Computer Science Course Notes — 1st Semester

Dario Loi

Contents

Advanced Machine Learning

First Lecture

A *well-posed learning problem* occurs when a computer program learns from experience E, w.r.t some task T and some performance measure P. learning occurs if its performance on T, as measured by P, improves with experience E.

Stages of intelligent systems:

1. Expert systems (1960s–1980s)
2. Unsupervised learning (1980s–1990s)
3. Data-driven learning (2000s)

This course will focus on the third stage. Increases in flexibility and model expressiveness correspond to exponential increases in the number of parameters and computational complexity, in terms of FLOPs and dataset size.

Second Lecture

The second lecture recaps a number of basic ML concepts such as SGD, Losses, Regularization, etc...

We skip these notes since they are already covered in so many previous courses, they are to be considered trivial at this point

Backpropagation

We take notes on backpropagation since it is usually always expressed in a different formalism in every course.

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.

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

Distributed Systems

First Lecture

DA RECUPERARE !!!

A Space-Time diagram shows the evolution of a distributed system over time, it is composed of a number of timelines, each associated to a process p_i. Events, occur across this timelines, and are represented by points on the diagram. If an event e_1 is causally related to another event e_2, then e_1 is to the left of e_2 and there is a line connecting them.

Second Lecture

Definitions

We now give a set of formal definitions for very natural concepts in distributed systems.

We define the *history* of a process p_i as the sequence of all events that occur in the timeline of p_i.

$$h_i = \left\langle e_i^1, e_i^2, \ldots, e_i^n \right\rangle$$

We can then have the history of the computation, which is a collection of all the histories of all the processes.

$$H = \langle h_1, h_2, \ldots, h_n \rangle$$

A *prefix history* is a sequence of events that occur in the timeline of a process up to a certain point.

$$h_i^k = \left\langle e_i^1, e_i^2, \ldots, e_i^k \right\rangle$$

A *local state* of process i after event e_i^k is the result of the computation up to that point, and is denoted as σ_i^k.

A *global state* is the collection of all local states of all processes at a certain point in time.

$$\sigma^k = \left\langle \sigma_1^k, \sigma_2^k, \ldots, \sigma_n^k \right\rangle$$

A *run* k' is a total re-ordering of the events such that the internal order of every process is preserved. This means that a *run* allows the interleaving of the events of different processes, but does not break the order of events in the history of a single process.

A *cut* C is a collection of the prefixes of every processes' history up to a certain point.

Consistency

A cut C is said to be *consistent* iff:

$$e \rightarrow e' \wedge e' \in C \Rightarrow e \in C$$

Or, in other words, if an event is in the cut, then all the events that causally precede it are also in the cut.

By sequentializing the events in a run and extracting a prefix, we can trivially obtain a consistent cut.

Consistency of Intersection

A possible course exercise is, given two cuts C_1 and C_2, to prove that their intersection is also a consistent cut.

Given the definition of consistency accomodated for the intersection of two cuts:

$$e \rightarrow e' \wedge e' \in C_1 \cap C_2 \Rightarrow e \in C_1 \cap C_2$$

We know that if e' is in the intersection, then it is in both C_1 and C_2. By the definition of consistency, we know that e must be in both C_1 and C_2, and therefore in their intersection.

Formally:

$$e \rightarrow e' \wedge e' \in C_1 \rightarrow e \in C_1$$

$$e \rightarrow e' \wedge e' \in C_2 \rightarrow e \in C_2$$

Therefore:

$$e \rightarrow e' \wedge e' \in C_1 \cap C_2 \rightarrow e \in C_1 \wedge e \in C_2 \rightarrow e \in C_1 \cap C_2$$

Consistency of Union

Given two consistent cuts C_1 and C_2, we can prove that their union is also a consistent cut.

