DSC-204A: Scalable Data Systems, Winter 2024

Parallelism Basics - 2

Lecturer: Hao Zhang *Scribe: Zixi Yuan, Jiacheng Qiu, Xujun Lian*

Recap

1 Weak and Strong Scaling

1.1 Speedup

Speedup is a way to measure the impact of the parallelism.

1.2 Strong Scaling

Strong Scaling refers to the efficiency of solving a fixed total program size or workload size with increasing numbers of workers.

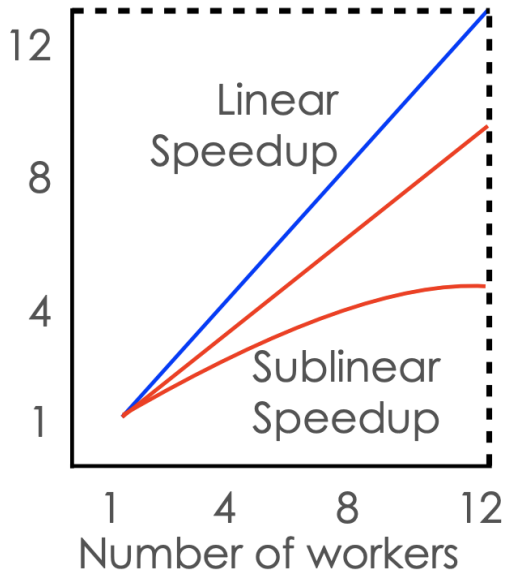

Figure 1: Strong Scaling

1.3 Weak Scaling

Weak scaling refers to the time to complete the task for all workers when we fix the workload or program size for each worker with an increasing workers.

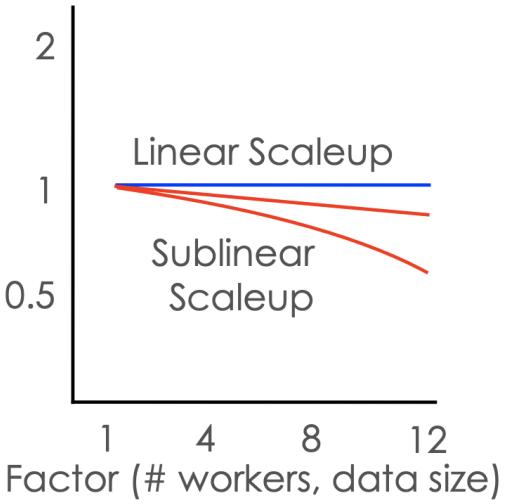

Figure 2: weak Scaling

1.4 Superlinear Speedup Possibility

Q: Is superlinear speedup ever possible?

- For strong scaling–yes: For example, when the first worker reads data from disk, it can create a cache in the computer, thus other workers will read data faster. Also, when adding more workers, the memory bandwidth and disk bandwidth increase a lot. These factors can also make the system go faster than linear.
- For weak scaling—yes: For example, when the workload becomes larger, the chip processes faster (case of GPU, that's why people usually train a large model rather than small to avoid underutilizing).

2 Some Clarification on Terms

2.1 Speed Up formula

- Speed up = Completion time given only 1 worker / Competition time given $n(> 1)$ workers

2.2 Different Terms with the Same Meaning

- Speedup, acceleration - strong scaling

- Scaling, scale-up - weak scaling
- Scalability - both
- "System A is very scalable"- When you add one more worker, the speedup increases by 1.
- "System A is more scalable than system B" - When you add one more worker, the speedup of system A is larger than that of system B.

3 IDLE Times in Task Parallelism

3.1 Due to varying task completion times and varying degrees of parallelism in workload, idle workers waste resources

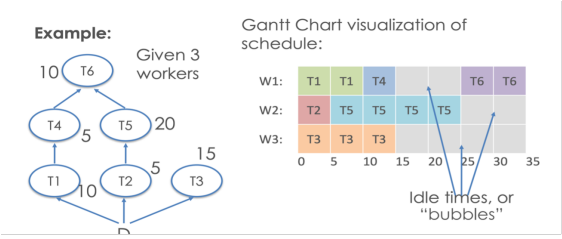

Figure 3: IDLE Times in Task Parallelism

3.2 Workload Completion Time and Task-Parallel Setup

- In general, the overall workload's completion time on task-parallel setup is always lower bounded by the longest path in the task graph.
- Possibility: A task-parallel scheduler can "release" a worker if it knows that will be idle till the end.
- Can save costs in the cloud.

3.3 Completion Time and Speedup

- As we can see in the picture above, Completion time with 1 worker = 65 (add all).
- Parallel completion time = longest path = $10 + 20 + 5 = 35$.
- Speedup = $\frac{65}{35}$.

