Computer Science Course Notes — 1st Semester

Dario Loi

Contents

Big Data Computing

First Lecture

Classroom: phdlyrl

Second Lecture

What's Big Data?

It is sometimes used as a buzzword, but it describes an actual phenomenon.

5 V's:

1. Value: Extracting knowledge from data.
2. Volume: The amount of data. (Measuring in terabytes, petabytes, exabytes)
3. Variety: Different formats, structured (SQL tables) and Unstructured (text, images, videos).
4. Velocity: The speed at which data is generated. (usually in real-time, you can't parse it all)
5. Veracity: The quality of the data. (Is it reliable?)

Scaling

When scaling, you have two options:

- Scaling *up*, you buy better hardware for your monolithic machine, however, moore's law is slowing down, and you will get diminishing returns, in general, you will not be able to keep up with increases in Volume/Velocity.
- Scaling *out*, you buy more machines, and distribute the workload, this is the way to go for big data, since this allows for exponential growth.

Scaling *out*, or *horizontal scaling*, also allows you for more flexibility, as once you deal with the orchestration of the workload, you can dynamically increase/reduce the number of allocated machines depending on a rolling cost/benefit analysis.

Network Bottlenecks

A bottleneck during a request is generated when:

- There is a skew in the ratio of communication/computation
- There is a non-overlappable part of communication/computation

Third Lecture

In this lecture, we will take a look at a use case of Big Data, *Distributed Deep Learning*.

Naturally, Deep Learning performs better with more data, this is actually an empirically derived law that has come up in recent OpenAI research: exponentially more data leads to a linear increase in model performance, irrespective of the model architecture (assuming a reasonable architecture).

How does a DNN work

To put it simply, we want to build a function estimator over a dataset D:

$$\hat{f} : D \rightarrow Y$$

Such that:

$$\hat{f}(x) \approx y$$

Where x is the input data, and y is the output data.

To do this, we use iterative optimization, where we start with a random function, and we iteratively update it to minimize the error between the output of the function and the actual output. We use the data from the dataset to *fit* our model, under an *inductive* bias, which is an assumption that the *test* data is similarly distributed (so that our fit will generalize).

Forward and Backward passes

Our iterative fitting process can be split in two passes:

1. Forward pass, where we compute the output of the model given the input.
2. Backward pass, where we compute the gradient of the error with respect to the model parameters.

We use the obtained gradient to refine the model parameters, and we repeat the process until we reach a stopping criterion (which indicates that the model has converged).

In general, most of these operations are matrix operations, which can be parallelized on the GPU in an efficient manner.

Parallelism Models

There are three orthogonal parallelism directions in which we can distribute the workload:

1. Data Parallelism
2. Pipeline Parallelism
3. Model Parallelism

The last two together are also known as *Model Parallelism.*

Data Parallelism

The simplest one is *Data Parallelism.* We split the data in n subsets, and we train n models in parallel, each on a different subset. We then perform *gradient aggregation* across these sets, and we update the model parameters.

In this way all the models are updated in parallel, and the only bottleneck is the gradient aggregation.

Gradient aggregation is *quite* costly, it contributes to a significant portion (20% to 50%) of total training time, with the portion increasing as the available bandwidth decreases.

Naturally, this means that this operation has been the subject of heavy optimization, algorithmically and infrastructurally.

Parameter Servers

An initial solution was to use a *Parameter Server*, which is a server that holds the model parameters, and is responsible for the aggregation of the gradients. This does *not* scale well, as it introduces a single point of failure, and a bottleneck in the network.

Splitting the Aggregation

We can split the aggregation into K servers, which will each aggregate a subset of the gradients, and then we will aggregate the results of these servers. This is a more scalable solution, but it still has a bottleneck in the aggregation.

Naturally, as long as K is small, this is a good solution, but as K grows, the aggregation time will grow linearly, and the bottleneck will be reached.

Cost Model

A cost model on a K-PS (Parameter Server) system is:

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

Where n is the number of bytes in the gradient vector, p is the number of GPUs/workers, k is the number of servers, and β is the bandwidth of the network.

Data Parallelism Recap

In data parallelism, each worker must keep a copy of the model, therefore you can also encounter out-of-memory errors, as the model grows in size.

Data parallelism is simple and easy to implement, but has limited potential to be scalable.

Replication of the model is also a waste of resources, as we are just keeping copies around, and we are not using them. It should only be used as a last resort if we have more resources than we can use after applying the other parallelism models, and we still want idle time to be minimized.

Model Parallelism

In order to train models that do *not* fit into a single GPU, we can use *Model Parallelism*. in essence, model parallelism is the process of splitting the model into k parts, and training each part on a different GPU.

Now, the bottleneck is the GPU-to-GPU communication, since we have a sequential dependency between the parts of the model (each part depends on the output of the previous part). This happens in both the forward and backward passes.

Pipelining

Naturally, you can reduce the *bubble overhead* by pipelining the computation, so that the forward pass of the next part can start before the backward pass of the previous part has finished. The same concept can be applied on subsections of the forward/backward passes that use layers that are distributed across a network.

Operator Parallelism

In the case of *Operator Parallelism*, we split the model into operators, and we distribute the operators across the network. This is a more fine-grained approach, and it is usually used in conjunction with the other two parallelism models.

Here, the bottleneck is the communication between the operators, and the synchronization of the operators, however, we reduce the idle time of GPUs drastically.

