SAPIENZA
UNIVERSITÀ DI ROMA

Faculty of Information Engineering, Informatics and Statistics
Department of Computer Science

Big Data Computing

Author:
Simone Lidonnici

3 november 2025

Contents

1
Introduction

With **Big Data** we refer to an actual phenomenon, defined by five properties called **5V**:

- **Value**: extracting knowledge from data is valuable.
- **Volume**: large amount of data.
- **Variety**: different format of data.
- **Velocity**: the speed at which the data are generated is very high.
- **Veracity**: reliability of the data used.

To do a computation on a lot of data at a high speed (like a google search), there are different problems that occurs:

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

1.1 Data center

To do this type of computations **data center** are used, which can upgrade their performances in two way:

- **Scale up**: buying new and more powerful components. This is a problem because the increase in the velocity required is faster then the performance increase.
- **Scale out**: buying more components and interconnect them to work in parallel.

A data center is composed by a series of rack interconnected between them with a private network. Every rack has servers inside them that contain CPUs, GPUs and memory. A server is considered a node in the network.
The major problem with the data centers is the bottleneck caused by the network. The network can't send all the data required for the computation in a fast way and this slow down the computation. Also, the increase in GPU power (that does the major part of computation in a data center) is much faster than the increase in network speed durind the years.

1.1.1 Reliability and Programmability problems

Another problem is related to **reliability**, so the probability that a server, disk or network component fails. Even if the probability of a single component is very low (one fail every 10 years) a large data center with thousand of components will have failures very frequently. This mean that checkpoint must be used, and every failure reset to the last one.

Last problem is the **programmability**, so how to write an efficient parallel program being aware of all the components. Things that have to be taken in account are how to store data, how to send data efficiently, what to execute on CPU or GPU and many other.

2

Distributed Deep Learning

Deep Learning performs better with more data and parameters, but also require more computation.
The training of a Distributed Neural Network (DNN) is done by doing many iteration of 3 staps:

1. **Forward propagation (Compute the function)**: apply the model to input and produce a prediction.

2. **Backward propagation (Compute the gradient)**: calculate the error and find the parameters that contribute more to the error to correct them.

3. **Parameters update (Use the gradient)**: use the loss value to update the model parameters.

2.1 Parallelize a DNN

There are 3 ways to parallelize a DNN:

- Data parallelism: divide the training dataset in batches and compute the model o each batch in parallel. After each iteration we synchronize the parameters of the different models by doing a gradient aggregation. This is done by doing an Allreduce on the weights vector across the GPUs.

- Pipeline parallelism: divide the model by the layers, assigning each layer to a different GPU. In the forward phase the first GPU output will be the input of the second GPU and so on. In the backward phase is the opposite. The problem is that the last GPUs have a lot of idle time during the forward phase and the first GPUs in the backward phase. This idle time is called bubble. To reduce this bubble we can divide the batch of data in micro-batches and do the forward and backward computation on the micro-batches. There are also other complex methods to reduce the bubble.

- Operator (or Tensor) parallelism: if all the parameters of a layer do not fit in a single GPU wehave to split the layer across multiple GPUs. An example could be splitting the matrices that we have to multiply (training require a lot of matrix multiplication) and using an Allreduce to sum the partial matrices. This requires a lot of communication.

In large models all three technique are usedat the same time.

2.2 Reducing communication overhead

To reduce the communication overhead we can compute and communicate in the same time, sending the gradients of every layer as soon as they are computed, so the computation of the gradient of the previous layer can be done during the communication of the gradient of the successive layer.

Another way is to compress the gradient and reducing his data volume, this can be done in three ways to have a **lossy compression**:

- Quantization: reduce the bitwidth of the elements (for example for float32 to float16). Some models converge even with only one bit per weight (knowing only if is positive or negative).
- Sparsification: sending only the larger k values or by setting a threshold and sending only the weights bigger than it.
- Low rank: decomposes gradient into lower-rank matrices.