Given the definition of consistency accomodated for the union of two cuts:

$$e \rightarrow e' \wedge e' \in C_1 \cup C_2 \Rightarrow e \in C_1 \cup C_2$$

Then, formally:

$$e \rightarrow e' \wedge e' \in C_1 \rightarrow e \in C_1$$

$$e \rightarrow e' \wedge e' \in C_2 \rightarrow e \in C_2$$

Therefore:

$$e \rightarrow e' \wedge e' \in C_1 \cup C_2 \rightarrow e \in C_1 \vee e \in C_2 \rightarrow e \in C_1 \cup C_2$$

Run Reconstruction

Assume that we have a set of processes $P = \{p_1, p_2, p_3, \}$, and a monitor process p_0 that is responsible for detecting deadlocks.

Everytime that an event is generated, its responsible process sends a message to the monitor, which then updates its local state.

The monitor process can then reconstruct the run by aggregating the local process states into a global state.

Assuming asynchrony and unbounded message delays, the monitor process does not reconstruct a run, since the order of the individual processes can be inverted.

If we assume a FIFO message delivery, then the monitor process can reconstruct a run. This can be trivially achieved by having a sequence number, in a system such as TCP.

This does *not* recover consistency, since the order of separate processes can still be inverted. However, this level of *local consistency* is sufficient for deadlock detection.

Simplifying Assumptions for Consistency

Assume that I have:

1. A finite δ_i notification delay on each process p_i.
2. A global clock C that is synchronized with all processes. We call an idealized version of this clock *Real Clock* (RC).

Then, the monitor process can offline-reconstruct a run by using the global clock to order the events.

Online-reconstruction can be achieved (and is preferred) by exploiting the additional δ_i information. Once we receive a notification, we await time $\max \delta_i$ before committing the event to the global state, in the meantime, we store the event in a buffer that is ordered by the global clock.

Real Clock Property

The clock property relates the *happens before* relation of messages to the real time of the system.

$$e \rightarrow e' \Rightarrow TS(e) < TS(e')$$

We can obtain a trivial clock by designing a system as such:

1. Each process has associates a sequence number $k \in \mathbb{N}$ to each event.
2. Each event i, which receives a message m from event j, has a sequence number given by the formula $\max(k_i, k_j) + 1$.

This gives us the clock property *by construction*, since each event has a sequence number that is always greater than the sequence number of the event that caused it and of any event that it is casually related to.

This device is called a *Lamport Clock*, or a *Logical Clock*.

History of an event

Given an event e, we can define its history as the sequence of events that causally precede it. These are all the events that the monitor process must have received before the event e is committed to the global state.

$$\Theta(e) = \{e' \mid e' \rightarrow e\}$$

This is a *consistent* cut.

Opaque detection of consistent cuts

Given a notification about an event, for example, e_5^4, the monitor process can ensure that e_1^4 can be committed to global state by checking that for every process, we have received a notification with lamport clock ≥ 4.

We might have some processes that consistently lag behind, and therefore do not reach the lamport clock in a timely manner. We can trivially solve this by sending a liveness check to all processes for which a sufficiently high lamport number has not been received. If the process is alive but not working, it will re-adjust its lamport clock and send a notification.

This will not impact throughput, since the monitor process is not a bottleneck, and the liveness check is a very lightweight operation, that is performed only when necessary.

Building a Better Clock

We can build a *better clock*, that is, one with the enhanced property:

$$e \rightarrow e' \iff TS(e) < TS(e')$$

This is called a *Vector Clock*.

It works by communicating the history of the current event in a compact way, we do this by sending a vector of length n where n is the number of processes in the system. The vector's entries are updated to be the highest lamport number detected for that relative process, either from a message that has been received, or from the local lamport clock (in the case of the current process).

This allows us to check whether:

$$VC(e) \subseteq VC(e')$$

$$e \rightarrow e' \iff VC(e) \subseteq VC(e')$$

When messages are sent, the vector clock is attached to the message, and the receiving process updates its vector clock by taking the maximum of the two vectors, and summing 1 to the current process.

The monitor process p_0 keeps track of its own vector, where each entry is the highest lamport number that it has received from each process. When it receives a notification, if the vector clock is consistent with the monitor's vector clock, then the event is committed to the global state.