Hierarchical Parallelism

Naturally, *Operator Parallelism* is applied on GPUs that are on the same node, so that the available bandwidth is maximized. Whereas *Model Parallelism* is applied on GPUs that are on different nodes, where communication is more costly.

This allows us to maximize the utilization of the available resources, and minimize the communication overhead.

Hybrid Parallelism

Hybrid parallelism (also called 3D-Parallelism), combines all three techniques, and offers maximum scalability and potential for resource utilization.

In general, one must be working with a sufficient quantity of data to *justify* the use of these techniques, as the overhead of the parallelism models can be significant, and can outweigh the benefits of the parallelism.

Challenges

Compute-Communication Overlap

When employing hybrid-parallelism, we perform computations and communications in a very fine-grained manner. The reason is that we want to perform gradient aggregation after each layer, so that we minimize the ***bubbling*** in our pipeline (we don't want to keep anyone waiting).

We want to maximize the overlap in windows of computation and communication, so that we can keep the GPUs and network as busy as possible.

Gradient Compression

Since DL is a stochastic process, one can afford to have some noise in the gradients, and still converge to a good solution. This allows us to compress the gradients, and reduce the communication overhead.

We have various techniques:

1. Quantization: Reducing the precision of the gradients.
2. Rank Decomposition: Decomposing the gradients into a low-rank matrix.
3. Sparsification: Sample the top k gradients, and send only those.

In sparsification, we can use various techniques to select the top k gradients, such as:

1. Top-k: Select the top k gradients by magnitude.
2. Thresholding: Select the gradients that are above a certain threshold λ.

There are theoretical proofs that show that asymptotically, model convergence is not affected by gradient sparsification, as long as the sparsification is done in a *smart* way.

Compression Tradeoff By compressing gradients, we lower the cost of a single iteration, but we increase the number of iterations needed to converge. This is a tradeoff that must be taken into account when designing a distributed deep learning system.

We have a real speed up if:

$$T_{comp} \cdot I_{comp} < T_{orig} \cdot I_{orig}$$

Aggregation Consistency

We observe that gradient accumulation acts as a *barrier*, which means that if a single GPU is, for some reason, considerably slower than the others, it will slow down the whole process. The probability of this happening is significant in large-scale systems, and it is a problem that must be addressed.

Possible solutions include: * Gradient Accumulation: Accumulate the gradients over multiple iterations, and then send them. * Ignoring Slow Workers: Ignore the gradients from slow workers, and only aggregate the gradients that you have currently available after a certain time.

Reduce Operations

We can use smart reduction techniques common in IPC (Inter-Process Communication) to reduce the number of operations needed to aggregate the gradients.

We keep a table of operations and total volume of communication:

Operation	Volume	Steps
Ideal	n	1
Parameter Server	$max(n, \frac{pn}{k})$	1
Näive AllReduce	$(p-1) \cdot n$	1
Ring AllReduce	$2 \cdot N$	$2(p-1)$
Bandwidth-optimal recursive doubling	$2 \cdot N$	$2\log_2(p)$

We explain each:

- Ideal: The ideal case, where we have a perfect system that can aggregate the gradients in a single operation.
- Parameter Server: The case where we have a parameter server, and we have to send the gradients to the server, and then the server sends the aggregated gradients back.
- Näive AllReduce: The case where we have a simple all-reduce operation, where each worker sends the gradients to all the other workers, and then each worker aggregates the gradients.
- AllReduce (ReduceScatter-AllGather): The case where we have a more sophisticated all-reduce operation, we scatter p subsets of the gradients to p workers, each worker aggregates the subset, and then we gather the results across all workers.

Ring AllReduce

We go in depth into the AllReduce operation. First, we build a logical ring topology of N workers, such that each one is connected to the next one, and the last one is connected to the first one.

We take our vector g, we divide it into N parts, and we send each part to the next worker. Each worker then aggregates the received part with its own part, and then sends the result to the next worker.

This results in a full aggregation of the gradients in $(p-1)$ steps, another full rotation is then performed to gather the results, and we have our aggregated gradients.

The communication volume is:

$$2 \cdot \frac{pn}{k} \approx 2n$$

Which is a 2-approximation of the ideal case.

We use a Ring AllReduce instead of a Tree AllReduce, since the Ring AllReduce is capable of only sending parts of the gradients, and not the full gradients, which can be problematic in the case of large models.

Bandwidth-optimal recursive doubling

A variation of the Ring AllReduce is the Bandwidth-optimal recursive doubling, this allows us to perform the aggregation in a logarithmic number of steps, and it is optimal in terms of bandwidth.

Latency-optimal recursive doubling

Using the same binary-tree like topology as the Bandwidth-optimal recursive doubling, we can build a latency-optimal recursive doubling, which is optimal in terms of latency, by sending the entire gradient vector in a single step.

Fourth Lecture

This lecture went into a general overview of the CUDA GPGPU programming model.

An NVidia GPU is essentially a collection of *streaming multi-processors* (SMs), each of which is a collection of individual *threads*, these threads are grouped into *warps*, which are the smallest unit of execution on the GPU.

Warps are made up of 32 threads, and they are executed in *SIMD* (Single Instruction, Multiple Data) fashion, which means that all threads in a warp execute the same instruction at the same time.

Fifth Lecture

Today we will finish up the contents of the fourth lecture.

We started discussing on how GPUs are better for big data workloads. The main reason is that their specialization allows for better efficiency in throughput-oriented workloads.

