SAPIENZA
UNIVERSITÀ DI ROMA

FACULTY OF INFORMATION ENGINEERING,
INFORMATICS, AND STATISTICS

Big Data Computing

DEPARTMENT OF COMPUTER SCIENCE

Professors:
Daniele De Sensi

Student:
Davide Pietragalla

Academic Year 2024/2025

Contents

1 Introduction

The term **Big Data** refers to the management and analysis of enormous amounts of data. They are commonly evaluated across five key dimensions known as the 5 V's:

- **Value**: extracting knowledge from data is extremely valuable
- **Volume**: very large amount of data (orders of TB or PB)
- **Variety**: different formats of structured (relational tables), semistructured (JSON files), and unstructured (text/audio/video) data
- **Velocity**: insane speed at which data is generated (e.g., Twitter stream)
- **Veracity**: reliability of the data used to drive decision processes

1.1 (Toy) Example – Google Search

The Google Search mechanism uses data structures called **Inverted Indexes**, where it stores which terms point to which documents. Assuming there are 130 trillion pages, representing a document ID requires $\log_2(130 \times 10^{12}) \approx 47\,\text{bits} \approx 8\,\text{bytes}$. Therefore, to store all the IDs, about 1 petabyte would be needed, not counting that IDs can appear multiple times and that there is also a need to store the terms.

Google handles approximately 5.9 million searches per minute, which means it has about 10.16 microseconds to serve each request. In order to make this possible, it must address the following challenges.

- Disks are not large enough
- Disks are not fast enough
- CPUs are not fast enough

1.2 Data Centers

In response to the previously described problems, data centers can adopt two scaling options:

- **Scale up** means upgrading system components with faster disks and more powerful CPUs; however, it requires time, money, and does not keep pace with the growing volume and velocity of data.
- **Scale out** (better) means purchasing more servers and disks to make them work in parallel, which is good because components can be added as the load increases; however, it introduces challenges related to reliability and parallel work.

To train a model like ChatGPT, it is estimated that more than 200,000 GPUs are needed; the cost of each GPU is approximately 30,000 USD, so the total cost would be around 6 billion USD. Additionally, in terms of energy, the monthly cost for around 10,000 GPUs would be 1 million USD per month, which also has an **environmental impact**.

1.3 The Network Bottleneck

In a network, if the nodes need to communicate more than they compute, the system slows down, creating a **bottleneck**. This also happens because in recent years GPUs have improved much faster than network bandwidth; in other words, even with very powerful GPUs, if the network is slow or saturated, the entire system slows down as it cannot move data quickly enough.

1.4 Other Challenges

Parallel work, as already mentioned, introduces problems of both reliability and programmability. **Reliability** refers to a server's ability to handle failures, which is important because if you are training on 3,000 GPUs at a cost of around 20,000 USD per hour, a single failure (without a checkpoint) can cost approximately 30,000 USD. **Programmability** involves challenges related to storing, communication between servers, failure management, and computational distribution between CPU and GPU.

2 Distributed Deep Learning

Deep Learning is a branch of machine learning that uses artificial neural networks with many layers (hence the term *deep*) to learn complex representations from data. These networks are particularly effective at processing large amounts of unstructured data, such as images, text, and audio, and their performance improves as the amount of data, parameters, and compute increases.

2.1 DNN Training

To train a **Deep Neural Network**, multiple iterations over three phases are executed:

1. In the **Forward Propagation** step, a batch of input is passed through the model to obtain a prediction (compute the function).

2. In the **Backward Propagation** step, the model is run in reverse to produce an error value for each trainable parameter, helping to identify which parameters contributed the most to the error (compute the gradient).

3. Finally, there is the **Parameter Update** phase, where the loss values are used to update the model parameters (use the gradient).

2.2 DDL Forms of Parallelism

Distributed Deep Learning is a technique that speeds up deep neural network training by distributing the workload across multiple machines or processors.

This distribution can occur through three forms of parallelism:

- **Data Parallelism**
- **Pipeline Parallelism**
- **Operator Parallelism** (or tensor parallelism)

In general, these three approaches have advantages and disadvantages depending on the context; however, in the case of very large models, all three are typically applied together.

The simplest approach is **Data Parallelism**, which applies a partitioning of the training set into n parts, on which a loop is executed to reduce the loss function:

1. for each partition, compute the forward pass on an independent copy of the same model

2. compute the loss function for each resulting prediction

3. compute the gradient for each model independently

4. apply **Gradients Aggregation**, which consists of averaging all the gradients (in this way, the independent models remain in the same state step by step)

5. use the aggregated gradient to update the model

where the steps on different partitions occur in parallel. Many studies claim that gradients aggregation contributes to saving 20%, 30%, or even 50% of training time.

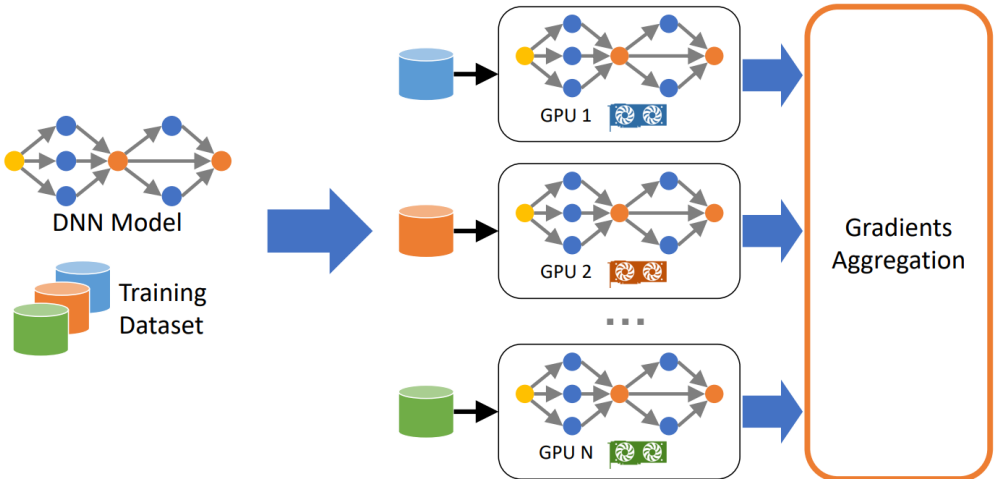

Figure 1: Data parallelism representation.

To enable communication between GPUs and update the parameters, an old approach is to use one or more **Parameter Servers**. The PS architecture follows a centralized communication model involving three main steps:

1. each worker **pushes** its gradients to the PS

2. the PS **aggregates** all the gradients

3. workers then **pull** the updated model parameters

this type of architecture often leads to the creation of a bottleneck, which is critical since, as seen earlier, the training process is very expensive. Specifically, if the number of PS is less than the number of workers, a bottleneck will occur.

Cost Model:

- let n be the number of bytes in the gradient vector
- let β be the network bandwidth per node
- let p be the number of workers and K the number of PS
- total communication time:

$$\max\left(\frac{n}{\beta}, \frac{pn}{K\beta}\right)$$

There are also cases where the model is too large to fit on a single GPU; in such cases, the idea is to split the computation into layers across multiple GPUs (called **Model Parallelism**); as a result, the first GPU computes its layers and passes the result to the second, and so on. The main issue with this approach is that GPUs spend a significant amount of time waiting to receive input, which, as we have already seen, is costly and inefficient. In this situation, **Pipeline Parallelism** comes into play. Instead of computing a large chunk of data and then communicating it, the input is split into smaller micro-batches. Each GPU starts processing and forwarding these micro-batches as soon as they are ready, without waiting for the entire batch to finish. This significantly reduces idle time and improves overall efficiency.

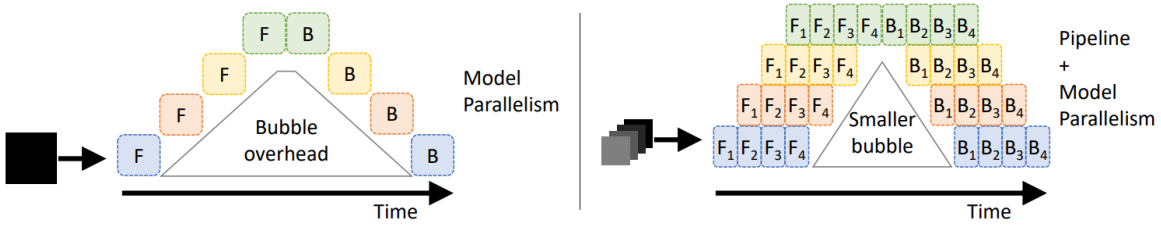

Figure 2: Model and pipeline parallelism compared.

Finally, **Operator Parallelism** leverages the idea of parallelizing each individual operation across multiple GPUs; for example, a matrix multiplication would be distributed among several GPUs. Nowadays, all three forms of parallelism are typically used in combination to efficiently train large-scale models.

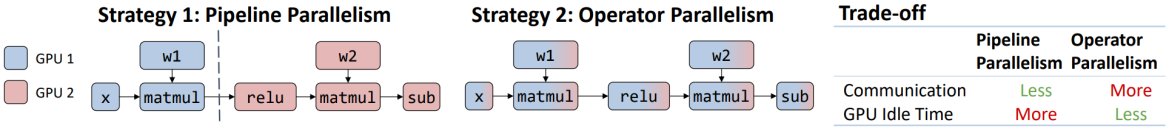

Figure 3: Pipeline and operator parallelism compared.

	Data Parallelism	Pipeline Parallelism	Operator Parallelism
Pros	✓ Simple and efficient ✓ Easy to implement ✓ No communication during forward/backward passes	✓ Layer/parameters can be distributed across GPUs ✓ Only communicate activations/gradients at layers where model is split	✓ Parameters can be distributed across GPUs ✓ No issues with bubbles
Cons	✗ Scalability limited by minibatch size ✗ Model must fit on a GPU ✗ Model replicated on all GPUs	✗ Scalability limited by number of layers ✗ Reducing the impact of bubbles can be tricky ✗ Minibatch replicated on all the GPUs ✗ User needs to decide how to partition the model	✗ Scalability limited by operator/layer size ✗ Minibatch replicated on all the GPUs ✗ Forward/backward passes extremely communication intensive

Figure 4: Comparing forms of parallelism.

2.3 Challenges of Scaling

Let's begin by considering data parallelism. The procedure described earlier involves performing gradient aggregation before updating the model. However, in practice, what actually happens is that the gradient aggregation and communication are performed layer by layer. This approach called **Computation-Communication Overlap** allows for overlapping computation and communication, improving efficiency in terms of time.

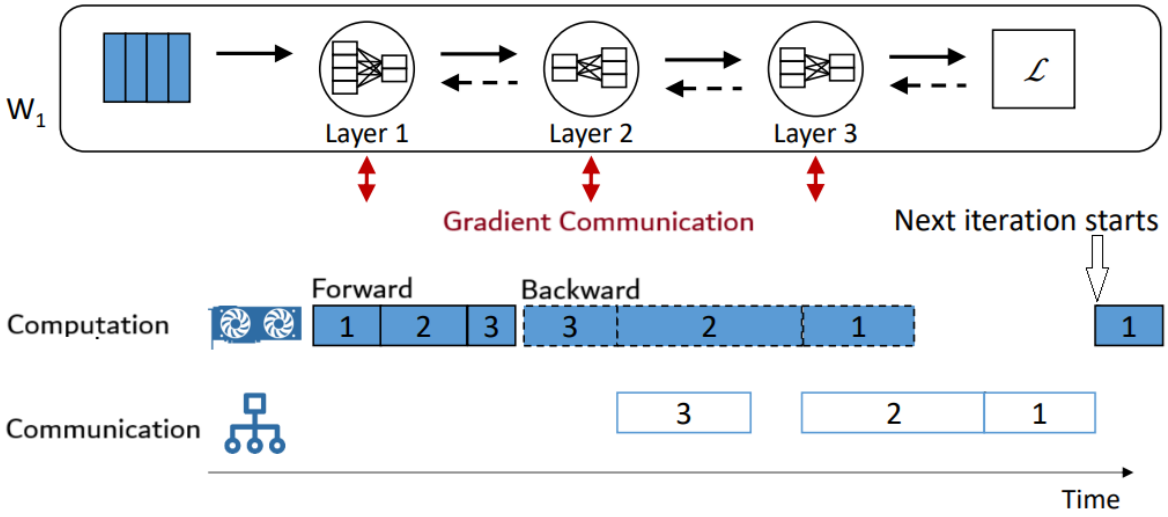

Figure 5: Computation-communication overlap.

A further issue with data parallelism is that gradient aggregation acts as a barrier in terms of time. If one GPU is slower than the others, it will delay the entire process. In such cases, it is possible to aggregate gradients after s iterations to reduce synchronization overhead, or alternatively, to ignore the gradients from the slower GPUs.

2.4 Gradient Compression

The **Gradient Compression** approach is based on the idea that we can reduce the volume of data by approximating its values. This works because, since the model is in the process of learning, it will eventually converge anyway. There are three main techniques for gradient compression: quantization, low-rank approximation, and sparsification.

Quantization consists in approximating the gradient values. For example, if a gradient is originally represented using 32-bit floating point, it can be compressed and sent using only 16 bits, effectively halving the communication cost. There are even models capable of compressing 32-bit gradients into just 1 bit while still working.

Low-Rank Approximation, on the other hand, leverages the idea that the gradient can be decomposed into the product of lower-rank matrices, similar to how kernels

are used in image filtering.

Finally, the idea behind **Sparsification** is that if you want to send an array of 32-bit values, you still send values at 32 bits, but not all of them. In fact, some elements in the array are skipped, with the assumption that this won't significantly harm convergence. For example, you might choose to send only the elements with the largest magnitudes, as they are the ones that have the greatest impact on the model.

Of course, although these techniques are theoretically well-founded, applying them requires careful **balancing**, as excessive approximation often leads to the need for more training iterations, and in some cases even more time, compared to not applying them at all.

2.5 Distributed Gradient Aggregation

As discussed earlier, the process of gradient aggregation has its own cost, and one approach to handling it is through the use of parameter servers. Moreover, the problem of combining data from different nodes and then distributing the aggregated result back to them is generally referred to as **AllReduce**.

Let n be the number of bytes in a gradient vector, p the number of workers (GPUs), and β the network bandwidth of a node. Using the **Parameter Server** approach, the **Communication Volume** is given by $\max(n/\beta, (pn)/K)$ where K is the number of available parameter servers.

An additional method, called **Naive AllReduce**, is one in which each worker communicates its gradient vector to all other workers, and then each of them performs the computations independently. In most cases, this approach performs worse than the previous one. In general, the communication volume is $(p-1)n$.

