahl's Law**, which expresses the theoretical maximum speedup as:

$$S(n) = \frac{1}{(1-p) + \frac{p}{n}},$$

where $p$ is the **parallelizable portion** of the workload, and $(1-p)$ is the **serial portion**.

Even if $n \to \infty$ (infinite processors), the speedup is limited by:

$$\lim_{n\to\infty} S(n) = \frac{1}{1-p}.$$

This means that the serial part of the computation imposes a **hard upper bound** on parallel performance. Therefore, simply increasing the number of workers does not guarantee linear speedup.

## 2.3 Weak and Strong Scaling

Scaling behavior can be analyzed in two ways.

**Strong scaling** refers to fixing the total data size and increasing the number of workers, observing how the runtime decreases. Under ideal conditions, strong scaling achieves linear speedup:

$$S(n) = n.$$

**Weak scaling** involves increasing both data size and the number of workers proportionally and observing whether the runtime remains constant.

In reality, **sublinear speedup** is common because of synchronization costs, data transfer overhead, and serial computation bottlenecks.

An interesting phenomenon is **superlinear speedup or scaleup**, where the observed speedup exceeds the number of workers. This can occur when adding more workers improves memory locality, reduces cache misses, or decreases overhead in specific tasks such as language model inference. For example, using 4 workers may result in a 5× or 6× speedup, exceeding the linear expectation.

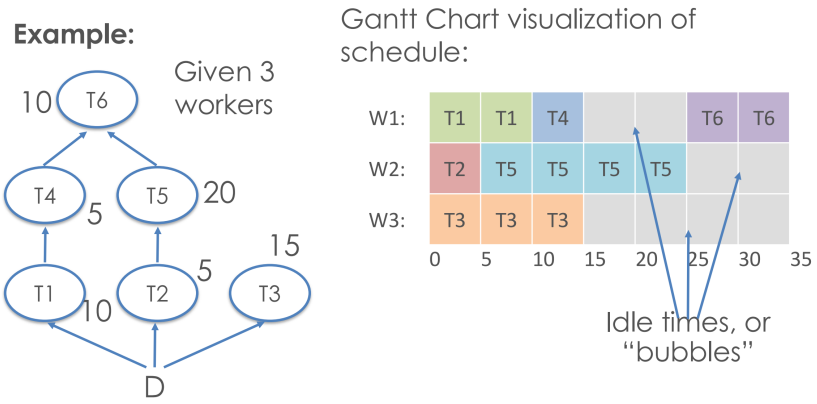
## 2.4 Idle Time in Task Parallelism

In any parallel setup, you first want to quantify what causes **idle time**, periods when some workers sit idle while others are still busy. Reducing idle time improves overall utilization, leading to **lower costs** and **higher efficiency**.

Idle time appears because task completion times differ and workloads exhibit varying degrees of parallelism. These unused periods are often visualized in **Gantt charts** as gray segments called **bubbles**.

To study them, we assign six tasks (T1–T6) to three workers (W1–W3):

- Each task has its own processing time (not all equal).
- The **x-axis** represents time; the **y-axis** represents workers.




Assigning a task to a worker is known as the **placement problem**.

- W1: Executes T1 (10 units) → unblocks T4 (5 units). T6 must wait until T5 finishes.
- W2: Runs T2 (5 units) → unblocks T5 (20 units).
- W3: Handles T3 (3 units) → then mostly idle.

*Bubble = idle time between tasks.*

Our goal is to design better **placement** and **partition** schemes to minimize bubbles. In machine-learning systems, bubble minimization can even be formalized as an optimization problem.

### 2.5 The Critical Path

In general, total completion time for a task-parallel workload is **lower-bounded** by the **longest path** in its task graph, which is the **critical path**.

- Optimizing the critical path removes bottlenecks.
- In debugging distributed systems, always identify and shorten the critical path to cut runtime.

**Example:**

- Sequential completion = 65 units
- Parallel (3 workers) = critical-path length = $5 + 20 + 10 = 35$ units
- Speedup = $\frac{65}{35} \approx 1.9\times$

This sublinear speedup reflects idle bubbles and dependency limits.

## 3 Data Parallelism

Modern ML frameworks (Transformers, GPT) parallelize entire **data-flow graphs** across GPUs. Each worker holds:

- An identical copy of model weights
- A distinct partition of data (D1, D2, D3)

During training:

1. Each worker performs forward + backward passes locally.
2. Gradients are synchronized via all-reduce.
3. The next iteration begins.

In simpler, non-ML systems (e.g., Spark, Ray, Hadoop), each worker processes its data shard independently and writes results to disk for later aggregation and no synchronization needed.

**Speedup formula remains:**

$$\text{Speedup} = \frac{\text{Completion time (1 core)}}{\text{Completion time (n cores)}}$$

### 3.1 Amdahl's Law

Even with unlimited cores, speedup is capped by the serial portion of the program.