The concept of *warps* was introduced, as well as the notion that control flow is incredibly disruptive to kernel throughput, as it can cause *divergence* in the warps.

GPU Interconnection

As mentioned before in the Deep Learning case study lecture, these GPUs are connected in a messy, non-uniform way, which can cause bottlenecks in the communication between the GPUs.

Tensor Processing Units

GPU hardware is still *too* general for workloads which only need linear algebra operations, such as fully connected layers in a neural network. They suffer from the Von Neumann Bottleneck, meaning that most of the cases, the processing units are waiting for data to be fetched from memory.

Accessing memory is impactful not only in terms of latency, but also in terms of power consumption, as a memory access can cost as much as 650x the cost of a FLop.

This is why Google developed the TPU, a specialized hardware accelerator, largely based on the concept of a systolic array, which is a type of parallel computing architecture.

A TPU is a collection of Tensor Cores, each posses a set of Matrix Multiplication Units(MXUs), Scalar Units, and high-bandwidth local memory.

MXUs are *not* designed according to the Von Neumann architecture, they are often termed as a spatial/dataflow architecture.

Dataflow Architecture

A dataflow architecture is composed by arranging simple circuits that perform elementary arithmetic in a 2D-grid, the grid possesses no memory, and the data *flows* through the grid, hence the name.

Matrix Multiplication

We give a recap of matrix multiplication, basically:

$$\underbrace{\begin{bmatrix} A & B \\ C & D \\ E & F \end{bmatrix}}_{M} \times \underbrace{\begin{bmatrix} G \\ H \end{bmatrix}}_{U} = \underbrace{\begin{bmatrix} AG+BH \\ CG+DH \\ EG+FH \end{bmatrix}}_{V}$$

Where each letter can be seen as an individual *tile* of the matrix.

We observe how, for example, the G tile is multiplied by the first column of M to result in the first column of U, same for the H tile. in fact, each row of the V vector is the result of the multiplication of the respective M row by the column of U.

Therefore, we can *flow* the data through the systolic array, and perform a final aggregation of the results to obtain the final result.

This can be done *without* the need for registers or memory, as the computation is done on the fly at each clock cycle.

TPU Interconnections

Interconnecting TPUs is usually done through a 3D torus network, which is a simple and efficient way to connect the TPUs.

Performance Claims

Google claims that TPUs are in general faster than the comparable NVidia GPU solutions, however, other studies have shown that this is not always the case, and that the performance of the TPU is highly dependent on the workload.

In general it is still ambiguous whether TPUs are a better solution than GPUs, and it is still an open research question. The point of the lecture is more to convey the concept of the dataflow architecture, rather than its performance (which may change in the future).

Neuromorphic Computing

Another recent attempt at supplanting the Von Neumann architecture is the concept of Neuromorphic Computing, which is a type of computing that is inspired by the human brain.

Spike trains are introduced as an input, and the synthetic neural network prodcues a spike train as an output, this is done by using a network of *spiking neurons*. In theory, since this replicates the structure of the brain, it should also be capable of possessing memory capabilities.

Network Topology Design

We now move on to talk about ways to design interconnection between nodes of a big data computing system.

GPUs are connected by *switches*, which act as crossroads for the data. These can connect at most r numbers of nodes, this is called the switch's *radix*.

Topologies are *regular* if they are a regular graph (rings), *irregular* otherwise.

Hop count is the number of switches that a message must pass through to reach its destination.

The maximum hop count attainable in a network is the *diameter* of the network.

A network is *blocking* if two nodes cannot be connected by two disjoint paths. If there is this possibility, the network is *non-blocking*.

A network is *direct* if for all the switches, there is at least a connected *node*, if they are *indirect*, there may be some switches that are only connected to other switches.

Bisection Cut

In order to obtain a measure of a network's efficiency, we can perform a *bisection cut* analysis. This is done by cutting the network across the smalles possible cut that divides the nodes into *nearly* equal parts.

The *bisection bandwidth* is the bandwidth of the smallest cut that divides the network into two equal parts.

This is calulated as:

$$\text{Bisection Bandwidth} := \text{Number of Cuts} \cdot \text{Cut Links Bandwidth}$$

It is simply the multiplication between the number of cut links, and the bandwidth across the cut.

We produce a table for these metrics on different topologies:

Topology	Diameter	Bisection Cut
Chain	N	1
Ring	$N/2$	2
Mesh	$2(\sqrt{N}-1)$	$\sqrt{N}$
Torus	$\sqrt{N}$	$2\sqrt{N}$
Trees	1	$2 \cdot \log_{r-1}(N)$

Trees

Trees are a classical computer science structure that usually gives some form of logarithmic improvement over a metric when implemented as a data structure. Here we can lower the bisection cut cardinality to a logarithmic funciton of the number of nodes, which is a significant improvement over the other topologies.

However, we are clearly bottlenecking the system when we have to pass through the root of the tree, as the root is a single point of failure. The same reasoning applies inductively where lower nodes are responsible for the inter-communication of all the children.

Fat Trees

We can solve this through a *fat tree*, a data structure where at each higher level we have a double number of links than the previous level, this allows the higher nodes to account for the increased responsibility of the lower nodes.

Fat trees metrics:

- r bottom level switch radix
- n number of servers/nodes
- Diameter: $2log_{r/2}(n)$
- Bisection cut: $n/2$

A fat tree is a *non-blocking* network by construction, since we explicitly add a number of links that is double the number of the previous level.