Lossy compression training converges with the same asymptotic rate, but in real life it requires more iterations.

Asynchronous sytems could be also used to speed up the process and overcoming the gradient aggregation that acts as a barrier. This can be done with different techniques, like aggregating after not one but many iterations or not waiting the slowest GPUs.

2.3 Collective operations used in DNN

In DNN different collective operations are used:

- Allreduce for gradient aggregation in data parallelism.
- Allgather and Reduce-Scatter in sharded data parallelism.
- Allgather and Allreduce for matrix multiply in operator parallelism.
- Alltoall in expert parallelism.

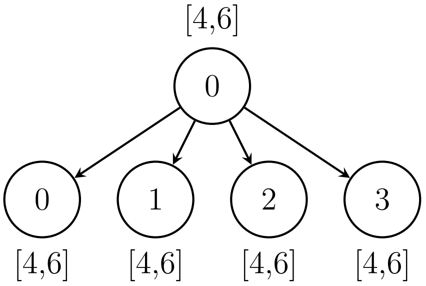
The cost model can be expressed with this formula:

$$T = k\alpha + \delta n\beta$$

In which α is the latency, a fixed cost, β is the cost per byte sent, n are the byte sent and δ is the fraction of data sent in respect to n, considered as the maximum between the input and the output.

2.3.1 Broadcast

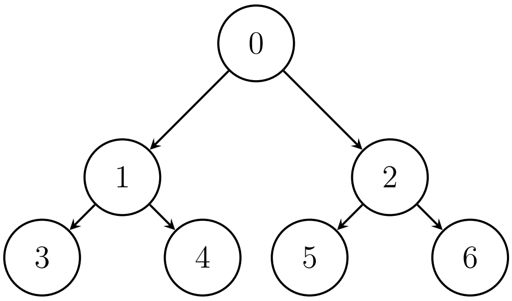
A simpler collective operation is the **Broadcast**, in which data from a process are copied to all the other processes.



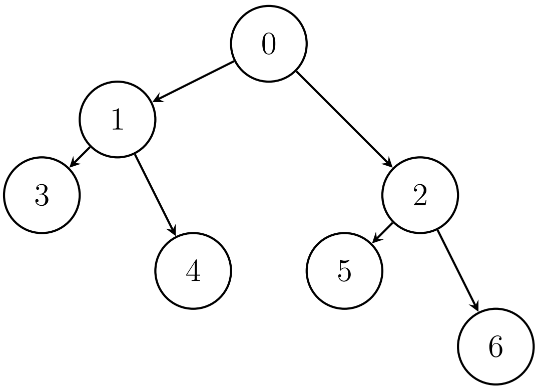
The basic algorithm to compute the Broadcast is the **chain algorithm**, in which at every step the last process to have recieved the data sends them to the successive. So, in step 0 the process 0 (who has the data at the start) sends them to the process 1, in te next step 1 will send them to process 2 and so on. The cost of this algorithm is:

$$T = (p-1)(\alpha + n\beta)$$

Another algorithm involves the binary trees, in which at each step the processes that have received the data at the last step send all the data to two processes that don't have yet.

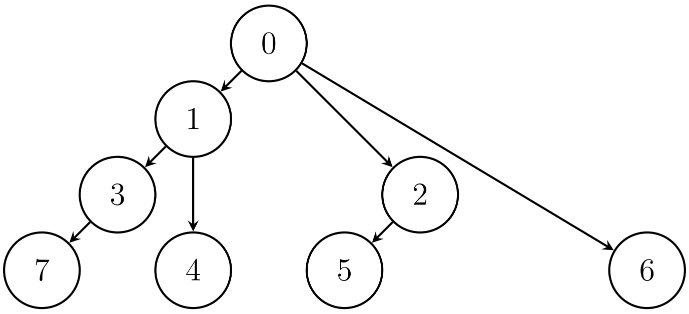


With concurrent communication:
$T = (\log(p+1)-1)(\alpha + n\beta)$



Without concurrent communication:
$T = 2(\log(p+1)-1)(\alpha + n\beta)$

Exists also an algorithm with binomial trees, in which a process doesn't stop sending after two steps but continue to send at every step.