There is also the idea of splitting the AllReduce operation into two parts:

- **Reduce-scatter**, where each GPU computes a slice of the sum vector,
- **Allgather**, where the slices are distributed so that all GPUs end up with the full sum of the gradient vectors.

The **Ring AllReduce** approach is based exactly on this idea. Starting from n GPUs, it implements a ring topology where each GPU communicates only with its next neighbor. In the first phase, each worker sends one slice (n/p parameters) to the next worker in the ring, and does this for $p-1$ iterations. In the second phase, each worker sends one slice of the aggregated parameters to the next worker, and by repeating this for $p-1$ iterations, each worker ends up with the fully aggregated gradient vector. The Ring AllReduce approach has a communication volume around $2n$, which is better than the previous cases.

It is important to note that communication volume is not the only relevant factor; the number of steps also plays a significant role. For example, in the case of Ring

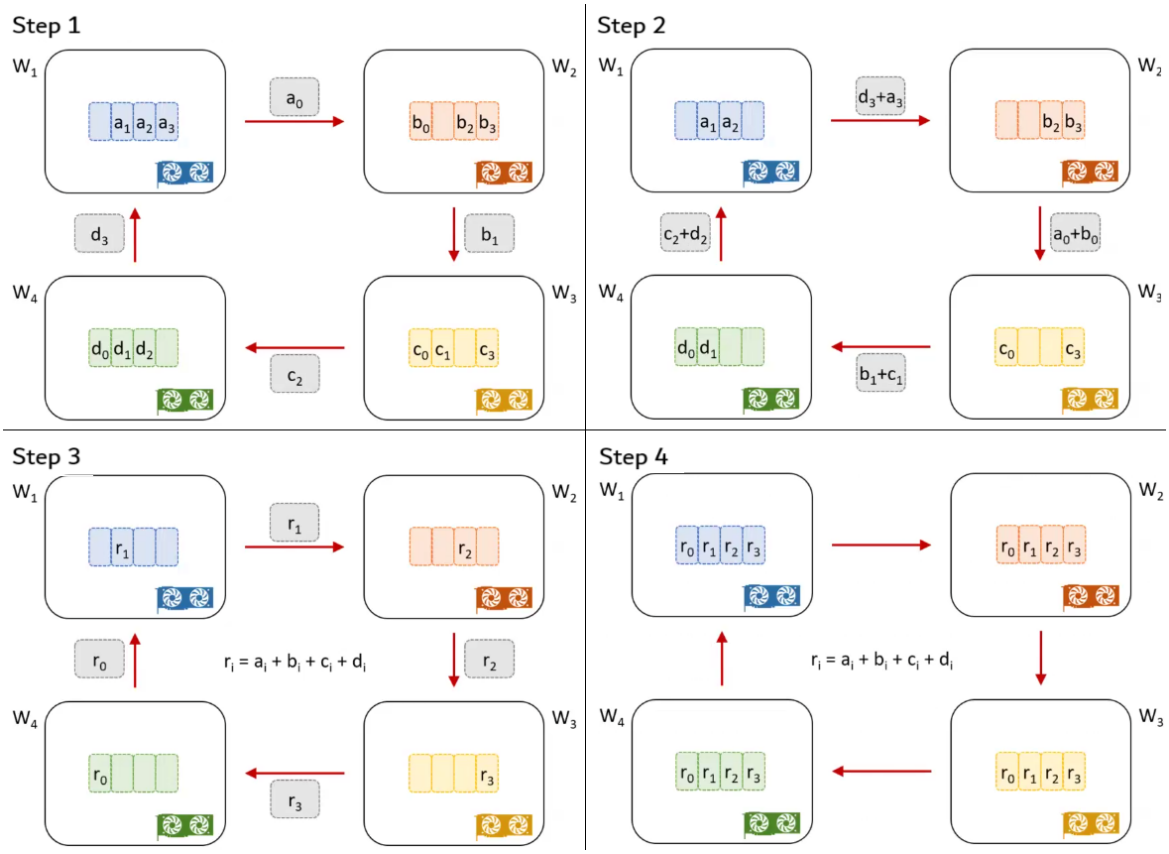

Figure 6: Example of Ring AllReduce approach with 4 GPUs.

AllReduce, the number of steps is $2(p-1)$. The number of steps matters because each communication incurs fixed costs, regardless of the message size. You can see the actual **communication cost model** $(T(n,p) = Cost)$ under the table in figure 7.

Algorithm	Comm. Volume	# Steps
Optimal	N	1
Parameter Server	max(n, (pn)/(K))	1
Naive Allreduce	((p-1)n)	1
Ring	$2n$	2(p-1)

Cost = (Comm. Volume) / β + (# Steps)*α

Figure 7: Gradient aggregation comparison.

There also exists an alternative method for performing reduce-scatter. **Bandwidth Optimal Recursive Doubling**, which aims to achieve a logarithmic number of steps with respect to the number of nodes, rather than a linear one. We start with a number of GPUs, referred to as hosts in this case, each of which has a gradient vector that will be divided into as many blocks as the amount of GPUs. Subsequently, half of the blocks will be communicated among two GPUs in an interlinked way, as shown in figure 8. Naturally, by communicating multiple blocks per step, we achieve a reduction in the

total number of steps. Exactly as in the classic reduce-scatter, the idea is then used in reverse to perform the Allgather phase. With this approach, we have a communication volume of $2n$, which is the same as Ring AllReduce, but we also have a number of steps equal to $2\log_2(p)$, which is lower compared to Ring AllReduce.

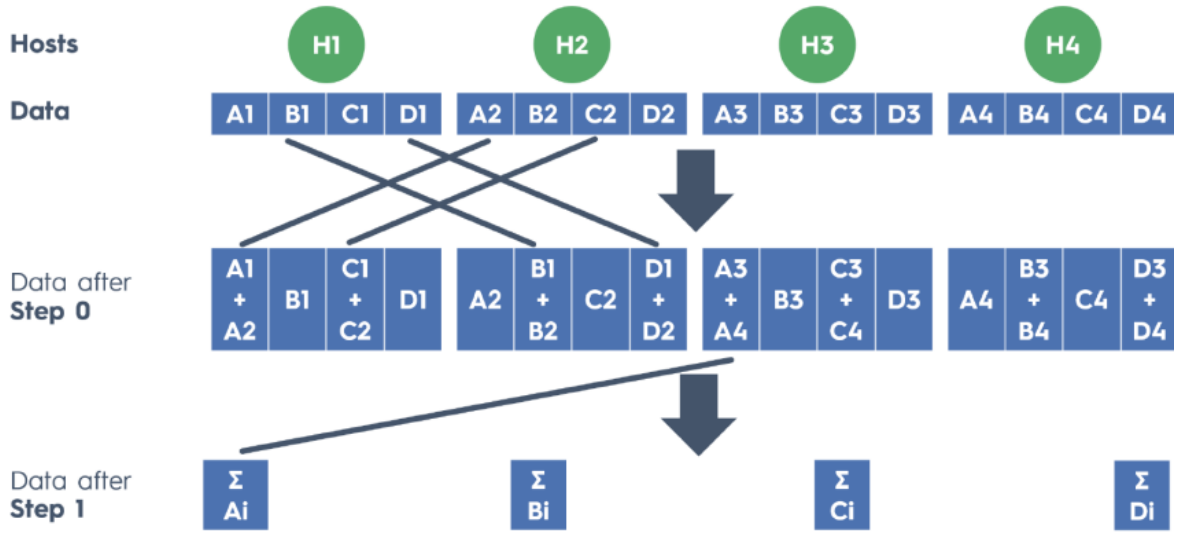

Figure 8: Example of Bw Opt. Rec. Doub. approach with 4 GPUs.

Comparing Ring AllReduce and Bandwidth-Optimal Recursive Doubling in theory, the latter should always be better than the former. However, this is not the case in practice. Fundamentally, the second case may cause issues depending on the network architecture used, one instance is shown in figure 9.

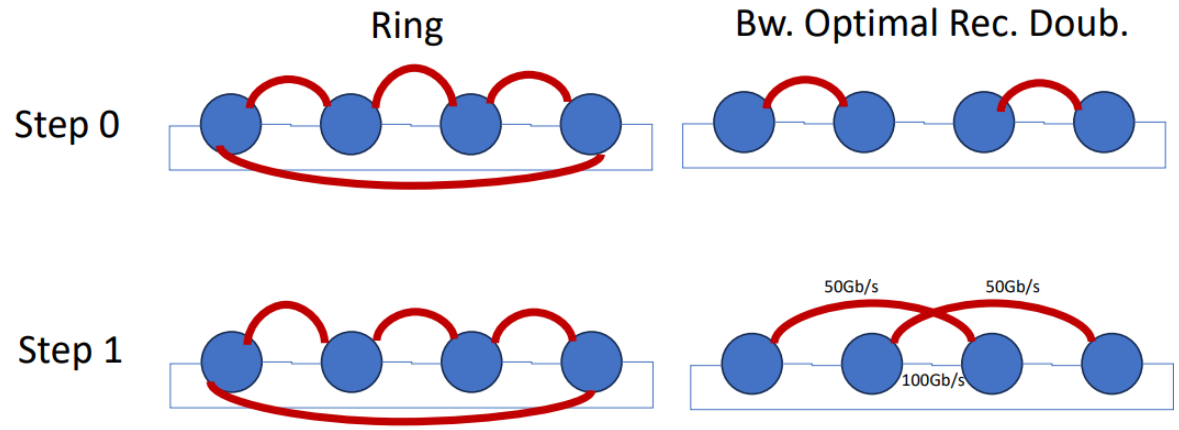

Figure 9: Ring vs Bw Opt. Rec. Doub.

The last gradient aggregation algorithm is the **Latency-Optimal Recursive Doubling**. Assuming we have p hosts, at each step i (from 0 to $\log_2(p)-1$), each GPU communicates with another at a distance of 2^i, exchanging its own gradient vector as shown in figure 10.

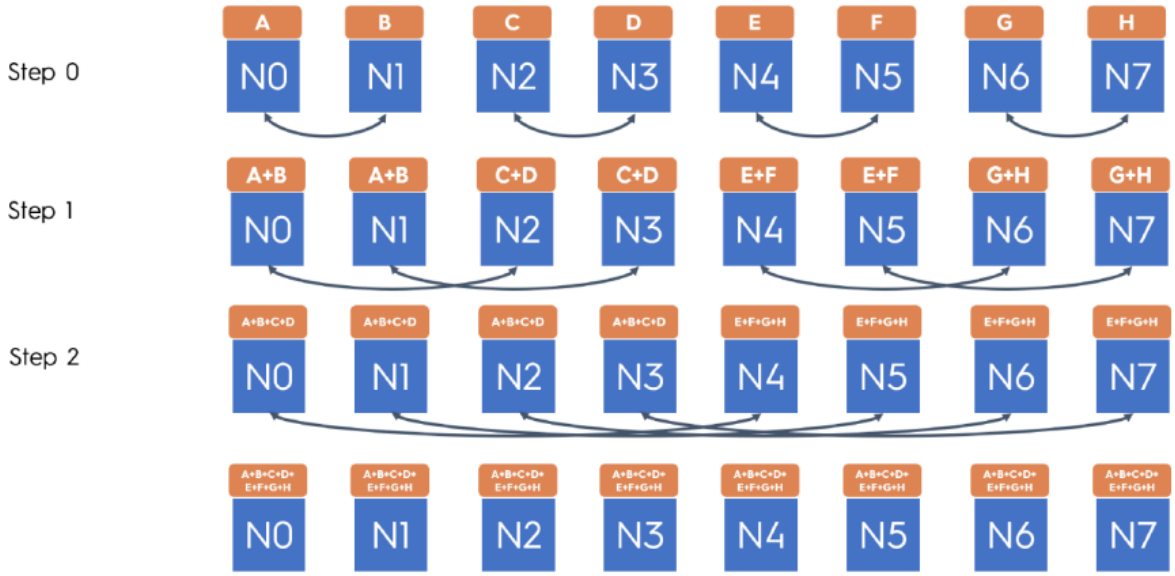

Figure 10: Example of Lat. Opt. Rec. Doub. approach with 8 GPUs.

Comparing the last 2 approaches described, Latency and Bandwidth, is not trivial. To determine which one works better, we have to consider the actual value of n and p. If n is very small, then latency is better, while if n is large, then bandwidth is better.

Algorithm	Comm. Volume	# Steps
Optimal	N	1
Parameter Server	max(n, (pn)/(K))	1
Naive Allreduce	((p-1)n)	1
Ring	$2n$	2(p-1)
Bw. Optimal Rec. Doub.	$2n$	2*log2(p)
Lat. Optimal Rec. Doub.	log2(p)*n	log2(p)

Figure 11: Gradient aggregation comparison.

3 Introduction to Hardware Architectures for Big Data Processing

Let's start by noting that in computer science there is an inherent tension between usability and specialization/efficiency. At some point you have to decide whether to build something that is simpler to use but less efficient, or harder to use but much more efficient.

Specialization refers to designing and optimizing software or hardware to perform a specific task extremely efficiently or to operate on a particular architecture. While **efficiency** refers to both performance and energy consumption.

As shown in Figure 12, a single-core CPU is the easiest model to use but also the least specialized and efficient. Next, we have multicore CPUs, which are more complex but offer better efficiency. Then come GPUs, which are even more complex and more specialized for certain types of tasks. Finally, there are more advanced architectures such as FPGAs and ASICs. ASICs (Application-Specific Integrated Circuits) are hardware architectures designed to perform a single task with maximum efficiency. For example, there are ASICs used for Bitcoin mining or TPUs (Tensor Processing Units), which are hardware devices designed by Google to accelerate their training processes.

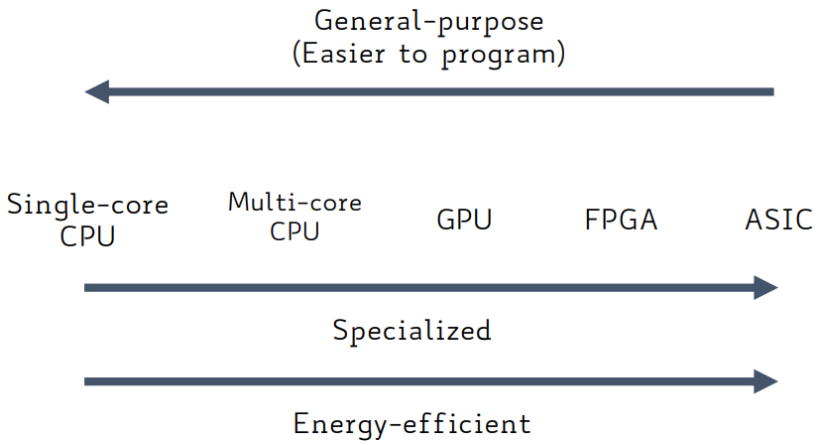

Figure 12: Tension between usability and specialization/efficiency.