Since vectors are ***not*** sets, we have to use component-wise comparison to check for causality.

$$VC(e) < VC(e') \iff \forall i \in \{1, 2, \ldots, n\} \quad VC(e)_i < VC(e')_i$$

So we are back to the original definition of the happens before relation.

Third Lecture

We now discuss some more advanced vector clock properties.

Vector Clock Properties

Strong Clock Property

Assume we have two events e and e' . Then, we have that:

$$e \rightarrow e' \iff VC(e) \leq^{\star} VC(e')$$

Where $\leq^{s} tar$ is a ***partial order*** relation, such as:

$$\forall VC(e_i)[k] \leq VC(e_j)[k] \quad \wedge \quad \exists k' VC(e_i)[k'] < VC(e_j)[k']$$

Meaning that all the elements of the vector clock of e are less than or equal to the corresponding elements of the vector clock of e', and at least one element is strictly less, so that we have a ***strict*** partial order.

Retrieving History Cardinality

Given a vector clock $VC(e)$, we can retrieve the number of events that causally precede it by taking the sum of all the elements of the vector clock.

$$|\Theta(e)| = \sum_{i=1}^{n} VC(e)[i]$$

Retrieving Causality

Given two events e_i and e_j, we can check whether they are causally related by checking the vector clocks.

$$e_i \rightarrow e_j \iff VC(e_i)[i] \leq VC(e_j)[i]$$

Existance of a Causal Event

We want to check whether

$$\exists e_k : \neg (e_k \rightarrow e_i) \wedge \left(e_k \rightarrow e_j\right)$$

This is true if and only if:

$$\exists k : VC(e_i)[k] < VC(e_j)[k]$$

Meaning that there is a certain event that ticked the clock of e_j but not the clock of e_i.

Snapshots

Now we assume that the processes do ***not*** send notifications, rather, the monitoring process p_0 polls processes with a snapshot request.

This was already discussed in the first lecture, and we know that what we retrieve cannot retrieve a consistent cut.

We can change the protocol to improve this method. Now p_0 sends a snapshot request to all processes, however, when a process receives a snapshot request, it has to broadcast it to any other process.

Only requests from p_0 are broadcasted, so that we do not flood the network with snapshot requests.

When a process receives a snapshot request, it sends a snapshot of its local state to p_0.

With this behavior, under the reasonable FIFO-channel assumption, we can reconstruct a consistent cut.

Proof of Consistency

Assume that we have e_i and e_j such that $e_i \to e_j$, and that e_i is in the snapshot C, we want to prove that e_j is also in the snapshot.

By contradiction, assume that $e_i \notin C$, then p_i received the snapshot request before e_i, but then, since the channels are FIFO, and e_j was spawned by a received message from p_i, then the broadcasted snapshot request sent to p_j reached p_j before e_j was spawned, and therefore e_j is also ***not*** in the snapshot.

This protocol is known as the *Chandy-Lamport* protocol.

Fourth Lecture

We want to design a protocol for *atomic commits*, this protocol should have the following properties:

1. If processes reach a decision (commit/abort), it must be the same one.
2. A process cannot reverse its decision after reaching it.
3. The commit decision can be reached only if all processes vote *Yes*.
4. If *there are no failures*, and all processes vote *Yes*, then the commit decision is reached.

Naturally, this could be satisfied by a *trivial* protocol, such as one that *always* aborts, or one that *always* commits. These protocols are not useful since they are not *safe*.

Generic Properties of Distributed Systems

Usually, there are two sets in which properties can be divided:

1. *Safety* properties, which basically ask "*if* something happens, it does not go *wrong*".
2. *Liveness* properties, which basically ask "*if* time goes on, *something* will happen".

You can trivially satisfy safety by doing nothing, you can trivially satisfy liveness by doing anything.

You *want* a tradeoff, such that you get as much liveness as possible while preserving safety.

Two-Phase Commit

We assume the existance of *two* different processes, *coordinators* and *participants*, we describe the protocol as follows:

1. The coordinator sends a *Prepare* message to all participants.
2. The participants decide Yes/No and send the vote to the coordinator. If a participant votes *No*, then the coordinator must abort.
3. The coordinator receives all votes, if all votes are *Yes*, then the coordinator sends a *Commit* message to all participants, otherwise it sends an *Abort* message.
4. The participants apply the decision.