Let

- $T_{\text{yes}}$: parallelizable portion
- $T_{\text{no}}$: serial portion
- $f = T_{\text{yes}}/T_{\text{no}}$

Then:

$$\text{Speedup} = \frac{n(1+f)}{n+f}$$

**Implications:**

- Even 95% of code parallelizable $\Rightarrow$ speedup $\leq 20\times$
- Beyond that, adding cores yields diminishing returns.
- Helps estimate realistic limits before allocating more workers.

Efficient parallel systems aren't just about adding more workers. They depend on how tasks are divided, how data is partitioned, and how dependencies are handled. Idle bubbles waste compute; critical paths define limits. Even perfect code hits the ceiling set by Amdahl's Law, so understanding these fundamentals is key to designing scalable ML and distributed systems.

# 4 Hardware Parallelization

## 4.1 Built-in Data Parallelism in Modern Processors

At the lowest level of a computer, parallel processing is already built into the CPU. Single-Instruction Multiple-Data (SIMD) is a norm in processor community. The programs your operating system submits to the chip are actually executed in parallel. Adding more ALUs is the main theme of chip design in the past 30 years.

| **Notion** | **Full Name** | **Description** |
|---|---|---|
| **SIMD** | Single Instruction, Multiple Data | One instruction operates on multiple data elements simultaneously (vectorized execution). |
| **SIMT** | Single Instruction, Multiple Threads | Extends SIMD to multiple **threads**. Each thread may be assigned a core or processing unit. |
| **SPMD** | Single Program, Multiple Data | Higher-level abstraction generalizing SIMD; each thread/process runs the same program on different data chunks. |

Table 1: Comparison of SIMD, SIMT, and SPMD Parallel Processing Notations

## 4.2 The Post–Moore's Law Era

But we are approaching the limits of Moore's Law. Apple is already producing chips using a 3 nm process, and if we try to make ALUs even smaller, we'll enter the quantum realm—where an entirely new computer

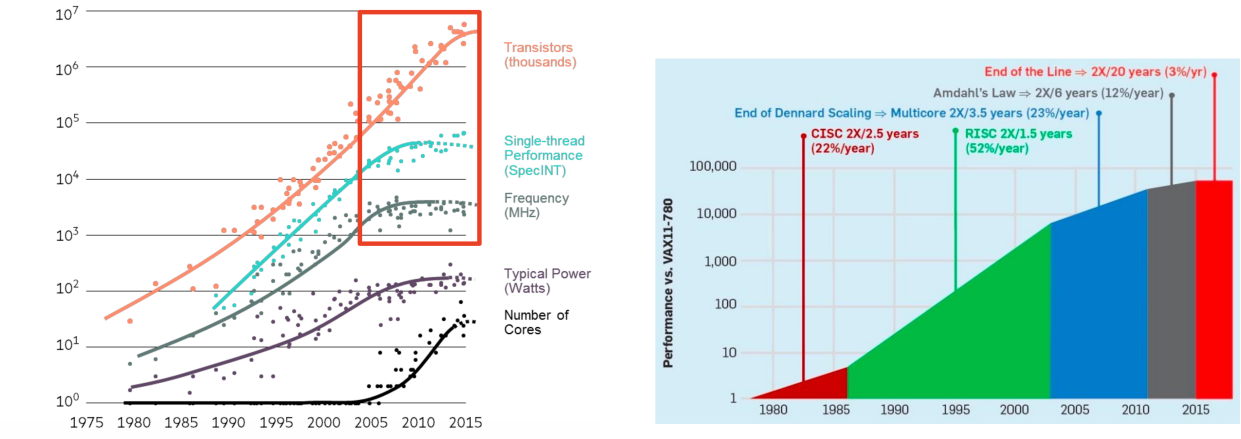



Figure 1: we are approaching the limits of Moore's Law

architecture will be required. Moreover, we can't simply add more ALUs either, since power consumption and heat dissipation impose physical constraints on computational power.

## 4.3 Specialized Hardware for Enhanced Computational Power

### 4.3.1 GPU

The space for cache is difficult to squeeze. But we can reduce the space for control units. This makes the chip less universal but more specialized and powerful for specific types of computations. This is basically the design philosophy behind GPUs. As shown in Figure 2, the control unit and L1 cache are shared among a group of ALUs, thereby increasing the degree of data parallelism. This design enables GPUs to achieve significant acceleration in matrix computations. In the era of image processing, 3D rendering, and machine learning, most computations are highly parallelizable and simple, making this architecture especially effective.