3.1 Introduction to GPUs

GPU stands for Graphics Processing Unit. Initially designed for image and video rendering, they later found widespread use in video games. With the introduction of programmable shaders in the early 2000s, GPUs became more flexible, enabling customized visual effects. Over time, they evolved into general-purpose parallel processors, known as GPGPUs.

GPUs consist of many simple cores that often share a control unit, following a **throughput-oriented architecture**. This reflects specialization: sacrificing flexibility and cache to maximize parallel data processing efficiency.

To decide whether to use a CPU or GPU a common rule is to use CPUs for non-parallel tasks and GPUs for highly parallel ones. In practice, applications often split work between both.

GPUs are particularly well-suited for machine learning tasks due to several key reasons. First, machine learning is highly data-intensive, and GPUs offer significantly higher memory bandwidth compared to CPUs, allowing for faster data transfer. Additionally, machine learning algorithms heavily rely on matrix operations, such as multiplications and convolutions, which are computationally demanding. GPUs are optimized for these types of operations, making them ideal for accelerating training and inference. In fact, the parallelization of matrix computations has been a major topic of research since the early days of computer science, and GPUs naturally align with this paradigm.

A GPU consists of an array of **Streaming Multiprocessors** (SMs), each of which contains multiple cores that share control logic and the instruction cache. Each SM can also include other specialized units, such as tensor cores or ray tracing units, and all SMs together share a **global memory**. More specifically, each SM contains the following components:

- **Register file**, which is dynamically allocated depending on the requirements of each thread,
- **Constant caches**, used to cache constant data,
- **Shared memory**, which is a fast, temporary memory explicitly managed by the programmer (it is not a cache),
- **L1 cache**, local to the SM.

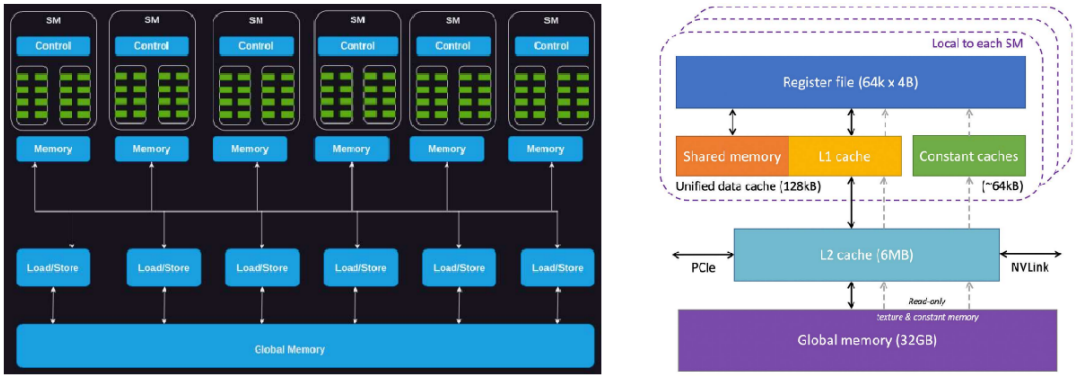

Figure 13: GPU architecture.

Each SM also shares an L2 cache, and, as previously mentioned, a global memory characterized by high bandwidth and high latency.

3.2 GPU Programming

GPU programming offers different options for developers, including:

- using standard language features, such as those available in modern C++,
- employing directive-based languages like OpenMP or OpenACC,
- utilizing frameworks that abstract the hardware details, for example Kokkos,
- leveraging libraries such as OpenCL,
- writing native code specific to the GPU architecture, for instance using **CUDA**.

A general concept to keep in mind in GPU programming is that the higher the level of abstraction of the option you rely on, the lower the performance will be.

3.2.1 CUDA Programming

CUDA is the interface provided by NVIDIA for programming their GPUs. The computation intended to run on the GPU is expressed similarly to a C/C++ function, called a **kernel**. Multiple threads execute the kernel concurrently, each working on a different portion of the input data. These threads are organized into **thread blocks**, and multiple thread blocks together form a **grid**.

Before computation, data must be copied from host (CPU) to device (GPU) memory, and results copied back afterward. Modern GPUs support unified virtual memory, letting CPU and GPU share an address space with automatic data migration, though with a performance cost. Physically, memory remains separate, so transfer costs still apply.

In CUDA, all the threads within the same block are processed by the same SM. Multiple blocks can be assigned to the same SM and executed simultaneously. Threads within a block are further divided into groups of 32 consecutive threads called warps. The SM executes all threads within a warp together by fetching and issuing the same instruction to all of them, following the **Single Instruction Multiple Threads** (SIMT) model. Given that all threads in a warp must execute the same instruction, it's easy to see how a so-called **branch divergence** can occur, as shown in figure 14.

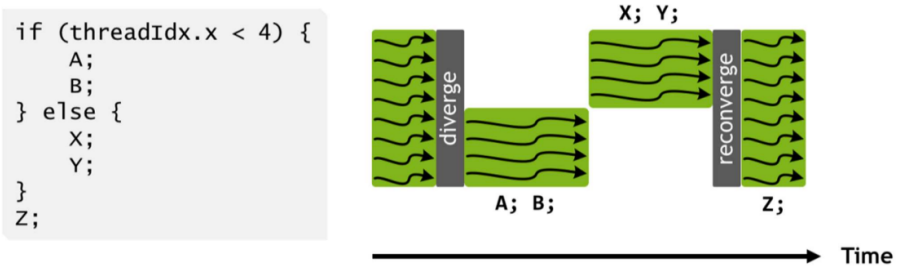

Figure 14: Branch divergence issue.

The reason GPUs are excellent for matrix operations is that there is no branch divergence, since each thread simply computes one element of the matrix.

3.3 TPUs

Modern computer systems follow the Von Neumann architecture, which separates CPU from main memory. This leads to the **Von Neumann Bottleneck**, where the time spent moving data can outweigh the time spent computing. For example, fetching from the L1 cache takes about 0.5 nanoseconds, and performing an addition even less, while accessing main memory can take around 100 nanoseconds. This means that a large portion of execution time and energy is spent on memory access rather than computation.

To cut time and energy costs, companies like Google have developed alternative hardware such as **TPUs** (Tensor Processing Units), specialized for processing multi-dimensional arrays (tensors).

TPUs are a type of **accelerator** or **Domain-Specific Architecture** (DSA), designed to work with CPUs (just like GPUs) and optimize **tensor operations**, i.e., matrix-based linear algebra tasks.

In practice, TPUs are composed of **tensor cores**, which themselves consist of three main components: a **scalar unit** (similar to a CPU), a **vector unit** (similar to a GPU), and a **matrix multiplication unit** specifically designed to perform operations on vectors and matrices (Usually referred to as a spatial architecture, because the computation is arranged in a 2D space). These tensor cores are also connected to **high-bandwidth memory** to support fast data access and throughput.

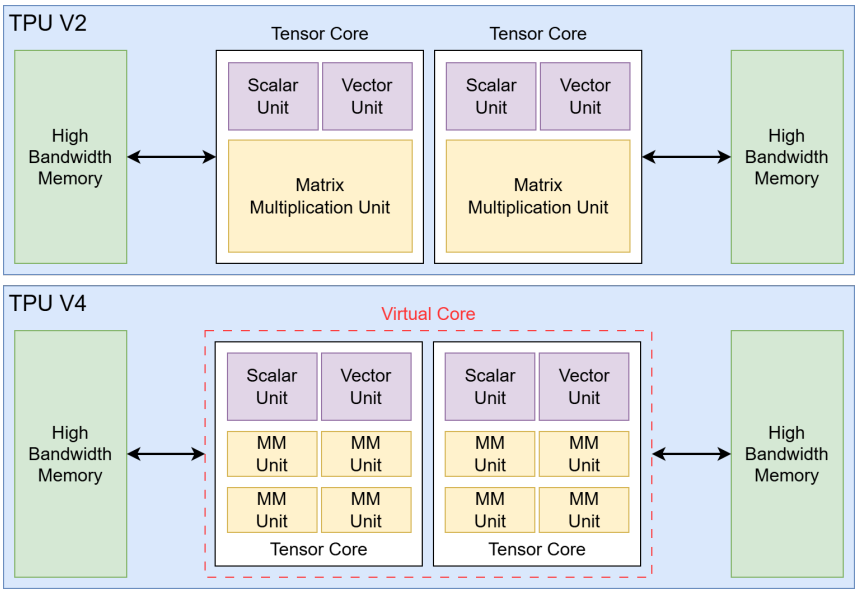

Figure 15: TPU versions compared.

Matrix multiplication is performed without relying on external memory by using

the matrix multiplication units, which employ an architecture known as **systolic array** (shown in figure 16) (watch this video:youtu.be/eK5fjuEFJu0?si=N3z9xfRy2RSpyOUc).

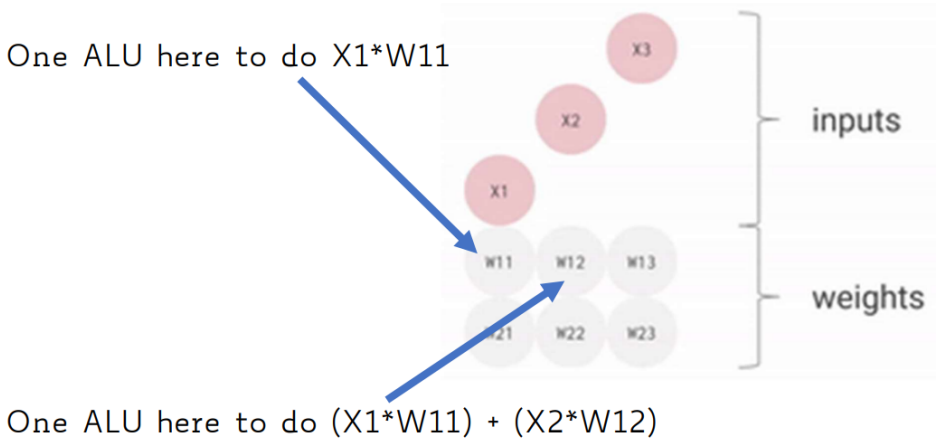

Figure 16: Systolic array architecture.

These operations can be performed using various data formats, but it has been shown that for many workloads, it is more effective to use a format with 1 bit for the sign, 8 bits for the exponent, and 7 bits for the mantissa, thus favoring the representation of **larger numbers** over higher numerical precision. This new data type is called **bfloat16**.

It is worth noting that TPUs do not follow the Von Neumann architecture, whereas GPUs do. However, one can outperform the other depending on the workload.

4 Network Topology Design

So far, we have explored various types of architectures and optimization strategies. In this chapter, we aim to connect all these elements together, and this can be modeled as a **graph problem**.

4.1 Some Definitions

- **Switch**: is a hardware device that connects multiple nodes (e.g., GPUs, or other switches); it receives data from one node and forwards them to the appropriate destination.
- **Switch radix**: number of switch ports (to how many other switches/servers it can connect to).
- **Regular/Irregular**: the topology is called *regular* if it can be represented as a regular graph (e.g., a ring), otherwise it is *irregular*.
- **Hops**: the number of links that must be crossed to reach a specific destination.
- **Network diameter**: the maximum distance (in number of hops) between any two switches in the network.
- **Path**: a route connecting one switch to another.
- **Average distance**: the average number of hops across all valid paths between switch pairs.
- **Blocking vs. Nonblocking**:
  - *Nonblocking*: every possible pair of switches can be connected through disjoint paths. In the ideal case, this means no congestion (on average).
  - *Blocking*: when disjoint paths cannot always be guaranteed, potentially causing congestion.
- **Direct vs. Indirect networks**:
  - *Direct*: each switch is directly connected to some server (node).
  - *Indirect*: not all switches are directly connected to servers; some are used solely for routing.

The **Diameter** is an indicator that helps determine whether a graph is efficient or not; in general, the smaller the diameter, the better. Another method is to analyze how dense the graph is; the denser it is, the more disjoint paths it contains, and the less likely it is to experience blocking.

The **Bisection Cut** is a further indicator, which is the minimum number of links that must be removed to divide the graph into two disjoint parts. The idea is that the smaller the bisection cut, the more likely it is that multiple data flows will need to traverse the same link, increasing the chance of congestion. This concept is also expressed through the **Bisection Bandwidth**, which is equal to the Bisection Cut $\times$ Link Bandwidth.

Another estimator is the **All-to-All Bandwidth**, which represents the maximum bandwidth achievable when every node communicates with all other nodes simultaneously.

Different topologies present various trade-offs in terms of cost and performance. These trade-offs often depend on the specific workload, as some less performant topologies may still be suitable depending on the communication pattern.

4.2 Ring, Mesh and Torus

The first network topology is the **Ring**, which has a diameter of $n/2$ and a bisection cut equal to 2, where n is the number of nodes in the graph. From now on, figures will show in orange the nodes that determine the diameter and in red the bisection cut. Although the following images illustrate switches, the nodes could be GPUs as well.

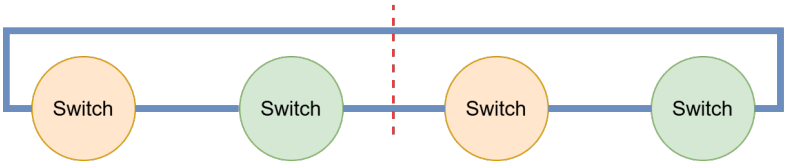

Figure 17: Ring network topology.

Next is the **Mesh**, which has a diameter of $2(\sqrt{n}-1)$ and a bisection cut of $\sqrt{n}$.

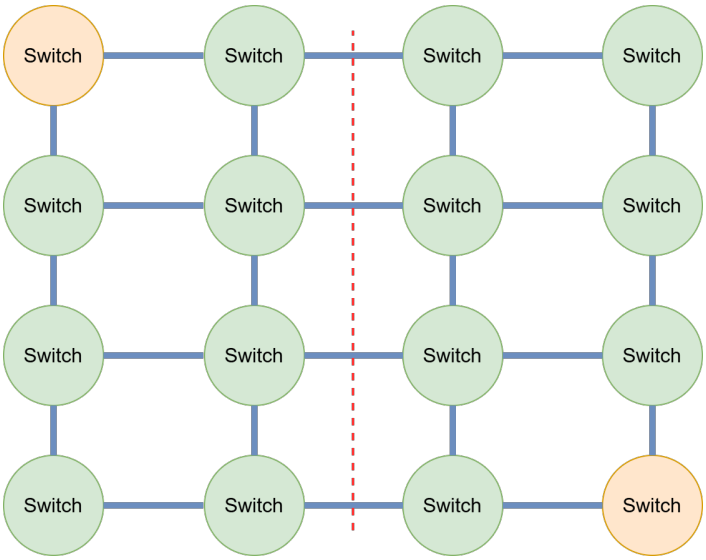

Figure 18: Mesh network topology.