If the coordinator fails, then the participants will communicate with each other to reach a decision.

If a vote fails to reach the coordinator after a certain timeout period, it will be assumed to be a *No* vote. This preserves the safety property.

This protocol satisfies the properties of atomic commits, and is known as the *Two-Phase Commit* protocol. This is *not* a *live* protocol, since it can deadlock/timeout.

Logs

Logs are a sequence of events that are written to a file, and are used to recover the state of a system in case of failure. We assume the records to be consistent, so that system failures do not corrupt the log. This means that the log is written to a stable storage, such as a disk.

Logs allow a system to be fault-tolerance since they can be used to recover the state of the system in case of failure.

You can have *two* types of behavior when logging, depending on the order of the operations:

- Log and then act
- Act and then log

In both cases, you might *die* in the middle of the operation, and you might have to recover the state of the system. Both systems are recoverable as a coordinator, but you *must* act in a consistent way to recover the state of the system.

As a participant, recoverability is *not* guaranteed if you act before you log, The best way to ensure recoverability is to log before you act, so that you can be sure about if you voted *Yes* or *No* before you die.

If you vote was *No*, then you can *always* abort, if your vote was *Yes*, then you can re-send the vote to the coordinator, and await the decision.

Fifth Lecture

Paxos

The previous lecture was a specific case of a more general problem, that of *consensus*.

In the consensus problem, we have a set of processes that must agree on a certain value, and we want to ensure that the value is agreed upon by all processes.

Informal Description

Paxos is an asynchronous protocol that solve the consensus problem. It has been developed by Leslie Lamport, and is used in many distributed systems.

Paxos is *more* live than the Two-Phase Commit protocol, since it can recover from failures and timeouts.

Paxos is in essence a *replication system*, meaning that you want to replicate values across multiple processes (that are potentially distributed across multiple machines) in a fault-tolerant way.

Paxos is capable of obtaining consensus on a *single value*, obtaining consensus across a *sequence* requires running *multiple instances* of the protocol.

By *faults* we are referring to *crashes*, that is, the possibility that a process might stop working. Processes could eventually recover, but there is no way to know and no way to guarantee that they will.

Structure of the Problem

We first start by defining different classes of processes that participate in the protocol:

1. *Acceptors*: Their role is to vote on whether a value is accepted or not.
2. *Proposers*: Their role is to propose a value to be accepted.
3. *Learners*: Their role is to learn the value that has been accepted.

The idea is that if I get enough votes from the acceptors, then I can be sure that the value is accepted.

The definition of *enough* is flexible, and can be defined by the system designer. Naturally, requiring more votes leads to more fault-tolerance, but also less liveliness.

Asking for *all* the acceptors to vote coherently essentially degrates the protocol to a Two-Phase Commit protocol.

The threshold chosen is called a *Quorum* (akin to parliamentary democracy).

We also have the notion of a *Round*, which are statically associated to proposers by some rule. A round must be started whenever a proposer wants to propose a value.

A simple rule for round selection is, given n proposers, the proposer i always starts rounds $k \cdot n + i$. They also have to start the smallest round that is greater than the round of the last proposal that they have seen.

Phases of the protocol

1. A proposer sends a `prepare(r)` message to all acceptors, proposing a value relative to round `r`, here we do not need to send the value but merely receive some promises that the acceptors will vote for us on this round.
2. Acceptors then respond with `promise(r, last_round, last_value)`, where `last_round` is the round of the last proposal that the acceptor has seen, and `last_value` is the value of the last proposal that the acceptor has seen. For the first proposal, `last_round` is set to $-\infty$ or some other sentinel value.
3. An `accept(r, value)` message is sent to all acceptors, where `value` is the value that the proposer wants to propose.
4. Learners receive an `accepted(r, value)` message, and learn the value.