New Content

4 Recall: Data Parallelism in ML

Explain: As we see in Figure 4 below, What we have are: 4 workers, training data. What we do: Partition data into 4 partitions and assign each partition to a worker, then perform gradient descent. That is the way

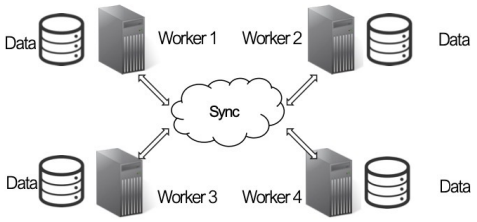

Figure 4: ML data parallelism

to train a model for today. Each worker processes a subset of the data independently. After processing, the workers synchronize their computations to update a shared model.

The model parameters are updated according to the following rule:

$$\theta^{(t+1)} = \theta^{(t)} + \varepsilon \sum_{p=1}^{P} \nabla \mathcal{L}(\theta^{(t)}, D_p^{(t)})$$

4.1 Data Parallelism Abstraction

Data parallelism: abstraction of SIMD/SIMT/SPMD.

4.1.1 Representation of Data Parallelism in Dataflow Graphs

How to represent data parallelism in dataflow graph notions? Copy graphs many times but partition the data. (P.S For task parallelism, we partition the graph not replicate graph, that's the difference)

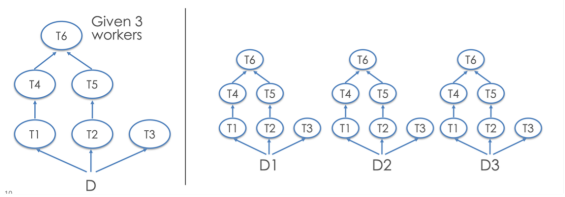

Figure 5: ML data parallelism

4.2 Built-in Data Parallelism in Modern Processors

Data parallelism is built in with today's Processors. Modern computers often have multiple processors and multiple cores per processor, with a hierarchy of shared caches. How it works: When a job is submitted to a processor, it will process in a data parallelism way. It will divide data into 4 partitions and then align each partition to each core.

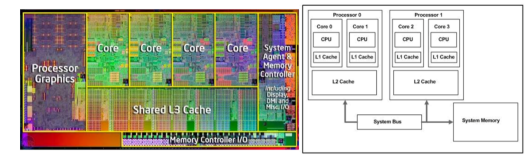

Figure 6: ML data parallelism

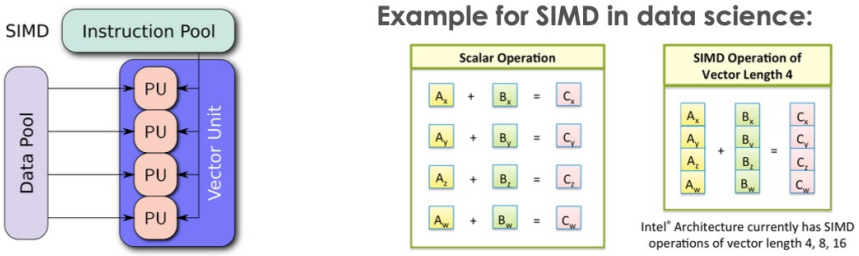

Figure 7: SIMD and Application

4.3 Single-Instruction Multiple-Data

The left side of the image illustrates the SIMD architecture, where a single instruction pool dispatches the same instruction to multiple processing units (PUs). Each PU operates on different data elements in parallel, which is effective for vectorized operations where the same operation is to be performed on multiple data points simultaneously.

The right side of the image compares a scalar operation to a SIMD operation. In the scalar operation, each computation is performed sequentially. In contrast, the SIMD operation demonstrates how four pairs of elements are added in parallel at once.

4.4 SIMD Generalization

- Single-Instruction Multiple Thread (SIMT): Generalizes notion of SIMD to different threads concurrently doing so.
  - Each thread may be assigned a core or a whole PU
- Single-Program Multiple Data (SPMD): A higher level of abstraction generalizing SIMD operations or programs
  - Under the hood, may use multiple processes or threads
  - Each chunk of data processed by one core/PU
  - Applicable to any CPU, not just vectorized PUs
  - Most common form of parallel data processing at scale

4.5 Quantifying Efficiency of Data Parallelism

4.5.1 speedup

Like we did with task parallelism, we can measure the speedup:

$$\text{Speedup} = \frac{\text{Completion time given only 1 core}}{\text{Completion time given n (>1) core}}$$

Figure 8: Measure of speedup

4.5.2 Amdahl's Law

Given n cores, we may or may not get a speedup of n. (Just like it did with task parallelism) In data parallelism, it's easier to analyse the speedup.

Amdahl's Law: Formula to upper bound possible speedup

- A program has 2 parts: one that benefits from multi-core parallelism and one that does not.
- Non-parallel part could be for control, memory stalls, etc.