Finally, we have the **2D Torus**. A Torus can have multiple dimensions, for example, a 1D Torus is equivalent to a Ring. In a k-dimensional Torus, each node is connected in all k directions to its neighboring nodes. For example, in the following 2D representation, each node is connected to its north, south, west, and east neighbors. In this case, the diameter and the bisection cut are $\sqrt{n}$ and $2\sqrt{n}$, respectively.

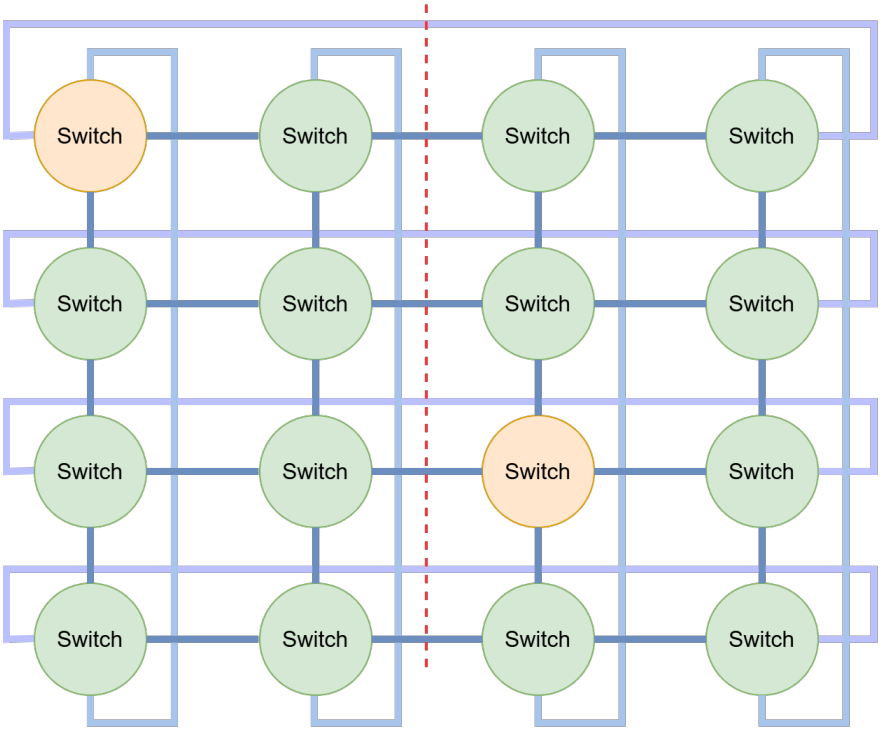

Figure 19: 2D Torus network topology.

4.3 Trees

With tree topologies, a distinction is made between nodes by representing switches with rectangles and GPUs (servers in general) with circles. A classic tree is the first type of indirect topology we encounter. It always has a bisection cut of 1, while the diameter is $2height$, where $height = \log_{r-1}(n)$ and r is the switch radix.

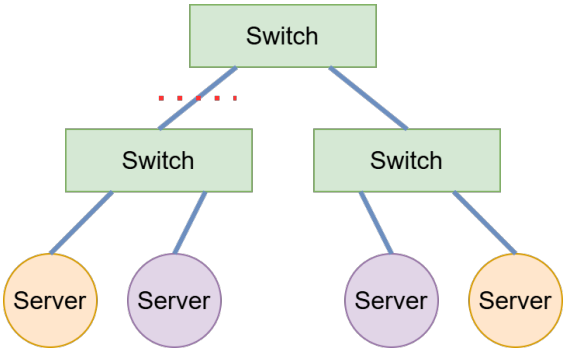

Figure 20: Tree network topology.

A classic tree topology is not well-suited for this kind of workload, as each switch typically has one uplink and $r - 1$ downlinks, which often leads to congestion near the root. To overcome this limitation, **Fat Trees** were introduced. In a Fat Tree, for every downward link there is a corresponding upward link of equal bandwidth, ensuring balanced communication at every level of the hierarchy. Assuming n servers, the height of the tree is $h = \log_{r/2}(n)$, making the diameter $2\log_{r/2}(n)$, and the bisection cut equals $n/2$. This architecture provides **full bandwidth**, meaning it is non-blocking, and also achieves **maximum all-to-all** and **bisection bandwidth**, as these cannot be improved further. The problem with this network is that nodes

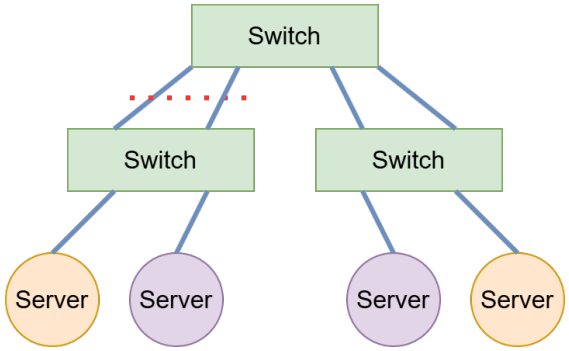

Figure 21: Fat tree network topology.

have different numbers of ports, which is impractical in real-world implementations. Therefore, it is a network that performs well in theory but is less suitable in practice. For this reason, in practice, each switch is replaced by smaller switches so that all of them have the same number of ports, and is called **Clos Network** (or folded). This

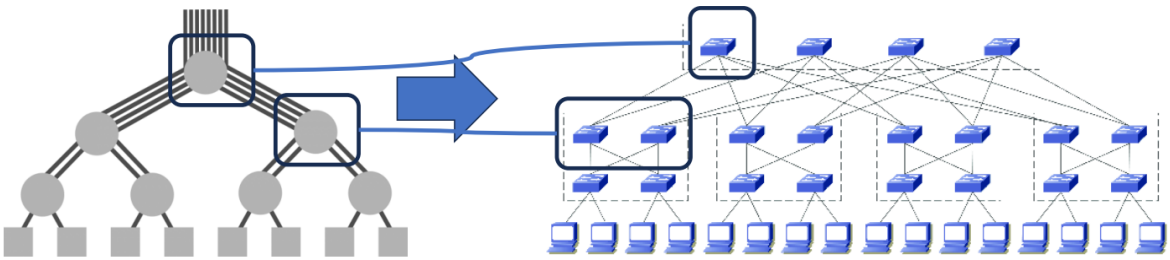

Figure 22: Clos network topology.

is the best network that can be built in terms of performance, but it is very expensive due to the high number of links. Over time, alternative designs have been developed that, while not matching its performance, are more affordable.

A less expensive alternative (although blocking) is to use, for every l downward links, only $l/2$ upward links. In real systems, however, it is common to use an equal number of cables in the initial layers, and then adopt a 2:1 ratio in the higher levels.

4.4 Dragonfly

The **Dragonfly** topology consists of fully connected groups of switches, where each switch within a group is connected to all others in the same group. Additionally, the groups themselves are interconnected, with at least one link connecting each group to every other group. For example, in a setup with 7 groups of 6 switches, each pair of groups is connected by a single link, though this inter-group connectivity may vary in general.

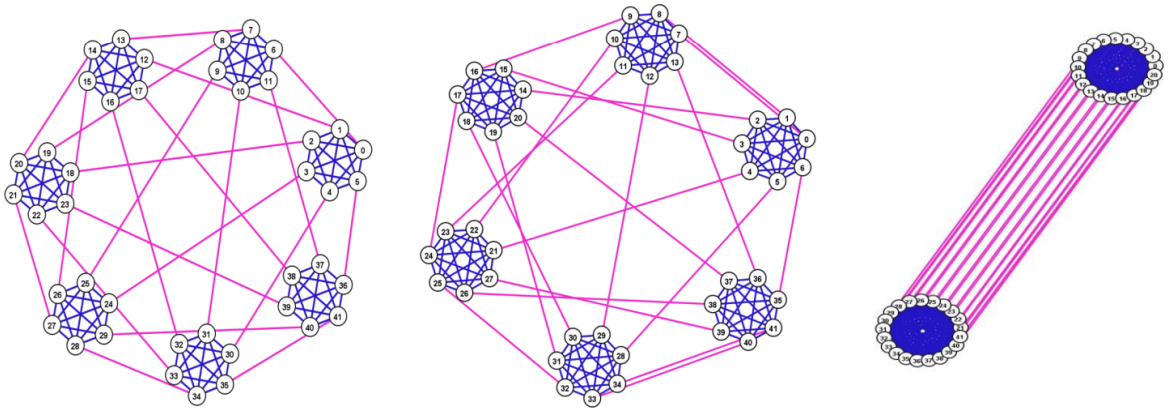

Figure 23: Dragonfly topology with different numbers of switches and groups.

The advantage of this type of structure is that it requires significantly fewer links compared to a Fat Tree, and it offers greater flexibility, as one can choose to have more switches per group but fewer groups, or the opposite depending on the workload. Another two very important strengths are that, for a fixed switch radix, the dragonfly topology can connect many more nodes, and for this reason we also have that the diameter is always equal to 3.

Dragonfly topology also has some disadvantages, for example, networks may not guarantee full bandwidth like non-blocking fat trees, they are harder to expand incrementally, make load balancing more complex due to the mix of minimal and non-minimal paths, and can introduce loops that may lead to deadlocks, though these can be avoided with additional hardware.

Finally, we have the **Dragonfly+** structure, which differs in that each group is actually a Fat Tree. This topology allows for larger groups and connects more nodes than Dragonfly with the same switch radix. It's as cost-effective as Dragonfly but easier to expand, has the same diameter (3), and requires fewer resources to avoid deadlocks.

4.5 HammingMesh

The intuition is to start from a Torus and connect each column to a Fat Tree and each row to a Fat Tree. This significantly reduces the diameter of the structure at scale.

4.6 Honorable Mentions

There exists one highly complex graph theory-based architecture which reduce the diameter to 2, it's name is **Slimfly**. This topology is cheaper than a Fat Tree since it requires fewer switches and links. However, it comes with some drawbacks: it only works for specific switch radix values, the wiring is less intuitive (compared to Dragonfly or Trees), and expanding the number of servers is more challenging than in Fat Trees

Some people also tried to work on random graphs, this is the case of **Jellyfish** topology, which is theoretically pretty good but it didn't find any practical development.

Finally, **Reconfigurable Data Center Networks** (RDCNs) are architectures where the connections between switches are not fixed but can change dynamically according to the current workload and traffic patterns. This approach introduces the possibility of optimizing the topology based on real-time communication needs, potentially improving performance and resource utilization. However, it also introduces challenges such as deciding whether to rely solely on dynamic links or to maintain a combination of dynamic and static connections. Reconfiguring the network, often involving the physical movement of components like lasers or mirrors, has a cost, so it must be done only when there is a clear benefit over time. Additionally, the routing logic becomes more complex, as it must adapt to a topology that can change over time, with links that may appear or disappear based on traffic behavior.

5 Data Communication

Traditional communication protocols like TCP or UDP are not well-suited to handle the scale and data volume of modern high-performance networks. As a result, new protocols have been developed from scratch to better meet the specific demands of these specialized network architectures.

First of all, we need to distinguish between **North-South Traffic** and **East-West Traffic**. As shown in the figure below, what we have so far referred to as a "server" is actually composed of multiple components, but as a whole, it connects to a switch, which in turn connects to the **Data Center Network** (DCN). The DCN is also connected to the external world (i.e., the Internet). Therefore, we differentiate between two types of traffic: North-South traffic refers to communication between the DCN and the external world, while East-West traffic refers to communication between servers within the data center.

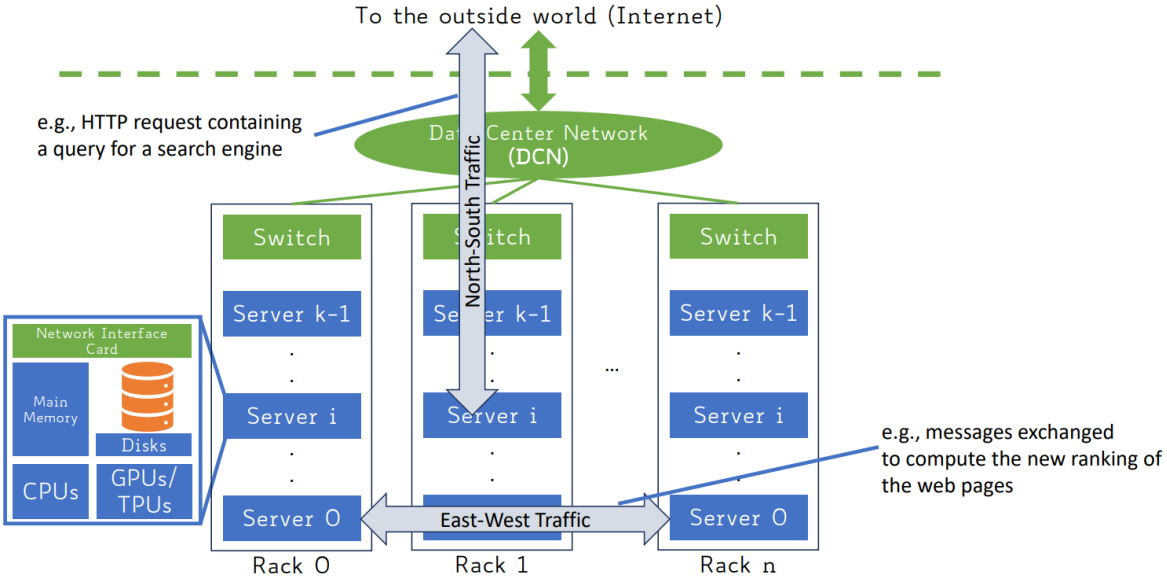

Figure 24: North-south and east-west traffic.

Clearly, for north-south traffic, the TCP protocol is suitable, but for east-west traffic, something completely different happens.

5.1 TCP/IP Limitations

One of the main reasons why TCP/IP protocols are not suitable is that too much time is spent on network processing rather than on application processing. The overhead occurs for two main reasons. The first is that if you want to send just one byte, you still have to wrap it in a data structure that includes various headers, totaling around 200 bytes. These headers are useful for communication over the internet but unnecessary for communication within a data center. The other reason is that all these

operations performed through sockets are system calls, and system calls are generally more expensive due to security issues.

To overcome this problem, several strategies have been adopted. First, some functionalities have been offloaded to the NIC hardware (such as DMA engines, interrupt coalescing, scatter/gather, fragmentation, segmentation, TCP offloading, etc.). Then, multicore CPUs are used to process packets in parallel. Finally, the operating system is bypassed by implementing most of the packet processing in user space (security is preserved thanks to a separation of data processing and control processing).