How to choose a value

A proposer follows the following rules to choose a value:

1. Take the `promise` message with the highest `last_round` value.
2. The value is the `last_value` of the `promise` message with the highest `last_round` value.
3. If no `promise` messages were received by any acceptor in my quorum, then I can propose any value that I want.

Models of Computation

First Lesson

Contact Information

Prof email: piperno@di.uniroma1.it

Actual lecture times:

- Wednsday 13:30 - 15:00
- Thursday 16:00 - 17:30, or 16:15 - 17:55

Course Contents

The course content will focus on **functional programming** and λ-calculus.

The main application of these languages is that of *function application*, these languages are devoid of *assignment* semantics, and are therefore called *pure*.

The main ingredients of our programs will be:

- Variables, which are usually denoted by lowercase letters
- Starting from these, you obtain the set of all programs, which are called λ-terms, this set is denoted as Λ.

Set Definition

We can provide an inductive definition of a set of λ-terms, Λ:

$$\frac{x \in V}{x \in \Lambda} \quad \text{(var)}$$

$$\frac{M \in \Lambda \quad N \in \Lambda}{(MN) \in \Lambda} \quad \text{(app)}$$

$$\frac{M \in \Lambda \quad x \in V}{\lambda x.M \in \Lambda} \quad \text{(abs)}$$

Applying these inductive rules (variables, application, abstraction).

Bactus Normal Form

Another way of describing a set of lambda terms is by using Bactus Normal Form (BNF).

$$\Lambda :: Var|\Lambda\Lambda|\lambda Var.\Lambda$$

Which is essentially a grammar for the set of lambda terms.

Lambda calculus is left-associative:

$$((xy)z) = xyz \neq x(yz)$$

All functions are unary (we assume currying). For example, the function $f(x,y)$ is represented as:

$$\lambda x.\lambda y.fxy$$

.

Functions in lambda calculus can be applied to other functions or themselves, they can also return functions.

Beta Reduction

The main operation in lambda calculus is ***beta reduction***, which is the application of a function to an argument.

$$\underbrace{(\lambda x.M)N}_{redex} \rightarrow_\beta M[N/x]$$

Where $M[N/x]$ is the result of substituting all occurrences of x in M with N.

Some other examples:

$$(\lambda x.x)y \rightarrow_\beta y$$

$$(\lambda x.xx)y \rightarrow_\beta yy$$

$$(\lambda xy.yx)(\lambda u.u) \rightarrow_\beta \lambda y.y(\lambda u.u)$$

$$(\lambda xy.yx)(\lambda t.y) \rightarrow_\beta \lambda y.y(\lambda t.y)$$

This rule can be applied in any context in which it appears.

Formal Substitution Definition

We give a formal definition of substitution, $M[N/x]$:

$$x[N/x] = N$$

$$y[N/x] = y$$

$$(M_1 M_2)[N/x] = M_1[N/x]M_2[N/x]$$

$$(\lambda t.P)[N/x] = \lambda t.(P[N/x])$$

As observed, substitution is always in place of *free* variables, therefore the abstraction is *not* replaced in the last rule.

If we had an abstraction of type $\lambda x.P$ where $x \in P$, it would be best to rename x in order to avoid name clashes.

Types of Variables

We distinguish two kinds of variables:

- Free variables: variables that are not bound by an abstraction
- Bound variables: variables that are bound by an abstraction

For example, in the term $\lambda x.xy$, y is a free variable, while x is a bound variable.

Bound variables can be renamed, whereas for free variables the naming is relevant.

$$\begin{cases} FV(x) = \{x\} \\ FV(MN) = FV(M) \cup FV(N) \\ FV(\lambda x.M) = FV(M) - \{x\} \end{cases}$$

A set of lambda terms where $LM(\Lambda) =$ is called *closed.*

Extra Rules

$$(\mu) \quad \frac{M \to_\beta M'}{NM \to_\beta NM'}$$

$$(\nu) \quad \frac{M \to_\beta M'}{MN \to_\beta M'N}$$

$$(\xi) \quad \frac{M \to M'}{\lambda x.M \to \lambda x.M'}$$

These rules allow us to select redexes in a context-free manner in the middle of our lambda term. We can then choose the order of evaluation of our redexes, while still taking care of the left-associative order of precedence. Our calculus is therefore not *determinate* but is still *deterministic*, meaning that there may be multiple reduction strategies but they all lead to the same result.