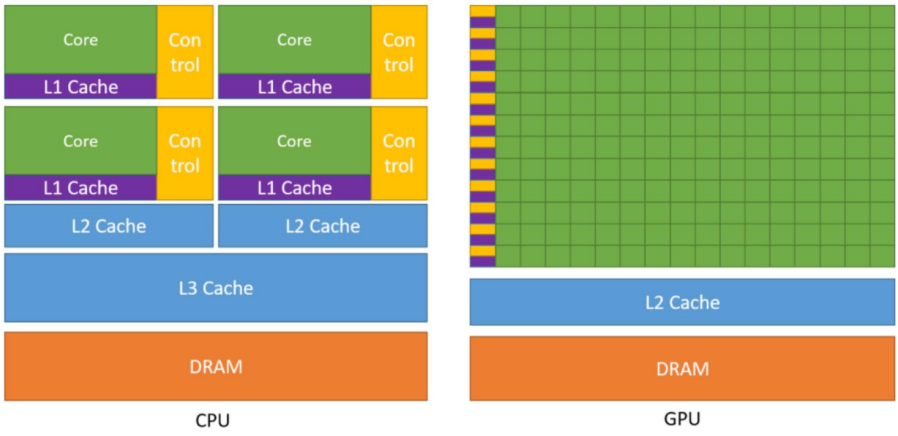



Figure 2: Comparison of CPU and GPU

### 4.3.2 NVIDIA and CUDA

The design of GPU was popularized by NVIDIA in early 2000s for video games, graphics, and multimedia, and now ubiquitous in Deep Learning.

NVIDIA released CUDA in 2007, followed by a series of higher-level libraries and APIs such as cuDNN (for CNN), cuSPARSE (for sparse tensors), cuDF (RAPIDS AI), NCCL (for communications), CUTE.

### 4.3.3 TPU

The design of the Tensor Processing Unit (TPU) follows a similar rationale to that of the GPU. For example, Google removes most of the control logic and focuses on building tensor cores that are optimized exclusively for tensor computations, as these are the primary workloads in deep learning.

The development trajectory of processing units has evolved from first minimizing the size of Arithmetic Logic Units (ALUs), to then reducing the area occupied by control units. This leads to the creation of increasingly specialized processing units that target specific workloads, thereby achieving higher computational efficiency.

There are several ways to specialize a processing unit:

- Designing specialized ALUs for tensor computations;
- Combining heterogeneous specialized cores;
- Reducing the precision of floating-point numbers;
- Optimizing the balance between memory capacity, bandwidth, and processing power.

Compared with other types of processors, GPUs offer superior computational throughput but are generally less power-efficient.

## 4.4 Case Study

### 4.4.1 Nvidia GPU Specifications

To achieve higher computational performance, some Nvidia GPUs are specialized for handling low-precision floating-point operations, while others are optimized for matrix computations with the so-called “224” sparsity pattern. As illustrated below, such matrices contain two non-zero and two zero values within each orange block. Although this kind of sparsity is difficult to obtain naturally in machine learning, the robustness of modern ML models allows for reduced numerical precision in exchange for substantially higher computational throughput.

### 4.4.2 Apple Silicon

Apple’s M-series chips are also highly specialized. They integrate both CPU ALUs and GPU ALUs onto a single die. CPU ALUs are designed for general-purpose computations, whereas GPU ALUs are optimized for tensor-based workloads. By sharing the same power and memory infrastructure between these two domains, Apple effectively balances graphical and general-purpose processing performance. The M-series chips are actually powerful enough to run models such as LLaMA on a single laptop.

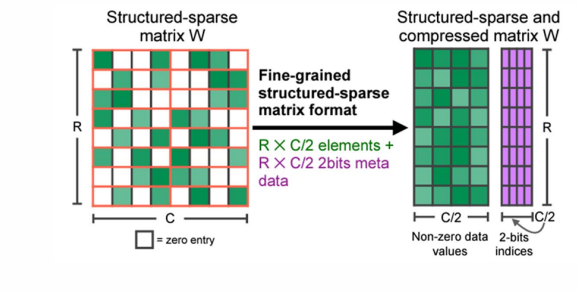



Figure 3: 224 sparsity

Since Apple designs both the hardware and the operating system, it can perform deep co-optimization between the two. This allows the chip to intelligently direct operating system tasks to the most suitable cores for execution, thereby maximizing performance while minimizing power consumption.

### 4.4.3 Leading Chip Startups

Due to the increasing demand from deep learning (DL) and machine learning (ML) workloads, many startups are developing chips that are even more specialized than GPUs, achieving superior performance in certain highly specific applications. Three startups are particularly noteworthy:

- **groq:** Develops chips that are claimed to be up to ten times faster than Nvidia GPUs in large language model (LLM) inference.
- **cerebras:** Produces chips specifically designed for machine learning workloads.
- **SambaNova:** Innovates not only in chip design but also in the system-level integration of hardware and software for enterprise customers such as banks, with a strong focus on machine learning applications.