TCP offload refers to shifting network processing from the kernel to the NIC (practice of specialization). While the idea is still occasionally used today, developing dedicated hardware often becomes less meaningful, especially since CPU performance doubles approximately every two years according to Moore's Law.

Over the past 20 years, the number of cores in CPUs has increased linearly. This is because, at some point, it was no longer possible to keep increasing the clock frequency of a processor, and as a result, the only viable solution was to integrate more cores onto the same chip. Multicore processing is useful, but it comes with challenges: cores may access the same data at the same time, causing conflicts, and using locks can slow things down. Badly designed data structures can also lead to cache issues; it's important to plan data layout carefully, decide which core handles interrupts, and try to keep packet processing and application logic on the same core for better performance.

5.2 RDMA

The solutions seen so far inevitably keep the operating system and CPU involved in network protocol processing, resulting in high latency. In the long run, this leads

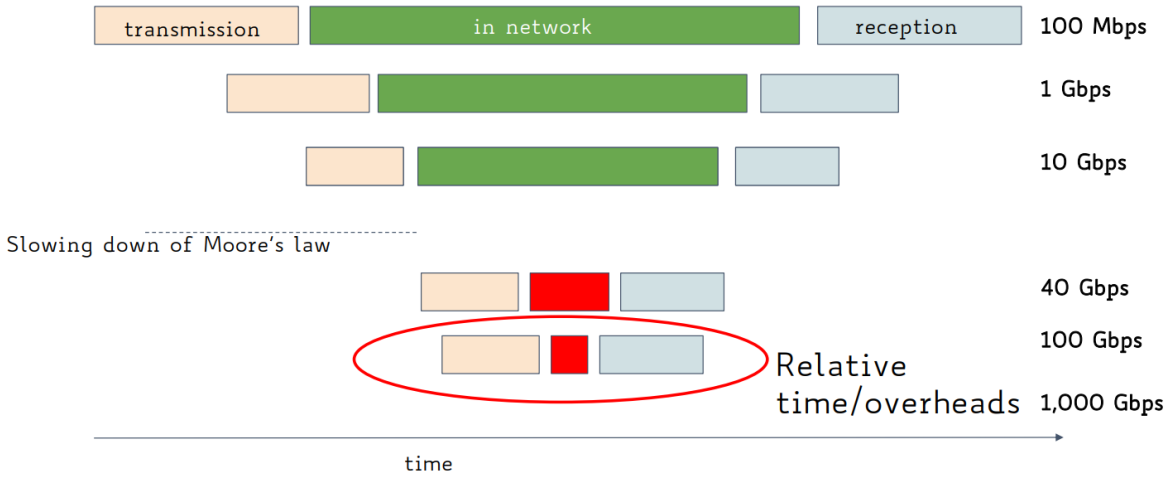

Figure 25: Shifting performance bottlenecks.

to a kind of bottleneck (figure 25), as the network becomes faster, but transmission

and reception operations can't improve significantly, ultimately limiting the overall performance of the system.

The consequence of all this has been the development of **Remote Direct Memory Access** (RDMA). The idea is to allow a node A to access the memory of a node B remotely and without going through the operating system (hence "direct"), as if it were its own memory. Today, more and more supercomputers and data centers, including ChatGPT, are adopting RDMA.

RDMA is first of all a networking technology (a concept) that can be implemented in several ways and several protocols (Amazon has its own). It basically speeds up the transmission of data over the network, improving performance, and it is used in many other fields such as picture processing.

In practice, two main ideas are leveraged. The first is based on **user-space networking**, where the part typically executed by the kernel (as in the case of the TCP network stack) is split between a network controller (a NIC) that handles multiplexing, security, and isolation, and another part that is executed in user space. This effectively removes almost all network-related operations from the operating system. Clearly, this means the network controller is not a standard NIC; it is referred to as an RDMA NIC or RNIC. The second idea is based on **kernel bypass**, meaning that the portion of the stack running on the NIC communicates directly with the user-space stack, once again completely bypassing the operating system. Moreover, the socket is conceptually split into two parts: data operations and control operations, as shown in the figure. The advantage of this approach is that control operations are performed only once, interact with the kernel at application startup, and are handled separately.

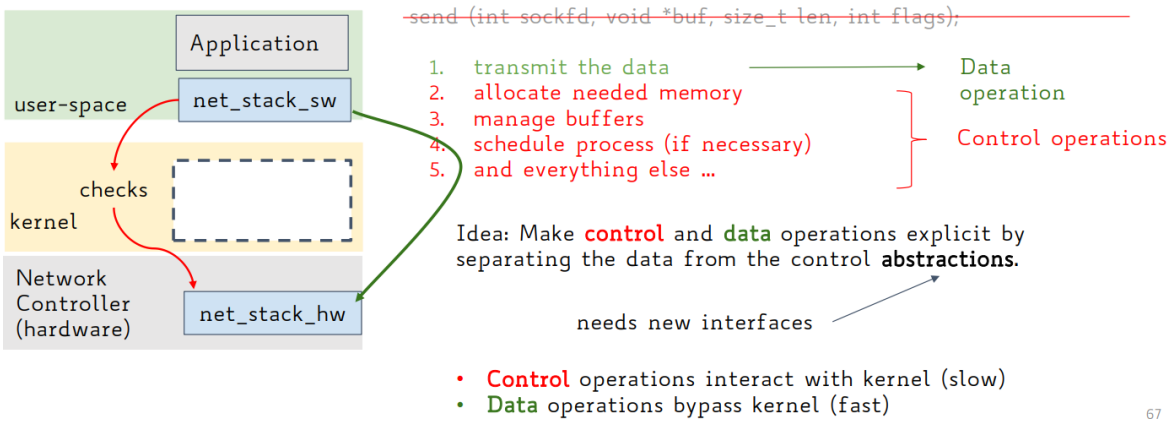

Figure 26: RDMA abstraction of operations.

Putting it all together, we obtain a sequence of distinct steps which can be data operations or control operations:

1. Allocate memory buffers and register them on the NIC (CO)

2. Allocate data and control queues which are used as a communication channel between the application and the NIC (CO)

3. Transmit some data by writing a control message into the data queue to inform the NIC of the intention to receive data into a specific memory buffer (DO)

4. Once the NIC receives the message, it writes directly into the requested memory and then notifies the application (DO)

5. When the need to transmit ends, the connection is closed (CO).

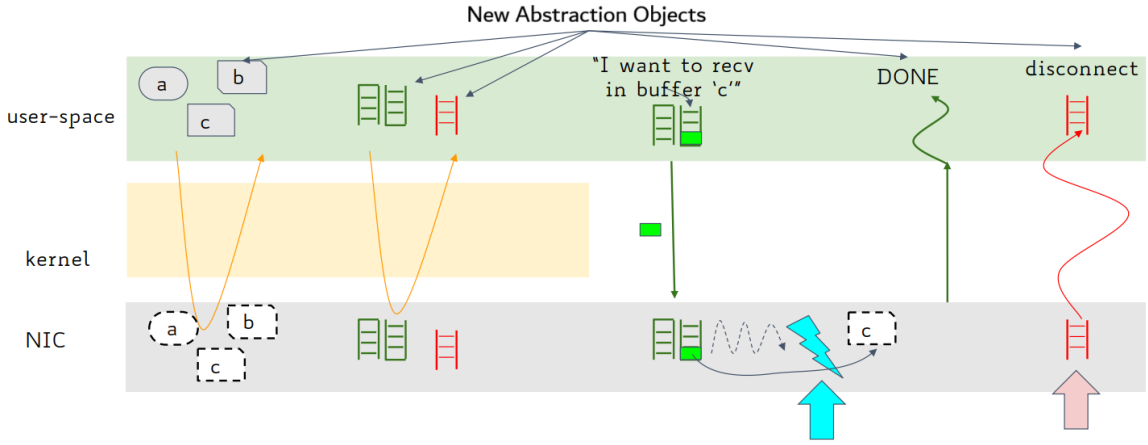

Figure 27: RDMA communication steps.

RDMA introduces several new communication abstractions. First, **memory buffers** are used as the source and destination for data transmission and reception. Then, there are connection send/receive queues, which represent a connection and are also known as **queue pairs**. These queues contain only metadata, not the actual data. Additionally, RDMA includes **control queues**: the network control queue, which handles events such as new connections, disconnections, and NIC status changes, and the I/O event queue, which reports when network I/O operations are completed or if there are errors. Network I/O is carried out by posting work requests to the queue pairs. Each work request includes a pointer to the buffer, its length, and possibly more information. A key difference from traditional socket-based communication is that with RDMA, only pre-registered memory buffers can be used for I/O. Unlike sockets, you cannot send or receive from arbitrary memory locations, but once the memory buffer is registered the NIC remembers it till the disconnection.

Challenges with RDMA include performance issues such as the learning curve for event-driven coding, managing quality of service, traffic, and security compliance, and complex congestion control. Regarding scalability, concerns involve limits on the number of concurrent connections a server can handle and how many memory buffers an RNIC can track. RDMA also presents some key challenges about security.

5.2.1 RDMA Operations

Here is the execution of the three operations provided by RDMA: **Send/Receive**, **Write**, and **Read**, assuming that all memory buffers are known in advance by all participants within a cluster or data center.

In the **Send/Receive** operation, both the client and the server declare their intent to communicate. This is known as a **two-sided operation**, as both sides must actively participate.

1. Both the client and the server create queue pairs (QP) using the kernel and CPU.
2. The client posts a send request, while the server posts a receive request.
3. The client's NIC reads the data from memory and transmits it over the network.
4. The server's NIC receives the data. The message includes the address of the destination buffer, and the NIC writes the data to that address. A completion event is then added to the control queue to notify the server.
5. The server application is notified via the control queue.
6. The server's NIC sends an acknowledgment indicating that the transmission is complete.
7. The client's NIC receives the acknowledgment and notifies the client process.

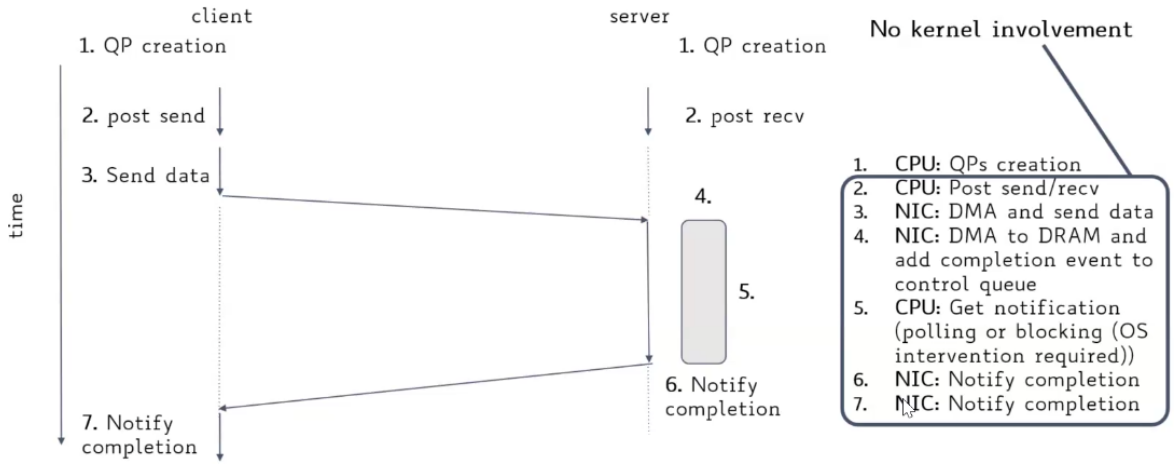

Figure 28: Send/receive timeline.

RDMA **Read** and **Write** are **one-sided operations**, which means the server doesn't have to authorize the operation. So basically, the client reads or writes to the server's memory as if it were its own.

1 Both the client and the server create queue pairs (QP) using the kernel and CPU.

2 The client declares the intention to read or write, specifying the source and destination buffers.

3 The NIC sends the request and, in the case of a write, also the content to be written.

4R The server's NIC responds by retrieving the content from the server's memory and sending it to the client.

4W The server's NIC writes to the server's memory and notifies the client.

5R The client's NIC writes to the client's memory and notifies the process of the operation completion.

5W The client's NIC receives the completion notification and in turn notifies the process.

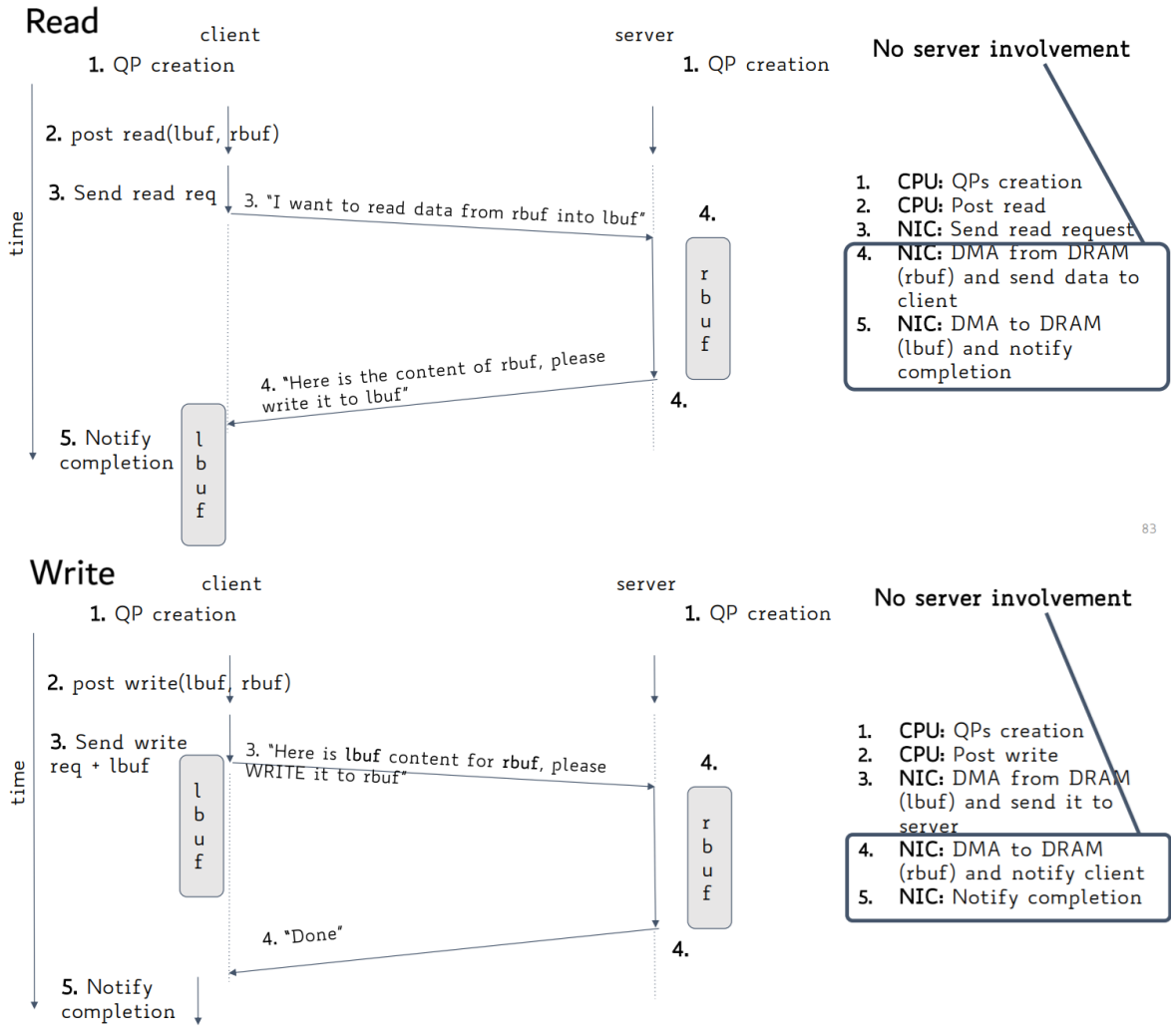

Figure 29: Read and write timeline.