In practice, we want to use switches that have the same radix level at each layer. This means that we need to aggregate multiple switches in a mesh network to form a single *virtual* switch node.

This is usually called a "folded CLOS network".

Blocking Fat Trees

We can save some money by using a *blocking* fat tree, where we use a radix r at the bottom level, and a radix $r/2$ at the upper level. This is a blocking network, but it is still a good solution for most cases.

The ratio between the lower and upper radix is known as the *blocking ratio*, for our example it's a 2:1 ratio.

Dragonfly

Another popular state-of-the-art topology is the *Dragonfly* network, which is a network that is composed of a number of *groups*, each group is a *clique* (a fully connected graph), and each group is connected to a number of other groups.

This allows for extremely low hop counts, and high bisection bandwidth.

For a fixed radix r, dragonfly networks can connect much more nodes than a fat tree. Diameters are also kept smaller. Naturally, this means that with similar requirements, the network will be cheaper to build.

Disadvantages include the lack of a guarantee of full bandwidth (blocking), expansion is more difficult, as is load balancing. A dragonfly topology also produces more loops, and therefore deadlocks.

Dragonfly+

Dragonfly+ is a slight variation where you have a fully connected *core* that connects roots of various trees that live at the edge of the network. The trees on the edge can

be fat trees, or blocking fat trees with various out-degrees.

Hamming Meshes

Hamming meshes are a type of architecture specific to deep learning workloads (where we employ three-dimensional parallelism). Here we wish to have:

1. A cheap architecture, such as a toroidal network
2. To still preserve some measure of speed for communications across the network

We solve this by essentially first producing a toroidal mesh, and then by interconnecting the edges (unwrapping the cylinder into a rectangle over an arbitrary cut) with fat trees.

In this way we create some sort of *information highway* that allows us to perform eventual edge-to-edge communications in a logarithmic number of hops.

Sixth Lecture

As a recap, in the previous lectures we have seen:

1. Introduction to Big Data (Lecture 1 and 2)
2. Overview of Distributed Deep Learning (Lecture 3)
3. NVidia GPU Architecture (Lecture 4)
4. Network Topologies and TPUs (Lecture 5)

Today, we discuss the protocols utilized to send and receive data across the network. So far in our computer science courses, we have encountered *TCP* and *UDP*, which are the most common protocols used in the internet.

These protocols are *not good enough* for moving these amount of data at these bandwidths.

Some numbers:

- Each server can inject from 400 Gb/s to 1.6 Tb/s of data
- Assuming 1500 bytes packets, we are talking of a range of about 30 to 130 million packets per second (mpps)
- This means one packet every 8 to 30 ns!

Can a standard operating system and network stack keep up with this?

Concerns about network performance will be explored in the next lecture, for now, we will focus about the endpoints.

North-south vs East-west traffic

Assume we have a *rack* of servers, each rack has an ingress/egress route through a switch. The switches are connected to a Data Center Network (DCN), which is arranged in an arbitrary topology and is finally connected to the outside world (internet).

> Traffic that goes from the servers towards the outside world is called *north-south* traffic.
>
> Traffic that goes from server to server is called *east-west* traffic.

North-South traffic can be perfectly accomodated by the standard TCP/IP stack, as it is a *bursty* traffic, and it is not time-sensitive.

East-West traffic, on the other hand, is a *streaming* traffic, and it is time-sensitive. This is the traffic that we are interested in optimizing.

Brief TCP/IP recap

The internet stack is composed of 7 Layers:

1. Application
2. Presentation
3. Session
4. Transport
5. Network
6. Data Link
7. Physical

We group 1 to 3 as the *Application Layer*, 4 as the *Transport Layer*, 5 as the *Network Layer*, and 6 and 7 as the *Link Layer*.

Each of these layers introduces an *header*, which in turn introduces overhead in the size of the packet, which reduces the effective bandwidth of the network.

From an application perspective, this works through sockets:

```
// request a socket from the OS
int sock = socket(AF_INET, SOCK_STREAM, 0);
struct sockaddr_in serv_addr;

// [...]

// set up the server address
serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons(PORT);

// [...] Client code

//connect to the server
connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr));

//send data
send(sock, buffer, strlen(buffer), 0);
```

```
//receive a reply
recv(sock, buffer, 1024, 0);
```

This C userland code is translated into a series of system calls, which use kernel space code to send and receive data. This *also* introduces overhead, as the kernel has to copy the data from userland to kernel space, and then back to the network card.

Historically, sockets are *forty* years old, the same age as the first mario bros game, or ARPANET. They were designed for a different era, and they are not well suited for the high-bandwidth, low-latency requirements of modern data centers.

In true UNIX philosophy, a socket is *just a file descriptor*, applications are therefore shielded away from managing anything but the *what to send* and *where to receive*, through two functions:

```
send(int socketfd, void* buffer, size_t length, int flags);
recv(int socketfd, void* buffer, size_t length, int flags);
```

Sockets were not designed for multi-threading, and therefore need a further layer of abstraction to be used in a multi-threaded environment to prevent data races.

Now, the questions are:

- What happens after calling `send` and `recv`
- Who is running the TCP state machine
- Who is managing the TCP window and retransmission
- Etc ...

The answer is probably the Operating System. In UNIX (the only sane operating system), the *kernel* is responsible for these tasks.

But this is *not* the only way to do things, as we previously anticipated, interacting with the kernel introduces overhead, and we can do better. Operating Systems run on the same hardware that user code runs on, for a data center, this means that CPU burst time that could be used for computation is instead used for managing the network stack.