This corollary is called the *Church Rosser Theorem*, discovered in 1936.

In general, a call-by-value-like semantic is preferrable when choosing evaluation paths, as it clears the most amount of terms as early as possible.

- Call By Value is *efficient*
- Call By Name is *complete*, **if** the lambda term is normalizable

Second Lecture

Alpha Reduction

Alpha reduction is the renaming of bound variables in a lambda term.

$$\lambda x.M \to_\alpha \lambda y.M[y/x]$$

This rule is used to avoid name clashes between bound variables.

Arithmetic Expressions

The set of all valid arithmetic expressions has a very precise syntax. In general, a syntax can be viewed either as a tool for checking validity or as a generator of valid expressions (a grammar).

We proceed to give a definition for arithmetic expressions

$$\frac{x \in \mathbb{N}}{x \in \text{Expr}} \quad \text{(num)}$$

$$\frac{X \in \text{Expr} \quad Y \in \text{Expr}}{X + Y \in \text{Expr}} \quad \text{(add)}$$

$$\frac{X \in \text{Expr} \quad Y \in \text{Expr}}{X \times Y \in \text{Expr}} \quad \text{(mul)}$$

Etc, etc... for all the other binary operations.

From this, we can successfully decompose any arithmetic expression into a syntactic tree. With this set of rules, we have a slight problem: we can't represent negative numbers. We could solve this either by adding a rule for unary minus, or by specifying the num rule over $\mathbb{Z}$ instead of $\mathbb{N}$.

Combinators

We define three combinators:

$$S = \lambda xyz.xz(yz)$$

$$K = \lambda xy.x$$

$$I = \lambda u.u$$

We have that $SKy \rightarrow_\beta I$

Exercise 1.1: β-reduce $S(KS)S$

This reduces to $\lambda zbc.z(bc)$, which is the B combinator (composition).

Exercise 1.2: β-reduce $S(BBS)(KK)$.

This reduces to $\lambda zcd.zdc$, which is the C combinator (permutation).

For exercise this, we show a step-by-step reduction:

$$\begin{aligned}
&S(BBS)(KK)\\
\rightsquigarrow&\lambda z.(BBS)z((KK)z)\\
\rightsquigarrow&\lambda z.(\lambda c.B(Sc))z((KK)z)\\
\rightsquigarrow&\lambda z.(\lambda c.B(Sc))zK\\
\rightsquigarrow&\lambda z.B(Sz)K\\
\rightsquigarrow&\lambda z.(\lambda c.(Sz)(Kc)\\
\rightsquigarrow&\lambda zc.Sz(Kc)\\
\rightsquigarrow&\lambda zcd.zd(Kcd)\\
\rightsquigarrow&\lambda zcd.zdc \quad \square.
\end{aligned}$$

Third Lecture

`Recupera!!!`

Fourth Lecture

First we do a simple exercise, β-reduce $\lambda uv.(\lambda z.zz)(\lambda t.tuv)$.

$$
\begin{aligned}
& \lambda uv.(\lambda z.zz)(\lambda t.tuv) \\
\rightarrow_\beta & \lambda uv.(\lambda t.tuv)(\lambda t.tuv) \\
\rightarrow_\beta & (\lambda t.tuv)(\lambda t.tuv) \\
\rightarrow_\beta & \lambda uv.(\lambda t.tuv)uv \\
\rightarrow_\beta & \lambda uv.uuvv \quad \square.
\end{aligned}
$$

Find a term X s.t $Xx = \lambda t.t(Xx)$

$$
\begin{aligned}
& Xx = \lambda t.t(Xx) \\
& X = (\lambda fxy.t(fx))X \\
& X = Y(\lambda fxy.t(fx))
\end{aligned}
$$

Find a term H s.t $H(\lambda x_1x_2x_3.P) = \lambda ax_3x_2x_1.ax_1x_2x_3$