From an architectural perspective, what happens is that when queue pairs are created, the addresses of the buffers are registered with the NICs. This is because work requests, which are posted to the QPs, refer to these registered buffers, and in this way, the NIC can access them without interacting with the operating system.

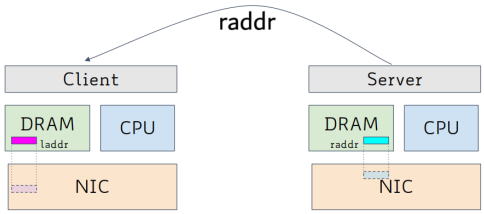

Figure 30: Architectural view of RDMA.

Subsequently, the server must communicate the buffer addresses to the client so that the client can specify them in its requests as shown in figure 8.

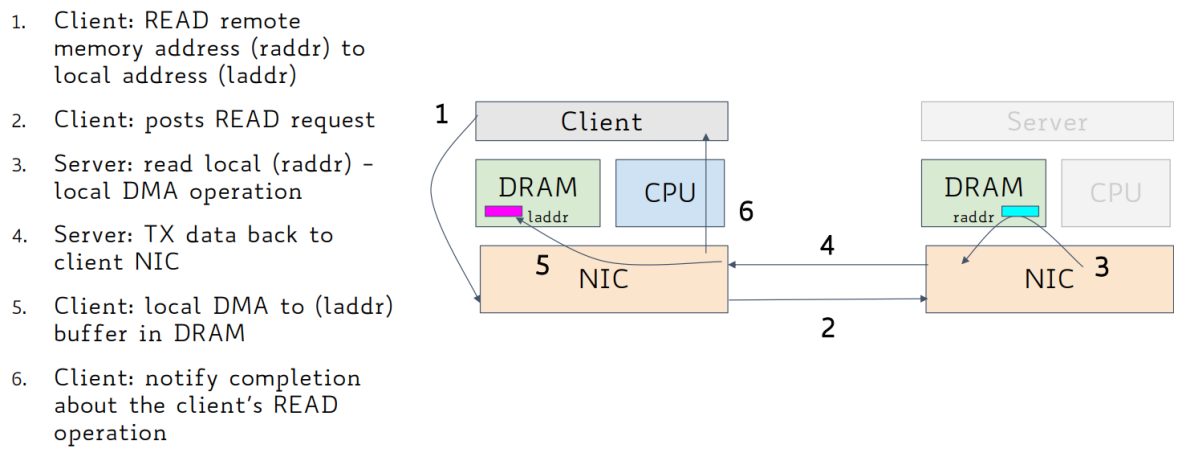

Figure 31: Architectural view of RDMA Read timeline.

In terms of performance, the two-sided operation takes 4.6 times less time compared to using TCP/sockets, while the one-sided operations take 10 times less time.

RDMA also supports other types of operations such as **Atomics** and **Transactions**. Atomic operations allow a node to perform synchronized updates on a remote variable (e.g., compare-and-swap or fetch-and-add), ensuring safe access in concurrent environments. Transactions, on the other hand, enable the execution of groups of RDMA operations as a single indivisible unit, ensuring consistency even in the presence of errors or failures.

5.2.2 RDMA Interfaces and Protocols

To use RDMA in practice, for what concerns **interfaces**, there are APIs that allow you to create queues, delete them, post messages, send and receive addresses, and so on, such as **libibverbs** or **libfabric**, which is based on libibverbs and some others with the objective to be more portable.

For what concerns **protocols** the most popular are **Infiniband** and **RoCE**, the most well-known and widely used RDMA technologies, both in data centers and supercomputers. They are based on open standards and managed by the InfiniBand Trade Association (founded in 1999). Mellanox (now part of NVIDIA) is the main driving force, with other key members including AMD, Intel, IBM, Microsoft, Cisco, and Broadcom. Alternatively, there's iWARP, which is essentially RDMA over TCP. It was created to extend RDMA to the internet, but it never gained much traction because it doesn't provide strong performance results. There are also others such as SRD which is developed by Amazon.

AWS chose to develop SRD instead of using RoCE because the latter had issues with congestion control on very large networks, lacked native support for traffic balancing over multiple paths, and did not handle out-of-order packet delivery well. Additionally, SRD uses fewer queue pairs, which is crucial for scaling to thousands of nodes.

InfiniBand (IB) has some issues, for example, the link layer includes the requirement that all switches in the network must be IB switches, which is problematic for data centers that prefer to use switches from multiple vendors (NVIDIA is the only vendor of IB switches). This is less of an issue in supercomputers, where uniform hardware is common. Additionally, since the outside world uses Ethernet, gateways are needed to translate between Ethernet and IB within the data center. In response to these issues, RoCE (RDMA over Converged Ethernet) was developed. It fundamentally replaces the link layer with standard Ethernet. With the introduction of RoCEv2, it also replaces the network layer by adopting IP and UDP protocols, allowing RDMA to operate over routable Ethernet networks.

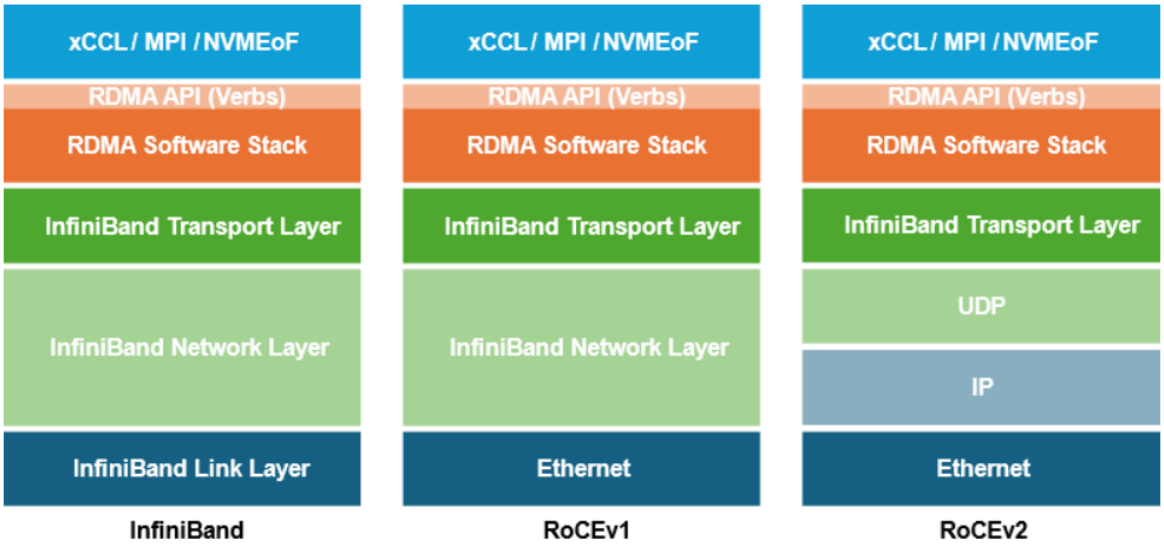

Figure 32: IB compared to RoCE.

Infiniband also guarantees reliable delivery without using TCP by defining three transport modes: Reliable Connected (RC), which uses one queue pair per connection

and ensures reliability with acknowledgments; Unreliable Connected (UC), which also uses one queue pair per connection but relies on a lossless network for reliability; and Unreliable Datagram (UD), where one queue pair can serve multiple connections for better scalability, but without reliability guarantees. Not all RDMA operations are supported in every transport mode. In general, the idea is that if you want to be

	SEND/RECV	WRITE	READ	WQE header
RC	✓	✓	✓	36 B
UC	✓	✓	✗	36 B
UD	✓	✗	✗	68 B

Figure 33: Operations available for each transport mode.

reliable, you have to sacrifice scalability, and if you want to be scalable, you have to sacrifice reliability.

Even though RDMA can run over UDP or TCP, an RDMA NIC is still required. For this reason, RoCEv2 and iWARP are mainly used for data center-to-data center traffic. To address this, there are software-based solutions like **Soft-RoCE** and **SoftiWARP**, which are full software implementations of RDMA over TCP and work with any NIC. They are useful for debugging RDMA applications or enabling gradual deployment, allowing RNICs to be installed on only a few servers at a time.

5.3 Smart NICs

Smart NICs (Smart Network Interface Cards) are advanced versions of traditional network interface cards (NICs) that include computational capabilities. They are designed to offload and accelerate specific tasks from the main CPU, improving system performance and efficiency.

A first distinction to be made is between **on-path NIC**s and **off-path NIC**s. In the first case, all packets pass through the NIC, which can analyze, or modify the traffic before it reaches the CPU or memory. In the second case, the NIC works in parallel, analyzing copies of the packets or intervening only when requested, thus the application traffic can go directly to the CPU without being processed by the NIC.

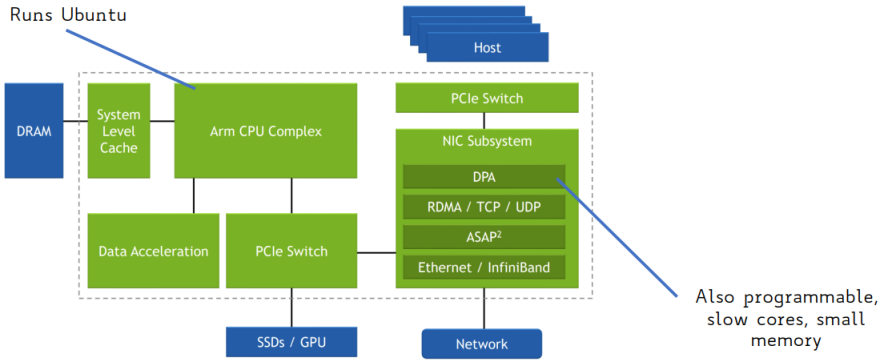

Figure 34: NVIDIA Bluefield off-path smart NIC.

An example of an off-path Smart NIC is BlueField by NVIDIA, which combines general-purpose cores running Ubuntu, uses DPDK (Data Plane Development Kit) for high-performance user-level packet processing, and supports DPA (Data Processing Architecture), enabling event-based programming where execution is triggered by network events such as packet arrivals instead of constant polling.

The general idea is that off-path NICs can perform operations with greater flexibility, while on-path NICs are faster but they can only carry out basic operations like flipping a bit or adding a constant.

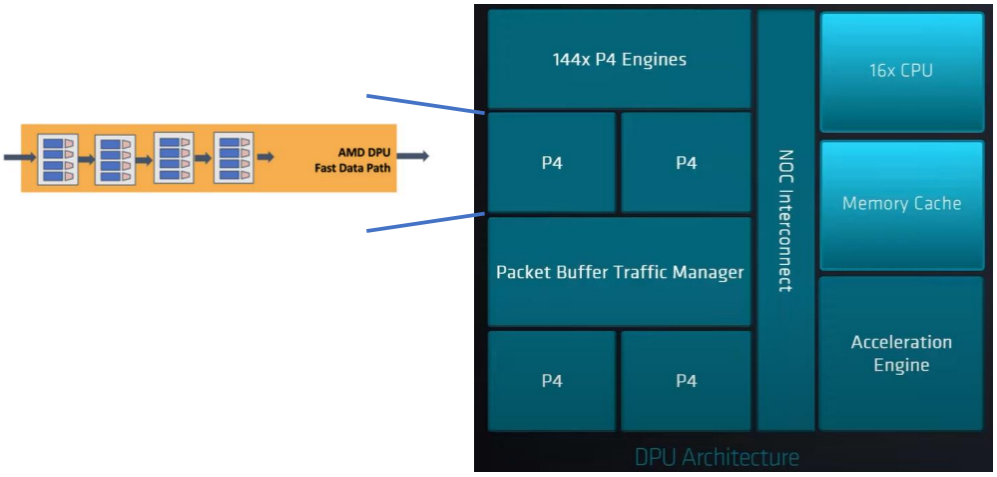

Figure 35: AMD Pensando on-path smart NIC.

One further idea for a smart NIC is the PsPIN, which is a programmable on-path NIC packet processing architecture. It operates in a run-to-completion manner, meaning each packet is fully processed before the next one begins. It includes a hardware scheduler and runs without an operating system or caches, as packet processing typically exhibits low temporal locality.

Alternatively, there are FPGAs, which are somewhere between integrated circuits and general-purpose processors. In general, they are devices that can be programmed to perform specific tasks; a kind of programmable hardware. However, they are notoriously difficult to program and debug. All these options, from FPGAs to general-purpose cores running full operating systems like Ubuntu, to fully specialized hardware, represent a broad spectrum of solutions.

5.3.1 Shuffle Use Case

A common and computationally intensive pattern in big data applications is **Shuffle**. Consider a simple example where we want to count the number of occurrences of each word in a large text file. The processing is divided into the following phases:

1. **Map Phase:** The input file is split into lines and distributed across multiple servers. Each server processes its share and produces a list of `(word, count)` pairs.

2. **Shuffle Phase:** All key-value pairs corresponding to the same word must be routed to the same server. This requires data partitioning based on the key (word), and the network becomes heavily involved in redistributing the data.

3. **Reduce Phase:** Each node aggregates the counts of the words it received and outputs the final results.