1 core: T_{yes} T_{no} — n cores: T_{yes}/n T_{no}

$$\text{Speedup} = \frac{T_{yes} + T_{no}}{T_{yes}/n + T_{no}} = \frac{n(1+f)}{n+f}$$

Denote $T_{yes}/T_{no} = f$

Figure 9: Amdahl's Law

Using the function, we can approximately estimate the best speedup we can achieve.

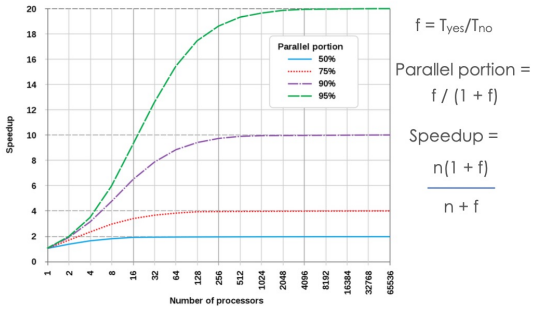

Figure 10: Amdahl's Law: processors and speedup

This graph shows that as the number of processors increases, the speedup for tasks with a larger parallel portion approaches a higher maximum speedup. However, for all curves, the speedup gain diminishes as

more processors are added, illustrating the diminishing returns due to the serial portion of the task. The flatter the curve becomes as the number of processors increases, the less benefit we get from adding more processors.

4.6 The Problem of Chip Design

4.6.1 Chip Design

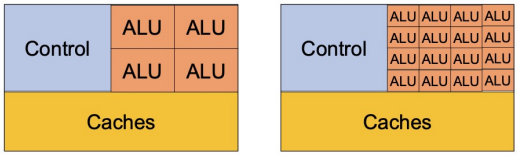

Figure 11: Chip Design

On the left side, there is a chip design and it represents a typical chip layout where the control unit decodes the instructions and the ALUs perform arithmetic and logical operations. The caches are used to speed up the access to data and instructions by storing them closer to the ALUs. On the right side, the diagram shows a hypothetical scenario where, through advancements in technology, the size of the ALUs has been reduced while maintaining their power. This reduction in size has allowed for more ALUs to be placed in the same area, potentially increasing the chip's computational power as there are more units to perform operations in parallel.

Besides, the smallest size is 3 nm (designed by Apple) now.

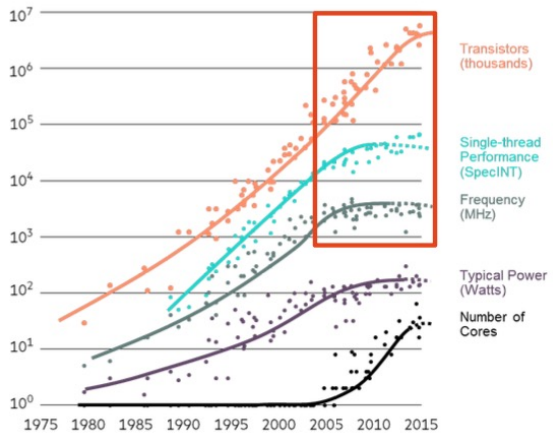

Figure 12: Scaling Limits and Efficiency Challenges

The graph depicts the evolution of technologies from 1975 to around 2015, highlighting key metrics that illustrate the advancement in computing power over time.

Quantum computing is the new idea, which does not use physical cores. In this way, we can put almost infinite quantum in the area where we can only put a few ALUs originally.

4.6.2 Hardware Accelerators: GPUs

Idea: How about we use a lot of weak/specialized cores

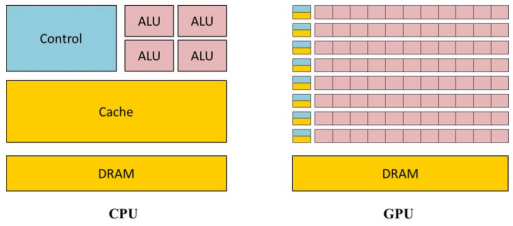

Figure 13: GPU

These ALUs are samller, weaker and specialized.

Graphics Processing Unit (GPU): Tailored for matrix/tensor ops.

- Basic idea: Use tons of ALUs (but weak and more specialized); massive data parallelism (SIMD on steroids); now H100 offers 980 TFLOPS for FP16!
- Popularized by NVIDIA in early 2000s for video games, graphics, and multimedia; now ubiquitous in DL.
- CUDA released in 2007; later wrapper APIs on top: CuDNN, CuSparse, CuDF (RapidsAI), NCCL, etc.

4.6.3 Other Hardware Accelerators

Figure 14: TPU

Figure 15: FPGA

Tensor Processing Unit (TPU): An "application-specific integrated circuit" (ASIC) created by Google in mid 2010s; used for AlphaGo.