In Linux, a single packed is defined by the `struct sk_buff` structure, which is roughly ~ 200 bytes in size. For packets that are lower than 200 bytes, this is a significant overhead.

Research has shown that 63% of CPU usage during the processing of a 64 byte packet is `sk_buff`-related. Furthermore, for a traffic of around 10 Gb/s, we need about 14.8 million system call procedures *per second*.

Overcoming the overhead

There have been a number of solutions to overcome these limitations:

1. Move functionalities to the NIC hardware (DMA engines, interrupt coalescing, scatter/gather, TCP offloading, etc...), using specialized network accelerators.
2. Exploit multicore CPUs to enable parallel packet processing.

3. Bypassing the OS and implement most of the processing in user-space.

TCP offloading consists in the implementation of the TCP stack directly in the Network Interface (NIC). This is done using an ASIC, an integrated circuit specifically designed for this.

This solution is becoming increasingly outdated:

- Moore's law means that after a couple of years, any consumer grade CPU will be able to outperform the ASIC.
- Offloading to the NIC still does not solve the tremendous overhead of the TCP protocol.
- Programming microcontrollers/hardware is *hard*.

This means that, since the concept of a data center was just starting when this concept was introduced, it never took off properly and only garnered a very limited market.

Phases of Moore's law

A brief segway into hardware evolutionary trends: we can identify three phases in the evolution of hardware:

1. When Moore's Law was in full effect (pre 2005)
2. When frequency scaling slowed down, *manycore* era (2005 - 2015)
3. Slow down of performance scaling w.r.t demand increase, scaling obtained through *specialization* (ASICs, GPUs, TPUs, etc ...) (2015 - present)

Manycore Network Scalability

Assume that:

1. The number of packets is increasing
2. The number of cores is increasing

The situation seems ideal, we can use our additional hardware to meet our demands! However, the handling of multicore programs mean that a series of issues arise:

- Synchronization over data structures.
- Lock contention.
- Cache Pollution/False Sharing.

Additional problems specific to socket handling also arise:

> Who is going to handle the interrupts?

Ideally, you would want the same core that processed the packet to handle the interrupts, to improve locality. This is not trivial to do in a real time environment.

Userspace Networking

In the Linux Kernel, which is the most widespread one, code customized for a specific use-case is looked down upon, since the kernel should be as general as possible. We

also discussed the overhead introduced by system calls at a length during this lecture.

The obvious solution to our problem is:

1. Implement a custom protocol with less overhead than TCP.
2. Implement it in user-space, so we can avoid the overhead of the kernel.

An early example of this is the *DPDK* (Data Plane Development Kit), which is a set of libraries and drivers for fast packet processing. This was introduced by Intel, and then open sourced.

RDMA

Another example is *RDMA* (Remote Direct Memory Access), which is a protocol that allows a computer to access the memory of another computer without involving the operating system of either computer.

A brief history of RDMA:

- 1980s - 1990s: Various labs such as Cornell, Berkeley, HP Labs wondered on how to build cheap supercomputers by connecting swarms of workstations. The goal was the integration of CPUs via the network as efficiently as possible.
- 1990s - 2015 (circa): adoption of RDMA in Supercomputers, Data centers still stuck on TCP/IP.

  The difference between a Supercomputer and a Data Center is now blurry. One could argue that in Supercomputers the priority is *processing and computation*, whereas in a Data Center the priority is *storage and retrieval*, this was true originally when the main task of Data Centers was to serve web pages, but now that they are used for Machine Learning, the priorities coalesced.

- 2015 - Today: As anticipated, RDMA became widely used in both Supercomputers and Data Centers. Around 70% of traffic inside Microsoft Azure is RDMA-based.

So, What is It?

RDMA is a networking *technology* to enable high-performance, low-latency network operations.

It is not TCP+Socket, it has *its own* programming API, abstractions and protocols.

In general, overhead with RDMA is *stable* when graphed as a function of message size. This means that the overhead of RDMA is practically none, this is reflected with a super low CPU usage when using RDMA.

A practical example: We want to sort 12.8 TB of data on 128 machines:

- 100 Gbps network
- 4 x NVMe devices (source and sink)

- Apache Spark

This workload was transferred from TCP to RDMA, the network usage went from 10 GBps to around 70 GBps, which reflected in a 6.5x reduction of the time needed to complete the task.

Workload transfer was performed transparently from the code by substituting the protocol underneath the Apache Spark code.

How does it work?

The key ideas behind RDMA are:

1. User-space Networking: the process manages its network resources
2. Kernel Bypass: NIC/hardware can be accessed directly from user-space

A more in-depth explanation of the inner workings of RDMA will be provided in the next lecture

Seventh Lecture

In the previous lecture, we discussed the limitations of the TCP/IP stack, and we introduced RDMA as a solution to these limitations.

RDMA Working Principles

RDMA, as an API, has a philosophy that resembles other high-throughput APIs, such as OpenGL/Vulkan. We split the operations in two groups:

1. Control Operations: Operations that are used to set up the environment for the data transfer.
2. Data Operations: Operations that are used to transfer the data.

Control Operations essentially perform buffer allocation and handshaking between the two endpoints, they interface with the kernel and are possibly very slow/latency prone. They only need to be performed *once*.

Data Operations are the actual data transfer operations, they are performed in user-space, and they are very fast, as they bypass the kernel.