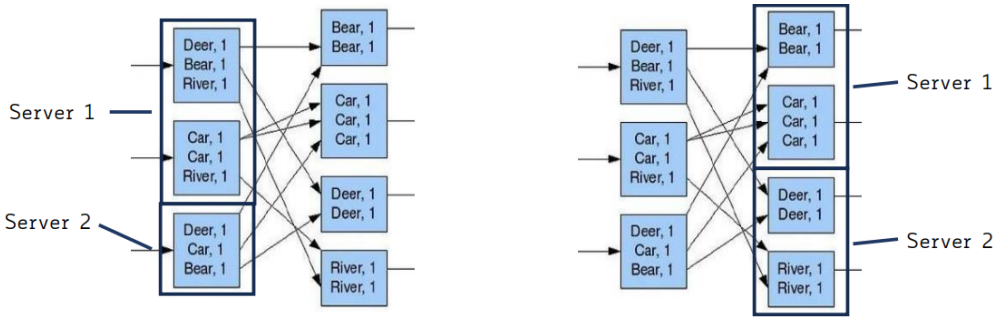

Figure 36: Shuffle phase.

This shuffle step is particularly network-intensive and can represent between 20% and 40% of the total job runtime. One of the major issues is that every partition must be sent over the network, generating a large number of small messages (often just a few kilobytes), which leads to high overhead due to the fixed header size (around 60 bytes or more).

Smart NICs are proposed as a solution to accelerate the shuffle phase by offloading the pre-processing operations, specifically aggregation, onto the NIC. This approach has several potential benefits:

- It reduces the volume of data sent over the network by aggregating word counts before transmission.
- It enables overlapping of communication and computation, allowing the CPU to process new data while the NIC performs aggregation.

However, this design presents two main challenges:

1. **Limited Memory:** SmartNICs typically have constrained memory, making it difficult to store large intermediate datasets during aggregation.
2. **Compute Imbalance:** The compute cores on the NIC are significantly slower than the CPU cores. In some cases, the host CPU may finish its tasks before the NIC, leading to underutilization.

To address these issues, a dynamic strategy is adopted:

- When the NIC becomes a bottleneck, part of the computation is offloaded back to the host dynamically.

- When the NIC memory is insufficient to hold the data, a mechanism called spilling is triggered, where excess data is transferred back to the host for processing.

Experimental results show that this hybrid aggregation approach, using both the host and the NIC, can reduce runtime by up to 30%, depending on the workload. Additionally, it significantly decreases CPU utilization, freeing resources for other tasks.

5.3.2 Gradient Aggregation Use Case

A second relevant use case for smart NICs in data-intensive applications is **gradient aggregation**. Gradient aggregation typically consists of summing parameter vectors across multiple workers at each iteration of the training process.

One of the most common algorithms for this task is Ring AllReduce (discussed in distributed deep learning chapter). In this method, the gradient vector is divided into slices, and each worker node iteratively exchanges and aggregates slices with its neighbor in a logical ring topology.

Each step introduces latency due to memory access and PCIe transfers. A proposed optimization involves offloading the aggregation to a smart NIC, bypassing host memory and CPU entirely. The process becomes:

- The smart NIC holds the local gradient slice.
- It receives a slice from the previous worker.
- It performs the summation directly on the NIC.
- It transmits the result to the next node in the ring.

This offload strategy has two main advantages:

1. **Computation-communication overlap**: While the smart NIC is performing the aggregation for one layer, the host CPU can proceed with computing the forward pass of the next layer. This overlap reduces idle CPU time and accelerates the training pipeline.
2. **Gradient compression**: Since deep learning models are robust to certain approximations, gradients can be compressed before transmission using techniques like quantization or sparsification. For instance, floating-point values can be compressed from 32 bits to 16 bits. Implementing such compression efficiently on a CPU is expensive, but an FPGA-based smart NIC can perform this operation through dedicated hardware, yielding both speed and energy efficiency.

Experimental results support this approach. The tests compared three configurations:

- No smart NIC (baseline).
- smart NIC without compression.
- smart NIC with gradient compression.

They observed that while the forward pass and the backward pass times remained unaffected across all setups, the exposed AllReduce time, which is the portion of gradient reduction not overlapped with computation, was significantly reduced when using smart NICs, especially with compression enabled.

In summary, the use of smart NICs (particularly FPGA-based) for gradient aggregation not only reduces network overhead but also enables overlap of critical stages in the training process, providing measurable improvements in runtime and CPU utilization.

6 Congestion Control

To explain how data moves once it enters the network, we must first recall what a switch is: a piece of hardware with a certain number of ports that can transmit data in both directions, connecting various servers or other switches to enable communication between network endpoints. However, when two packets arrive at the switch from different ports and both need to be forwarded to the same output port, one of them must be placed in a buffer. Since these buffers are not infinite, once they fill up, packet loss can occur. To reduce the number of packets waiting in queues, it's essential to start with a good topology, as a non-blocking network is less likely to experience packet congestion. However, congestion control mechanisms must also be implemented to address the issue effectively.

When two senders each transmit packets at 100Gb/s over separate links to a switch, which then needs to forward both streams over a single 100Gb/s link, the idea is to make the senders aware of the situation so they can reduce their transmission rates to 50Gb/s each. This way, the switch can forward both data streams without causing congestion.

In practice, a common approach is to use a while(true) loop that:

- measures congestion or available bandwidth,
- adjusts the sending rate accordingly.

This can be done on the sender side, where the sender waits for an acknowledgment from the switch and measures the latency at each iteration. If the latency increases, the sending rate is reduced. Alternatively, it can be handled on the switch side, where the switch simply communicates its available transmission capacity to the senders.

Congestion can be determined along three dimensions and at the moment it's not clear which is better:

- delay,
- queue size,
- packet loss.

While the rate to adopt can be determined by the sender, the switch, or the receiver, although the entity that actually takes action is always the sender.

6.1 Random Early Detection (RED)

RED is a congestion control technique in which the congestion notification is implicit. This means that once congestion is detected, some packets are simply dropped. As a

result, it is assumed that TCP is being used, since it will eventually detect the packet loss and retransmit the missing packets.

The term "Early" originates from the concept of **Early Random Drop**, which refers to dropping packets before the queue becomes full. Specifically, each arriving packet is dropped with a certain probability (depends on the queue size) as soon as the average queue length exceeds a predefined threshold. This approach helps to signal congestion early, avoiding sudden bursts of packet loss when the queue overflows.

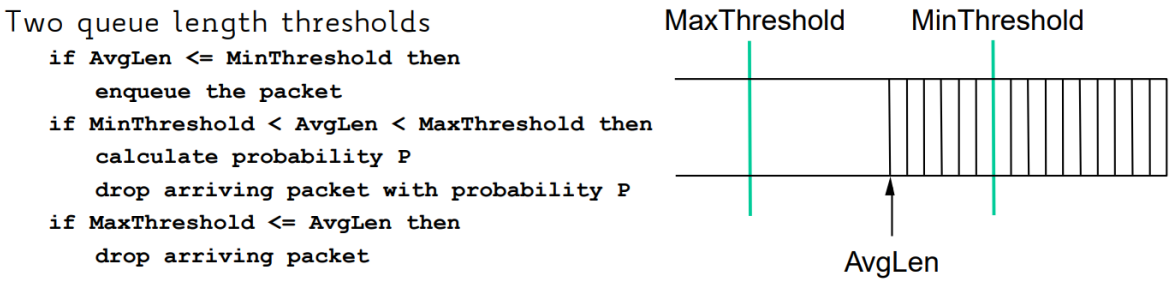

Figure 37: Random early detection implementation.

This approach is preferred over waiting for the queue to fill up and then dropping all packets, because that scenario is extreme and would take more time to react to congestion.

6.2 Explicit Congestion Notification (ECN)

A better idea, since packet loss is not desirable, is to notify the sender to slow down. This is the concept behind **ECN**, which is also used in the IP protocol. In fact, the IP header includes 2 bits (called ECN) specifically designed to signal congestion.

What happens is that when a switch detects congestion, it marks the ECN bits to signal congestion to the receiver. Then, when the receiver sends back the ack, it includes those marked bits, effectively notifying the sender. Congestion is determined by the switch through RED, but instead of dropping packets they are simply marked. Once the sender receives a marked packet, it slows down its transmission rate, regardless of how many marked packets it receives, which might be a problem.

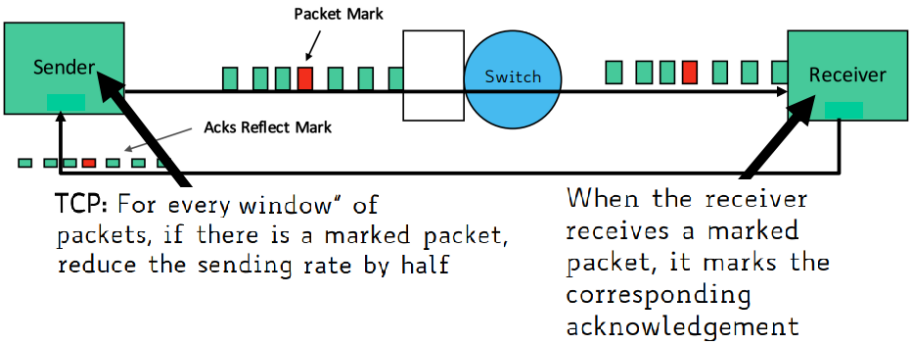

Figure 38: Explicit congestion notification process.

6.3 Data Center TCP (DCTCP)

The idea behind this technique is to use both ECN and RED, while also allowing the sender to react based on the severity of the congestion, unlike ECN. It means that the sender has to keep track of the marked packets it received. Another improvement is that with **DCTCP** marks are set based on the instantaneous queue length rather then on the average (granting a faster reaction).

6.4 Priority-based Flow Control

The techniques seen so far are based on the TCP protocol, which, as we've discussed, differs from RDMA. RDMA relies on the InfiniBand transport protocol, which assumes a lossless network by design. This assumption generally holds true when InfiniBand is used throughout all levels of the network. However, when using RoCE (RDMA over Converged Ethernet), this guarantee no longer holds by default, and additional mechanisms must be put in place to ensure that the underlying Ethernet network does not drop packets.

PFC is a technique that makes Ethernet lossless. The idea is that during communication between two switches, if a queue exceeds a certain threshold, the receiving switch sends a so-called pause frame to the transmitting switch. When the queue is cleared, the receiving switch signals the transmitter to resume transmission. A first

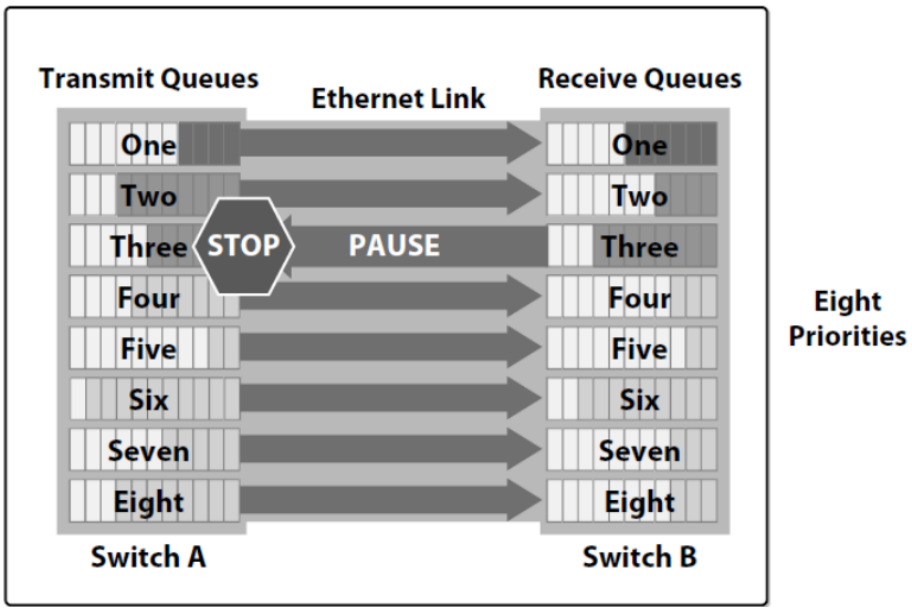

Figure 39: Priority-based flow control process.

challenge is **setting the threshold** correctly: if the queue becomes too long, it might be too late to avoid congestion; however, if the threshold is set too low, there is a risk of underutilizing the available resources. An additional problem is the **limited number of queues**. These queues, as shown in the figure, are called priority queues and are

groups of queues assigned to a single port, used to prioritize certain applications. In other words, less important queues are paused, while those of privileged applications are maintained. This is a problem because when a queue is paused, it's not paused just a single application, but most likely a group of applications that share the same queue. A further problem is the possibility of encountering **deadlocks**, for example when three switches connected in a ring request a pause from one another. Finally, another issue is **congestion spreading**, which means that if two switches forward traffic to a third switch causing congestion, the congested switch will not differentiate which of the two is contributing more and will pause both of them.

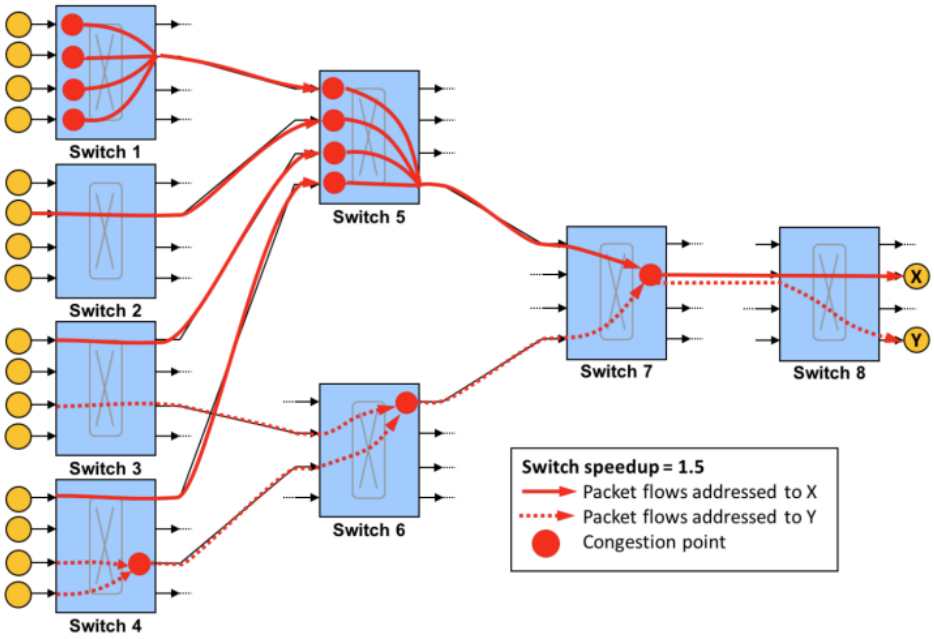

Figure 40: PFC congestion spreading.