Field-Programmable Gate Array (FPGA): Configurable for any class of programs; 0.5-3 TFLOPS but very low power consumption. Cheaper; new hardware-software integrated stacks for ML/DL.

4.7 Comparing Modern Parallel Hardware

	Multi-core CPU	GPU	FPGA	ASICs (e.g., TPUs)
Peak FLOPS	Moderate	High	High	Very High
Power Consumption	High	Very High	Very Low	Low-Very Low
Cost	Low	High	Very High	Highest
Generality / Flexibility	Highest	Medium	Very High	Lowest
Fitness for DL Training?	Poor Fit	Best Fit	Poor Fit	Potential exists but not mass market
Fitness for DL Inference?	Moderate	Moderate	Good Fit	Best Fit
Cloud Vendor Support	All	All	All	GCP

Figure 16: comparison

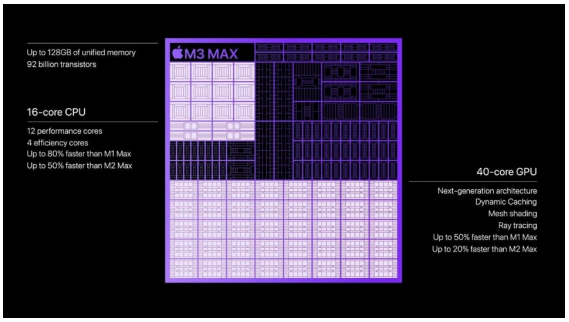

Figure 17: Apple M3

Take the Apple M3 chip as an example, it uses different strategies from other companies (e.g. Intel). The CPU has 16 cores, divided into 12 performance cores for demanding tasks and 4 efficiency cores for less intensive tasks, which helps balance performance with power consumption.

The companies involved in designing chips tailored for machine learning models are experiencing a significant boom, with their value soaring at a rapid pace. As the demand for sophisticated artificial intelligence (AI) and machine learning (ML) applications continues to escalate across various industries, these companies are at the forefront of innovation, developing specialized hardware optimized for the computational requirements of deep learning algorithms. With advancements in neural network architectures and the increasing complexity of AI tasks, the need for efficient, high-performance chips has become paramount. Consequently, companies specializing in chip design for deep learning are witnessing substantial growth and appreciation in their market value.

4.8 Multi-node Distributed Systems

Multi-node distributed systems refer to complex computing architectures with multiple interconnected nodes collaborating to perform a task. Unlike traditional single-node systems, multi-node distributed systems distribute memory, storage, computing, and networking resources across several interconnected nodes or machines.

4.8.1 Paradigms of Multi-Node Parallelism Implementations

Depending on what kind of resources need to be shared, there are three ways of implementing parallelisms in distributed systems.

- **Shared-memory parallelism (vertical scaling):** Multiple computing units (like CPUs) and storage disks connect to a shared pool of memory within the distributed system. It behaves like a single but super powerful computer with a large memory. It has the advantage of simple implementation and high performance. But at the same time facing the shortcomings of super-linearly growing costs with sub-linearly growing performance. For instance, the cost of 2TB memory is more than 2 times higher (super-linearly) than the 1TB memory, but the performance will not double (sub-linearly) because of the difficulty of scaling up. Also, the geolocation is restricted as all the system parts have to be physically close enough to each other to connect without a network.

- **Shared-Disk Parallelism (data warehouse):** Data is stored in an array of disks, and disks are virtualized into systems like Network File System (NFS). The computing units and memory units (RAM) are independent, but they have access to the shared Network File System. It has the advantage of low cost as disks are relatively cheap computer components and only I/O-related communication with disks is needed. However, when multiple clients are accessing the same piece of data at the same, the system will have a transaction problem as it needs to coordinate the order.
- **Shared-Nothing Parallelism (horizontal scaling):** This will be introduced in the next lecture.

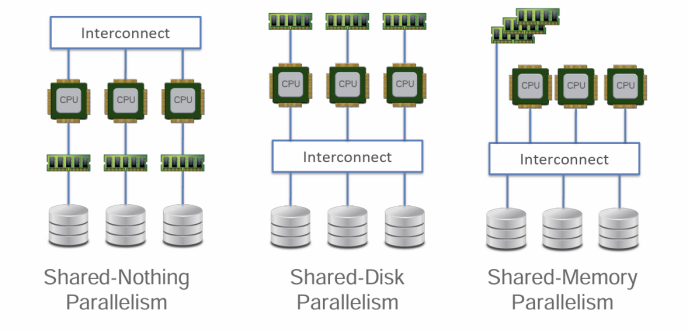

Figure 18: Three Paradigms of Multi-Node Parallelism Implementations