RDMA Workflow

Let's explore the Control/Data flow of an RDMA operation:

1. Allocate Memory Buffers and register them with the NIC. The Kernel performs security checks, page table translation and memory pinning.
2. Allocate Data & Control Queues. Here the kernel is also responsible for security checks and memory mapping.
3. `recv` a amessage:
    1. Tell the kernel to prepare a registered buffer for message retrieval.

   2. Write the `recv` request on a memory-mapped control queue
   3. NIC writes the message directly to the buffer, bypassing the kernel.
   4. The NIC writes a completion message to the completion queue.
   5. The user-space application reads the completion queue to know that the message has been received.
4. Close the connection. Disconnection request pushed/received on the control queue.

Out-of Band Initialization

RDMA might utilize out-of-band operations, that is, operations that rely on traditional protocols (TCP/IP) in order to ensure that the connection is correctly setup.

Similarly to how the kernel calls are used to setup the environment for the RDMA operations, the out-of-band operations are used to setup the environment for the RDMA operations. We only pay the cost of these operations initially, and then we can perform the fast RDMA operations.

Required Abstractions

We need the following objects:

1. Memory Buffer: A buffer that is allocated in user-space, and registered with the NIC.
2. Connection `send`/`recv` Queues: A Queue Pair (QP), representing a connection between two endpoints. It sends metadata about the connection, and the NIC uses it to send/receive messages.
3. Control Queues:
   1. Network control queue: `connect`/`disconnect` requests, NIC up/down
   2. I/O event queue: Completion events, errors, etc...

Network I/O happens by posting work requests (WRs) to the QP (essentially commands).

| RDMA
| Socket (TCP/IP) | |
—————————————————————————————-
|
——-
| | Buffers to/from which we want to `recv`/`send` data must be preregistered | Every `send`/`recv` can be done on a different buffer without the need for registration | | They can be re-used multiple times, the NIC knows where to write/read data | For every `send`/`recv` we must pay a "control" overhead at each step in order to specify the buffer |

One-Sided Operations

Another optimization is the removal of the two-sided requirement for handshaking in order to setup the QP.

Imagine that in a data center context, all the buffers are known up front to *everyone.*

A client/peer can then initiate a transfer by itself, without the need for the other peer to be ready.
* An *RDMA WRITE/READ* can be initiated by anyone just by specifying the `from`/`to` buffers. This reduces the amount of network traffic and "server" involvement, which leads to an even greater performance gain.
In general, `send`/`recv` still *look* like TCP/IP (but are already much faster), whereas `write`/`read` follow the original RDMA philosophy of making network traffic looking like local memory access.
Real RDMA Interfaces
So, in practice, what is used to substitute sockets?

* `libibverbs`: The Infiniband Verbs library, which is the most common RDMA library. Integrated in the linux kernel since 2005. * Lots of propietary options, such as EFA, Slingshot, etc... * `libfabric`: A more high-level abstraction that can be used to interface with multiple RDMA interfaces. The underlying primitives (QPs, control queues, etc...) are the same, but the API is different. ### RDMA Network Protocols The most popular protocol, used in the majority of RDMA implementations, is *Infiniband*. Infiniband is a high-speed network technology that is used in supercomputers and data centers. It is an open standard specified in a ~400 page document.

The protocol was maintained by InfiniBand Trade Association (IBTA), but it is mostly pushed by Mellanox (now acquired by NVidia).

Another protocol is iWarp, which is basically RDMA over TCP/IP. This sounds weird, as we are trying to avoid TCP/IP, but it is used in some very special edge cases, such as long range connections between RDMA devices. iWarp never took off and is never used in data centers.

Infiniband Transport Modes

Sicne InfiniBand does *not* use TCP, it has different modes that provide different level of guarantees:

1. Reliable Connected (RC): One QP per connection, reliability using ACKs. 2. Unreliable Connected (UC): One QP per connection, no reliability (you should have lossless networks, which we will explore in the next lecture). 3. Unreliable Datagram (UD): One QP for multiple connections (it scales better), no reliability.
Not all verbs can be used in each transport mode:

Mode	SEND/RECV	WRITE	READ	WQE header
RC	Yes	Yes	Yes	36 B
UC	Yes	Yes	No	36 B
UD	Yes	No	No	68 B

RoCE

RoCE (pronounced "rocky") spawns from the NVidia monopoly over the Infiniband hardware, and the desire to use RDMA over Ethernet. RoCE is a protocol that encapsulates Infiniband packets into Ethernet packets, and it is used to connect Infiniband devices over Ethernet.

We have two versions:

1. RoCE v1: Encapsulates Infiniband packets into Ethernet packets, and sends them over the network.
2. RoCE v2: Also uses UDP/IP, to allow for routing.

RoCE v2 is the main solution whenever the network is not a dedicated Infiniband network.

To properly implement RoCE, we need to have a lossless network, as Infiniband is a lossless network, and we need to ensure that the packets are not dropped. This requires switches that support local congestion control, and that can handle the increased traffic.

Software RDMA Stacks

There are also an existing number of software implementations of RDMA, which do not require a NIC. This makes it *look* like you have an RDMA system if you have none. This is useful for debugging and Datacenter to Datacenter communication.

Another possible application is the gradual rollout of RDMA in an existing datacenter, aka *Incremental Deployment*.

Two of the implementations are:

- SoftiWarp: A software implementation of iWarp.
- SoftRoCE: A software implementation of RoCE.