The cost of this algorithm is:

$$T = \log p(\alpha + n\beta)$$

The Broadcast can also be done as a Scatter (takes a vector from a process and divide it evenly between all processes) plus a Allgather. The cost of the Scatter, using the binomial trees, is similar to the Broadcast algorithm, with the difference that the data sent halves every step, so for a step k the data sent will be $\frac{n}{2^k}$. The cost of the Scatter is:

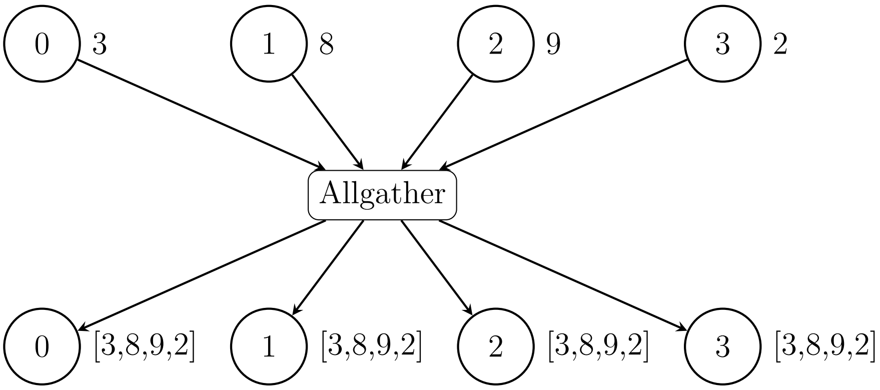
$$T = (\log p)\alpha + n\beta$$

For the Allgather we use the Recursive doubling algorithm explained in Section 2.3.2.
The total cost of this approach for the Broadcast is:

$$T = \underbrace{(\log p)\alpha + n\beta}_{\text{Scatter}} + \underbrace{(\log p)\alpha + n\beta}_{\text{Allgather}} = 2[(\log p)\alpha + n\beta]$$

2.3.2 Allgather

The **Allgather** is a collective operation that takes data from every process, combine them in a vector and copies the resulting vector to al the processes.

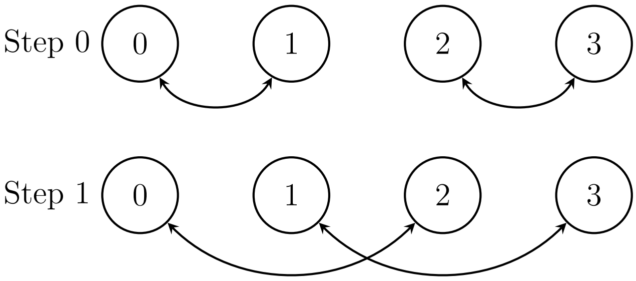


The basic algorithm to compute the Allgather consists in $p - 1$ phases in which every process sends his data to another process. The cost is:

$$T = p\alpha + n\beta$$

Another simple algorithm is the **Ring** algorithm, which use the same structure of the basic algorithm, but instead of sending the data to a different process every step, a process i will always send data to process $i+1$. The cost of the algorithm remain the same, but the difference is that the every link in the system is used by only one communication at a time.

The **Recursive doubling (Butterfly)** algorithm has a logaritmic number of phases and consist in exchanging data in a bidirectional way between process with distance that doubles at every step. In a step k, a process i will exchange all the data with a process at distance 2^k.