6.5 HPCC

The idea behind **HPCC** is to use more than just a single bit to indicate congestion. The goal is to improve the convergence toward the correct transmission rate by providing more precise feedback between switches. Each switch attaches a timestamp, queue length, number of bytes, and link bandwidth to the packet, and the receiver sends all this information back to the sender within the acknowledgment. In other words, we

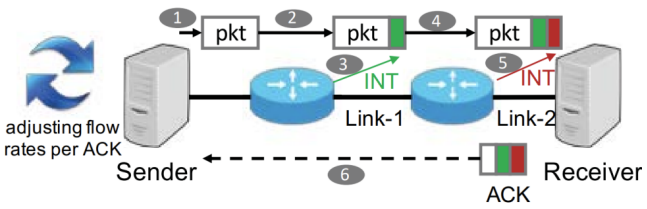

Figure 41: HPCC process.

are controlling congestion at the cost of transmitting more bytes. This means that packets shouldn't be too small for the approach to be worthwhile.

6.6 ECN vs Delay

ECN-based algorithms have some limitations, the main one being the use of only a single bit to signal congestion, which results in imprecise feedback. In theory, switches could enrich packets with more accurate information such as queue length (as seen in HPCC), but not all switches support this capability. An alternative is to handle everything directly at the hosts, for example by estimating congestion through the measurement of round-trip time (RTT), without requiring any special support from the switches. This is exactly the idea behind **Delay**.

Actually, there are pros and cons to both approaches, but a key difference lies in the fact that in ECN, the packet is marked reflecting the state of the queue at the moment it is about to leave the queue, whereas in the case of delay-based methods, the packet reflects the state of the queue at the moment it arrives at the queue.

6.7 Load Balancing

The process of **Load Balancing** aims to distribute the transmission load across multiple network paths. However, it is not capable of resolving all types of congestion. For instance, when congestion occurs at the edges of the network, packets must inevitably pass through the congested switch to reach their destination, making load balancing impracticable.

Load balancing within data centers is simpler compared to that on the Internet, because there are no autonomous systems going down, becoming unreachable, or similar issues. In general, the structure of a data center is static, and as a result, the paths between a node A and a node B are known in advance. Thus, we can just assume that each switch knows all the possible paths towards the destination.

The most efficient algorithms are often proprietary, as some companies choose not to disclose certain developments to limit competition. In general, the performance of known algorithms depends on the network topology.

Among the adopted solutions, we can identify:

- **Centralized solutions**: An external controller, which has global visibility of the network state, makes the best decisions at a given moment. However, this is difficult to implement at large scale, since there are only a few microseconds to collect the state of every switch.
- **Distributed solutions**: Each node makes its own decisions independently.
  - **In-Network**: Decisions are made by the switches.

    - **Congestion Oblivious**: Upon congestion, the path is changed randomly, without necessarily improving the situation.
    - **Congestion Aware**: Upon congestion, the path is changed with the goal of optimizing performance.
- **Host-Based**: Decisions are made by the end-hosts.
    - **Congestion Oblivious**: Upon congestion, the path is changed randomly, without any guarantee of performance improvement.
    - **Congestion Aware**: Upon congestion, the path is changed based on network conditions to select a better one.

6.7.1 In-Network Congestion Oblivious

In this context, the two main strategies are **ECMP** (Equal-Cost Multi-Path) and **Random Packet Spraying**. Let's begin with a common problem: suppose we have a tree topology and we want to send data from node A to node B, as shown in figure 42. The most efficient strategy is typically to route the packets upwards in the tree until reaching a common ancestor of both nodes, and then route downwards to reach B. This is known as up/down routing.

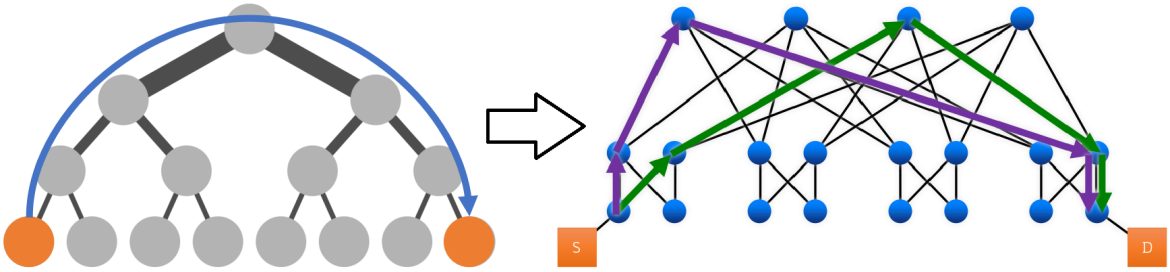

Figure 42: Up/down routing.

However, in real-world scenarios, each logical node in the tree is itself composed of multiple physical nodes (e.g., racks, switches, etc.). As a result, nodes A and B may have several common ancestors. The primary goal is to choose the path that is least congested among the available ones. It is also worth noting that there are multiple possible paths to reach the common ancestor node, but once that node is reached, there is typically only one path to node B. Therefore, the routing decision must be made before reaching the ancestor node, otherwise, it will be too late to avoid congestion.

What ECMP does is take the source and destination nodes as input, typically the source and destination IP addresses and ports in the case of TCP, and feeds them into a hash function. This function returns a value that maps to one of the available ancestor nodes. Moreover, packets belonging to the same connection are forced to follow the same path. This is crucial because routing packets 0 and 1 over different

paths could result in packet 1 arriving before packet 0. This reordering would be problematic for protocols like TCP, which expect packets to arrive in order and might incorrectly assume that packet 0 was lost.

The primary issue that can occur with ECMP is that the hash function may end up mapping a large number of flows to the same specific paths or nodes, leading to congestion. This phenomenon is known as **flow collision**. Another issue is known as link fault; in large-scale systems, it can happen that a link fails. In the case of ECMP, it takes some time to detect and recover from such a failure. During that interval, all packets routed through the failed link are lost.

An alternative to this is the so-called Random Packet Spraying, where each packet is sent along a randomly selected path. This naturally introduces the problem that packets belonging to the same connection may arrive out of order, which can be interpreted as packet loss by protocols like TCP. One possible workaround is to send all packets of the same connection along the same path, but this increases the likelihood of collisions. In practice, hybrid strategies can be designed to balance between these two extremes.

6.7.2 In-Network Congestion Aware

To achieve better performance, it is possible to select paths based on certain information. A specific example is **CONGA**, where each switch maintains a table that tracks the congestion level of every other switch. In practice, the congestion of a path is considered to be the maximum queue length along that path.

To ensure packets do not arrive out of order, CONGA uses the concept of flowlets. In practice, packets belonging to the same connection are sent along the least congested path. However, if a time gap greater than a predefined threshold δ occurs between the arrival of packet i and packet $i + 1$, it is assumed that all previous packets have likely arrived, and the switch may take the risk of changing the path. Each group of packets sent before such a pause (greater than or equal to δ) is referred to as a **flowlet**. The main issue with this approach is that setting an appropriate value for δ is not straightforward. If δ is too small, there is a risk that packets may arrive out of order; if it is too large, the same path might be used for the entire connection.

6.8 Centralized Load Balancing

An example of centralized load balancing is **Hedera**, a system that identifies large network flows by continuously monitoring edge switches to measure the number of transmitted bytes. When a flow exceeds a certain threshold, such as 10% of a host's link capacity, it is classified as "large". For small flows, ECMP is generally efficient, while for large flows, Hedera estimates bandwidth demand and attempts to intelligently allocate these flows across available paths to maximize bisection bandwidth.

Nonetheless, the system has some limitations: accurately estimating bandwidth demand is not trivial, and its responsiveness might be insufficient for short-lived or highly dynamic flows. Some flows may even start and finish before the load balancer becomes aware of them. As a result, Hedera is **only** suitable in scenarios with many large and long-lived flows, such as Big Data workloads.

6.9 Host-Based Congestion Aware

In all the solutions discussed so far, some level of support from the switches is required. This is not the case with **host-based solutions**, which shift most of the complexity to the hosts rather than the switches, and this is their main advantage.

Fundamentally, they leverage ECMP, with the difference that the hash function takes as input a defined value called entropy, which can be manipulated so that the host can choose a different path upon detecting congestion. The congestion itself is detected using ECN.

7 In-Network Compute

It refers to the idea of assigning switches, in addition to their forwarding role, a certain amount of computational load. This can be implemented using an ASIC, or through a programmable switch on which the packet processing behavior can be explicitly defined (this is the case of HPCC congestion control).

Programmable switches, much like some smart NICs, are based on an abstract concept called a **Reconfigurable Match-Action Table** (RMT). This is essentially a table where specific match conditions are defined, along with corresponding actions to take when those conditions are met. These match-action rules must be very simple, typically without loops or complex conditionals, so the flexibility of the processing is quite limited. While this approach was widely adopted in the past, it seems unlikely to remain considerable in the future.

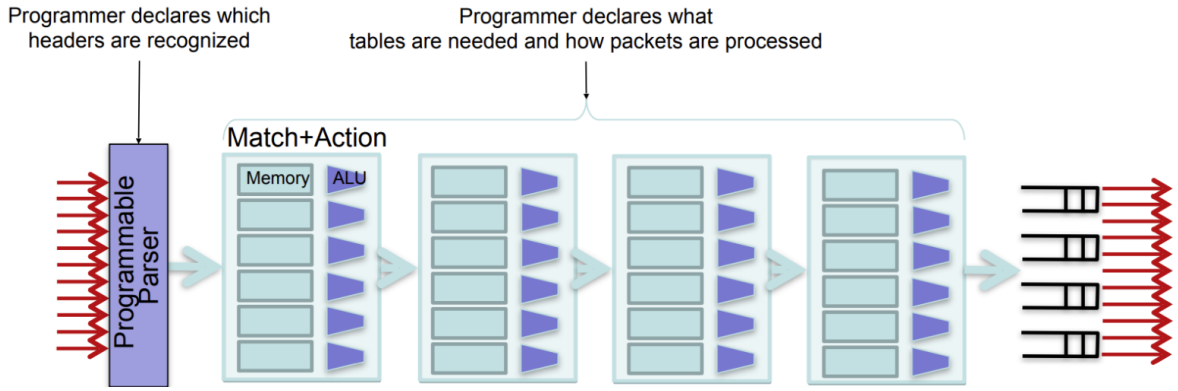

Figure 43: Reconfigurable Match-Action Table.

The most relevant use cases for In-Network Compute today are Map-Reduce and AllReduce operations.

7.1 Map Reduce Use Case

In this case we are counting word occurrences using a MapReduce approach. Which means that the computation can be divided into multiple phases: splitting the input, mapping key-value pairs, shuffling data based on keys, and finally reducing the values.

Thanks to in-network compute we can mostly accelerate the shuffling and reduction phases. Instead of letting the hosts perform the reduction, switches can intercept key-value pairs from different nodes, identify those with the same key, and perform partial summation directly in the network.

Of course switches have limited memory and cannot store large amounts of data, but in the case they are full, they can simply "spill" the rest of the computation to the next switches or, in the worst case, to the final reducer host which will have enough resources to complete the job.

This approach drastically reduces network load: data volume is reduced by up to 88%, processing time by 84%, and packet count by 80%, offering significant performance improvements for large-scale distributed computing tasks.

7.2 All Reduce Use Case

Gradient aggregation is a fundamental operation in distributed training of machine learning models. Each node, or worker, computes a local gradient vector, and these vectors must be summed to obtain a global gradient. A classic approach is to use a parameter server in which each node sends its vector to the server, which performs the summation and returns the result. However, this centralized architecture poses scalability issues.

To overcome this, the idea of in-network gradient aggregation was introduced. Instead of sending the gradients to a centralized server, the gradients are sent to the network switch, which performs the summation directly and returns the result to the workers. This method is efficient when all workers are under the same switch.

When scaling to a large number of workers (for example 100,000), a single switch is no longer sufficient. The solution is to build a tree topology where servers are the leaves and switches are the intermediate nodes. Each server sends its gradient to its parent switch, which performs a partial summation and forwards the result up the tree. This continues until the data reaches the root, where the final aggregation is performed. The root then broadcasts the fully aggregated gradient back to the servers along the same tree.

Figure 44: In-network AllReduce.

This tree-based in-network aggregation approach has several key benefits. Each host sends only one vector of size n, where n is the size of the gradient vector. The entire process completes in one communication round (still referred to the hosts). This is an optimal strategy compared to traditional AllReduce algorithms, which typically require at least two rounds and more total data movement.

In practice, this technique has been implemented by several companies, often using ASICs embedded in switches. However, the approach presents a number of challenges. The first one is about **load balancing**, because the tree structure is fixed and each

Algorithm	Comm. Volume	# Steps
Optimal	n	1
Parameter Server	max(n, (pn)/(K))	1
Naive Allreduce	((p-1)n)	1
Ring	$2n$	2(p-1)
Bw. Optimal Rec. Doub.	2n	2*log2(p)
Lat. Optimal Rec. Doub.	log2(p)*n	log2(p)
In-network Allreduce	n	1

Figure 45: AllReduce algorithms.

node expects data from predefined children. If a link becomes congested, the system cannot dynamically reroute traffic without breaking this expectation. Thus, congestion can severely impact performance, worse than traditional host-based methods. One proposed solution uses a timer-based strategy, so that the switch waits for a small interval after receiving data from one child, hoping to receive the remaining gradients. After waiting, it aggregates what it received during this window and forwards it. This forms a dynamic, congestion-aware aggregation tree in a best-effort manner.

The second issue is about **floating point addition**. The sum $(a+b)+c$ may differ from $a + (b + c)$, due to cancellation and rounding errors. In this context, the order in which a switch receives and aggregates vectors affects the final result. For example, receiving vectors in the order A, C, B, D results in computation (((A+C)+B)+D), which may differ numerically from other orders. A typical solution is to buffer all inputs and perform the summation in a fixed order, but this requires large memory, which switches typically lack. This remains an open research problem.

The third problem concerns the lack of **floating-point support**. Switches are optimized for throughput and latency, and generally do not have floating-point units, since floating-point operations are costly at both of them. Therefore, aggregation must use fixed-point arithmetic. In such representation, some bits are reserved for the integer part and others for the fractional part, using a positive power of two for the integer and a negative one for the fractional part. Although less precise, fixed-point math is often sufficient for deep learning workloads.

The last issue is about **packet loss**. If a packet is lost, some resiliency mechanism is needed. Since switches cannot run full TCP stacks, an idea is to implement a simple retransmission protocol following the condition: if no acknowledgment is received within a timeout, the sender retries. However, it becomes difficult to determine whether an incoming packet is a new message or a retransmission of a previously received one. Designing reliable aggregation protocols in the switch fabric is therefore complex.