SRD

AWS (Amazon) runs on an Ethernet network. A propietary protocol (SRD) was chosen instead of RoCE, as it is more efficient for their use-case.

The main RoCEv2 limitations that spawned the development of SRD are:

- Issues with congestion control
- No native load balancer

Moreover, Amazon prefers to have out-of-order packet delivery and less Queue Pairs (to scale better).

SmartNICs

Another way to reduce latency is to push more of the computation towards the edge of the machine, letting the NIC perform some of the computation. This also increases the overlap of computation and communication so that the CPU bursts can be used for different tasks

SmartNICs are also known as DPU, IPU, etc...

On-path vs off-path

An On-path Smart NiC is, as the name says, *on the path* of the data, meaning that the cores receive the data directly and then send it to the traffic manager.

Off-path is a different solution in which the data NiC can decide to route the data towards the cores or have it egress directly towards the receiver.

An example of off-path DPU is NVidia Bluefield. This hardware allows for the computation of MPI primitives (Scatter/Gather/AllReduce) without crossing the PCIe, which is very valuable.

Common Use Cases: Shuffle Suppose we have an input, for example a text file. We want to count how many times each word occurs.

1. We receive the file in input
2. We split the payload across each server
3. We perform a *mapping* (word to frequency)
4. We perform a shuffle, where each node receives all the records for a specific word
5. We perform a reduction, summing all the individual records' values
6. We perform a gather, obtaining the final result

The shuffle is quite intensive, and will probably result into an all-to-all communication. Therefore it is clearly the bottleneck of this `MapReduce` operation.

There are two challenges for NiC acceleration of this kind of workload:

1. SmartNiC has limited memory (4Gb to 16Gb), but we are dealing with *big* data
2. SmartNiC are slower than host cores (otherwise we wouldn't need GPUs)

General Idea:

- We merge the partition (we perform an initial node-by-node reduction/aggregation) so that traffic is reduced
- We aggregate the records on the NiC, so that we offload this merging from the CPU to the interface
- Some load-balancing is performed so that if the host has finished the mapping, it can dynamically *steal* work fron the NiC to perform the aggregation in parallel and reduce the bottleneck caused by the limited performance of the NiC's cores
- If we run out of memory, we perform *spilling*:
  - If mapper NiC runs out, offload to reducer
  - If reducer NiC runs out, offload to host

Research has shown that SmartNiC leads to a $\approx 30\%$ reduction in shuffle function execution, while also reducing CPU utilization dramatically.

Ninth Lecture

Load Balancing

Last lecture we discussed *load balancing*. The difference between congestion control and load balancing is that the second can be used to solve *some* congestion issues, but it is not a general solution. Some congestion problems, especially those that occur at the edge of the network, need explicit congestion control (slowdowns, etc...).

Congestion leads to accumulation of packets in the network, which leads to increased latency, and eventually to packet loss. This manifests into a massive increase of variance in the throughput, and a decrease in the average throughput.

Routing in Data Centers networks

Data centers are different from the internet. The network is *much* more regular and more *static*. Addresses and physical locations are known in advance and do *not* change too often.

We can assume that switches know all the possible paths in advance, however, the fastest path cannot be assumed to be computable, as the network is too large.

We still operate on a best-effort basis, using local knowledge to route packets. The *most* advanced algorithms for this are propietary (*bad*), hence, we do *not* know how the big companies do it.

Load Balancing Landscape

We show various categories of load balancing algorithms:

- Centralized: A single entity decides the routing of the packets. (Hedera, Planck, Fastpass, ...)
- Distributed: Each switch decides the routing of the packets. (Jellyfish, DCell, BCube, ...)
  - In-network: the decisions are performed by the switches, but the switches are not aware of the global state of the network.
    - Congestion Oblivious: ECMP, WCMP, packet-spraying, ...
    - Congestion Aware: Flare, TeXCP, CONGA, DeTail, HULA, ...
  - Host-based: The decisions are performed by the individual hosts.
    - Congestion Oblivious: Presto
    - Congestion Aware: MPTCP, FlowBender, ...

A Congestion Oblivious algorithm is an algorithm that does not take into account the current state of the network, and it is not aware of the congestion that is happening in the network. Congestion Aware algorithms, on the other hand, are aware of the congestion that is happening in the network, and they take this into account when making routing decisions.

In-Network Cong. Oblivious Load Balancing

One of the most simple families of solutions is that of in-network congestion oblivious load balancing. For our examples we consider up/down routing across a fat-tree topology.

ECMP We first consider the ECMP (Equal-Cost, Multi-Path) algorithm.

Due to the fat-tree topology, we have many equal cost paths going up to the core switches. When going down, there is a single path (refer to slides for a visual

representation).

We can uniquely identify a transmission by hasing the pair of source and destination IP addresses and sockets, we can then use this hash to decide which path to take. This also ensures that packets that belong to the same flow are always routed through the same path, meaning that hopefully they will arrive in order, minimizing the retransmission in the case of TCP/IP usage.

Issues are:

1. The hasing function does not necessarily guarantee an even flow distribution (leading to under/over-utilization of some links)
2. If a collision occurs for a pair of long (elephant) flows, it will last for the entire duration of the flow.

In practice, this drops the bisection bandwidth by 61% for a network of $\approx 25k$ servers.

Another issue is that the algorithm is *not* falut-tolerant, meaning that if a selected path fails, the packet is lost.