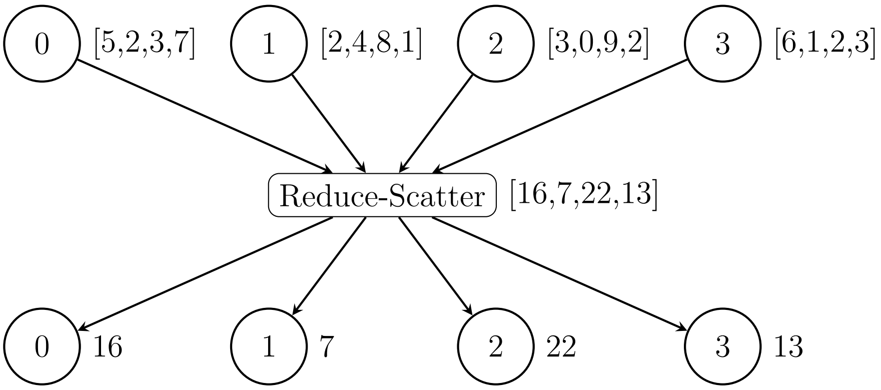


The cost is:

$$T = (\log p)\alpha + n\beta$$

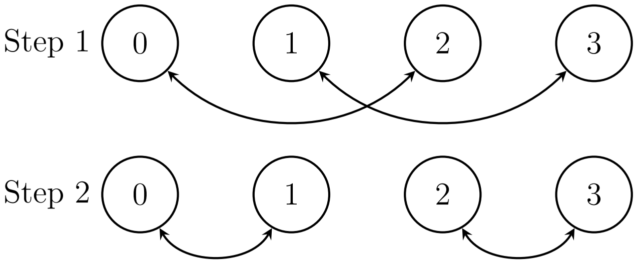
2.3.3 Reduce-Scatter

The **Reduce-Scatter** is a collective operation that takes a vector of data from every process and execute an operation on them. After calculating the resulting vector, it is splitted evenly between the processes, following the indexes.



The basic and Ring algorithm for the Reduce-Scatter are the same algorithm of the Allgather but in reversing order of communication.

Instead of Recursive doubling, for Reduce-Scatter we use the **Recursive halving (Butterfly)** algorithm, which works in a similar way but with the distance that halves instead of doubling. Also the data exchanged is more in the first stage and halves every stage.

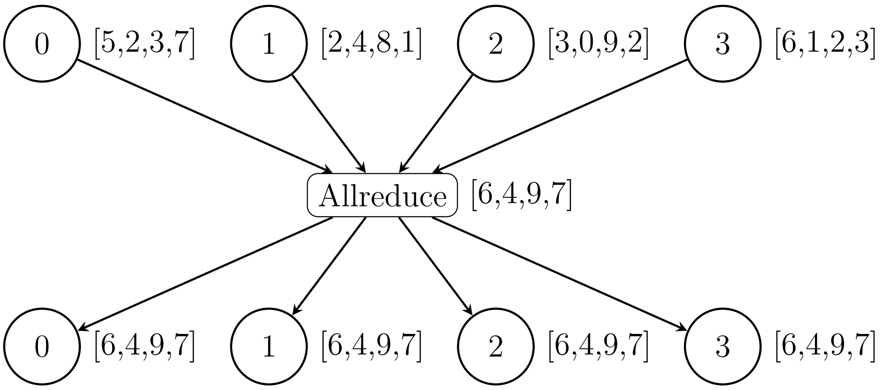


The cost is equal to the Recursive doubling algorithm of the Allgather:

$$T = (\log p)\alpha + n\beta$$

2.3.4 Allreduce

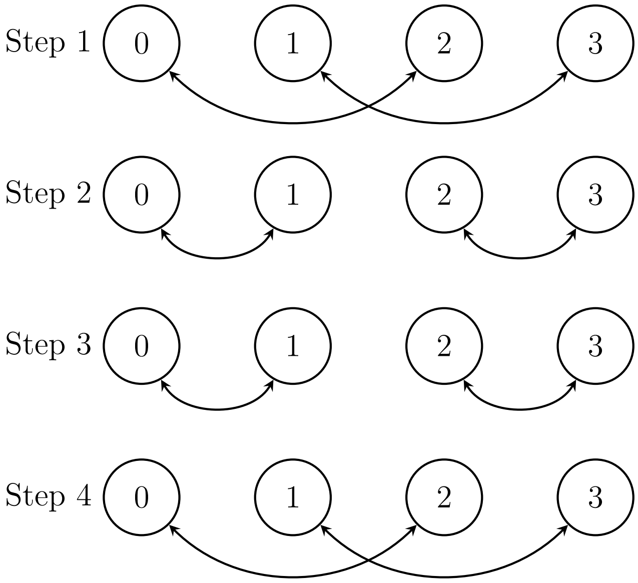
The **Allreduce** is a collective communication that takes a vector from every process, execute an operation on them and the resulting vector is copied in every process.



The Ring algorithm is done by executing a Reduce-Scatter and an Allgather with Ring algorithm, so the cost will be the sum of the two:

$$T = 2(p\alpha + n\beta)$$

The **Rabenseifner (Butterfly)** algorithm use a Reduce-Scatter with Recursive halving algorithm followed by an Allgather with Recursive doubling algorithm.



Also in this case the cost is the sum of the two algorithms:

$$T = 2[(\log p)\alpha + n\beta]$$

There is an algorithm that isn't the sum of a Reduce-Scatter and Allgather, the **Recursive doubling** algorithm. The algorithm is similar to the one to execute the Allgather but every step the processes send all the vector. The cost is:

$$T = (\log p)(\alpha + n\beta)$$

2.3.5 All-to-All

The **All-to-All** is a collective communication that takes a vector fro every process and moves the data so that at the end every process i has all the data with index i in the starting vectors. If we imagine the vector as columns of a matrix, the result of the All-to-All is the transposition of the matrix.

$$\begin{array}{c|c|c|c} p_0 & p_1 & p_2 & p_3 \\ \hline 00 & 10 & 20 & 30 \\ 01 & 11 & 21 & 31 \\ 02 & 12 & 22 & 32 \\ 03 & 13 & 23 & 33 \end{array} \xrightarrow{\text{All-to-All}} \begin{array}{c|c|c|c} p_0 & p_1 & p_2 & p_3 \\ \hline 00 & 01 & 02 & 03 \\ 10 & 11 & 12 & 13 \\ 20 & 21 & 22 & 23 \\ 30 & 31 & 32 & 33 \end{array}$$

The basic algorithm is a linear algorithm in which in every fase k every process i sends data to process $i + k$. The cost is:

$$T = p\alpha + n\beta$$

The **Bruck (Butterfly)** has a logaritmic number of phases and works similarly to the Recursive halving algorithm for Reduce-Scatter. The difference is that half the vector is sent at every step.

p_0	p_1	p_2	p_3
00	10	20	30
01	11	21	31
02	12	22	32
03	13	23	33

$\xrightarrow{\text{Shift}}$

p_0	p_1	p_2	p_3
00	11	22	33
01	12	23	30
02	13	20	31
03	10	21	32

$\xrightarrow{\text{Fase 1}}$

p_0	p_1	p_2	p_3
00	11	22	33
30	01	12	23
02	13	20	31
32	03	10	21

$\xrightarrow{\text{Fase 2}}$

p_0	p_1	p_2	p_3
00	11	22	33
30	01	12	23
20	31	02	13
10	21	32	03

So the cost of the algorithm is:

$$T = (\log p)\left(\alpha + \frac{n}{2}\beta\right)$$

3

Network topologies

The network **topology** define how the nodes in the system are connected between them. The topologies can be of different types and can be divided in categories using different parameters:

- Blocking and Non blocking: a network is non blocking if every pair of switch can be connected through disjoint path. So in an ideal case there is no congestion.
- Direct and Indirect: a network is direct if every switch is connected to at least one server.
- Regular and Irregular: a network is regular if can be rappresented with a regular graph.