Random Packet Spraying In order to avoid this deterministic behavior, we can use a random packet spraying algorithm. This algorithm randomly selects a path for each packet, and it is therefore more fault-tolerant than ECMP. However, the spraying is likely to lead to out-of-order packet delivery, which is not ideal for TCP/IP. Out-of-order delivery might also not play well with some RDMA protocols.

In-Network Cong. Aware Load Balancing

CONGA Each leaf switch keeps track of the congestion on all the paths towards all the other leaf switches. This is done by building a Congestion-to-Leaf table (N destination leafs $\times M$ possible paths).

Congestion on a path is the maximum queue length on that path (or the average queue length, or the queue length of the bottleneck switch, etc... Whatever utility function can be picked as long as it is *correlated* with congestion).

We then consider the concept of *flowlet*, which is a series of packets that belong to the same flow, sent in a short time frame. The idea is that we want to keep the packets belonging to the same flowlet together (to avoid reordering), but we want to spread the flowlets across different paths. We use the information in the Congestion-to-Leaf table to decide which path to take.

A new flowlet is started when a time-δ passes between two bursts of packets from the same flow. The assumption is that, as long as the individual flowlets are ordered (one after another), we just need to keep the same flowlet packets on the same path in order to minimize reordering.

From a different perspective, it is as if I was *heuristically* making sure that there are *no* in-flight packets when I select a new path for the same flow.

Selecting this δ is non-trivial, as it is essentially a parameter that determines how the protocol acts. A low δ means that we approximate packet spraying, a high δ means that we approximate ECMP. Moreover, RDMA flows are not as bursty as TCP/IP flows, so this flowlet approach might not be as effective.

Centralized Load Balancing

Hedera We assume that ECMP load-balancing is *efficient* for small flows. We then want to detect *large flows*.

A scheduler continuosly polls edge switches for flow byte-counts. When a flow reaches a certain threshold, it is considered a large flow. You can then use an estimate of flow deands to heuristically reschedule the flows in order to maximize bisect bandwidth.

This is only worth it for flows that last *minutes*, otherwise the overhead of the polling and scheduling is too high.

Host-Based Cong. Aware Load Balancing

The issues with centralized load balancing is that it is extremely unreactive and slow. Distributing over the network might also require changes to the hardware or transport protocols.

Moving the load balancing to the host is a more flexible solution, as it does not require changes to the network, and it can be done in software.

Flow-bender A possible solution is *Flow-bender*, which is a host-based congestion-aware load balancing algorithm. The idea is to use ECMP to spread the load across the network, and then use a congestion-aware algorithm to force a re-hash of the flow if the path is congested.

Load Balancing in non-Tree Topologies

In a tree topology, all paths are of equal length. If we have a more complex topology, we need to take into account the length of the paths. It might happen that a longer path (in terms of links) is less congested than a shorter path, and therefore results in a faster delivery.

More compelx algorithms that are able to estimate the latency of the paths are needed in this case. This problem is extra-hard and even state-of-the-art production systems do not have a good solution for this. Meaning that these estimates are often wrong.

In-Network Compute

So far we assumed that the switches mostly take care of moving packets from port A to port B. However, we can also use the switches to perform some computation on the packets.

Existing smart switches operate in the same way as Smart-NICs, using Reconfigurable Match-Action Tables (RMTs) to perform computation on the packets.

Use Cases

We can use these smart switches to gather some data about congestion (In-Network Telemetry, or TIM), perform load balancing, or more complex use-cases such as MapReduce/AllReduce (as seen with Smart-NICs).

In-Network Allreduce

We can build a logical tree-topology on top of the physical network, and then use the switches to perform the Allreduce operation. This is a very efficient way to perform Allreduce, as it is done in a logarithmic number of hops, and requires no computation on the hosts.

We extend our performance table for the Allreduce operation to include the In-Network Allreduce:

Operation	Volume	Steps
Ideal	n	1
Parameter Server	$max(n, \frac{pn}{k})$	1
Näive AllReduce	$(p-1) \cdot n$	1
Ring AllReduce	$2 \cdot n$	$2(p-1)$
Bandwidth-optimal recursive doubling	$2 \cdot n$	$2\log_2(p)$
In-Network AllReduce	n	1

We can see that in-network Allreduce is ***optimal*** in terms of bandwidth, as it requires the least amount of data to be transferred, in the least amount of steps.

Practical implementations have been present with different variations in HPC networks for several years. (Cray Aries, Mellanox SHARP, Tofu, etc...), it is also possible to perform other operations such as Broadcast, Gather, Scatter, etc...

Practically, in-network compute is vulnerable to congestion, and its performance degrades with the increase of the number of operations that are performed on the switches.

Lossy approximation algorithms can be used so that congested packets are ***not*** aggregated, and the performance is kept high and robust to congestion.

Network reductions are also not reproducible, as the order of the packets is not guaranteed, and IEEE 754 floating point operations are not associative. There is no known solution to this problem without buffering the packets, which would defeat the purpose of the in-network compute.

Furthermore, there are usually no floating point ALUs in the switches, so the operations are performed in fixed-point arithmetic, which can lead to precision loss

and more complicated algorithms. This still requires casting from IEEE 754 to fixed-point, which is not trivial and usually must be done in-hardware for it to not cause performance degradation.

Lastly, switches are not fault tolerant, since fault tolerance is implemented at the network layer (TCP/IP), we can implement *simple* mechanism (such as timeout retransmission) to ensure that the packets are delivered correctly. In general, we approach the problem with a best-effort mindset, and we do not guarantee that the packets are delivered correctly.