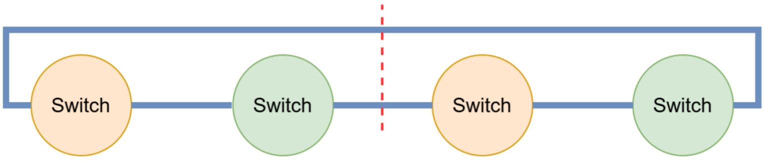
To start we need to explain some definitions:

- Switch radix: number of switch ports.
- Network diameter: maximum distance between two switches.
- Bisection cut: minimum number of links that needs to be cut to divide the network in two almost equal parts. The total bandwidth of the links cut is the Bisection bandwidth.
- All-to-All bandwidth (global bandwidth): the bandwidth the network has whe running a linear All-to-All. Non blocking topologies have full global bandwidth.

3.1 Torus and Mesh

3.1.1 Ring (Torus 1D)

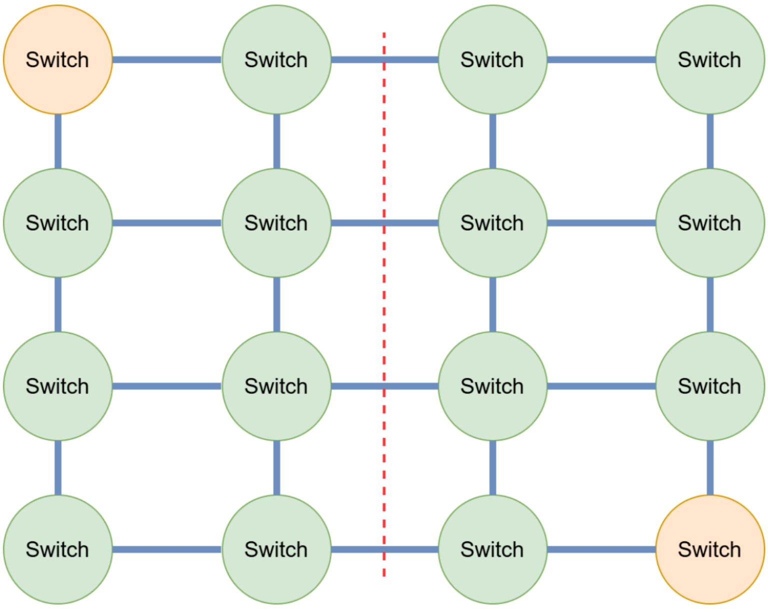
The **Ring** topology is a topology in which every server needs only two links, one to the previous server and one to the successive server.



This topology has a diameter of $\lceil \frac{n}{2} \rceil$ and a bisection cut of 2.

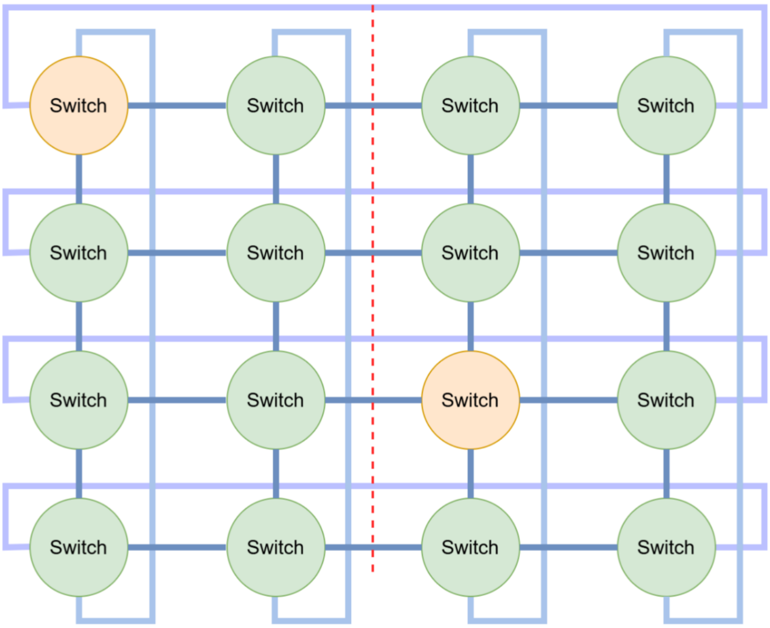
3.1.2 Mesh 2D

The **Mesh** topology is a grid and has a diameter of $2(\sqrt{n}-1)$ and a bisection cut of $\sqrt{n}$.



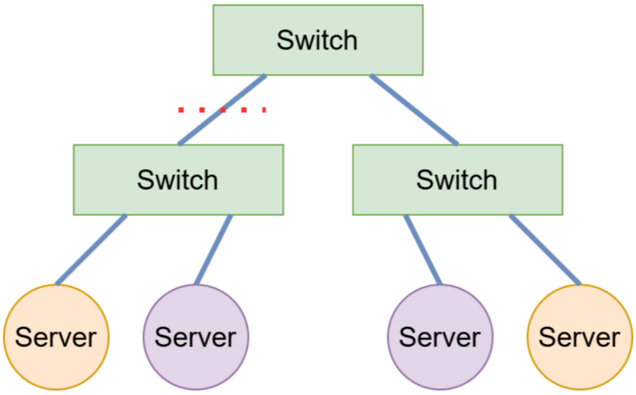
3.1.3 Torus 2D

The **Torus 2D** is similar to the mesh but with the first and last switch of every row column that are connected. The diameter is $\sqrt{n}$ and the bisection cut is $2\sqrt{n}$.



3.2 Trees

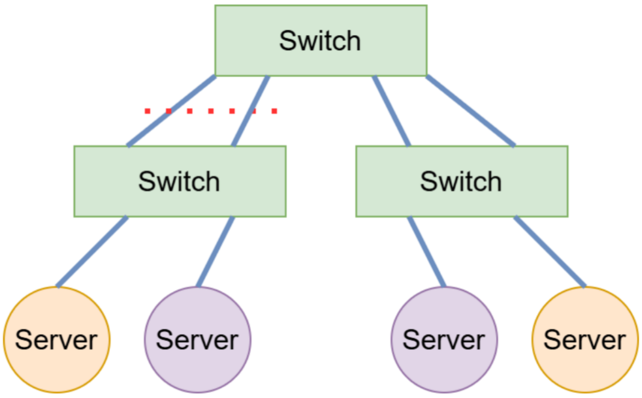
The tree topologies are based on the switch radix r. The basic tree has an height $h = \log_{r-1}(n)$, a diameter of $2h$ and a bisection cut of 1 (doesn't make much sense if the root has more than two child).



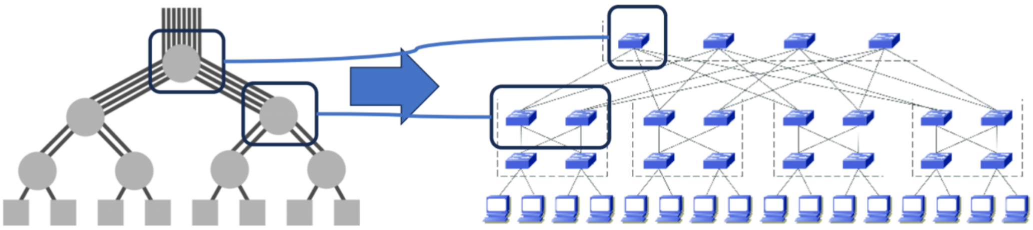
The basic tree is a blocking topology, because the switch have only one link going up.

3.2.1 Fat-tree

A **Fat-tree** is a tree in which every switch has the same number of links going up and going down. This makes this topology non blocking, providing full global bandwidth. The number of layer needed is also lower than the basic tree, having height $h = \log_{\frac{r}{2}}(n)$, and consequantly the diameter will be smaller.



With this change the switch on different layer have different radix but in a real network every switch has the same radix. So the switch are divided in smaller ones to make every one have the same radix.



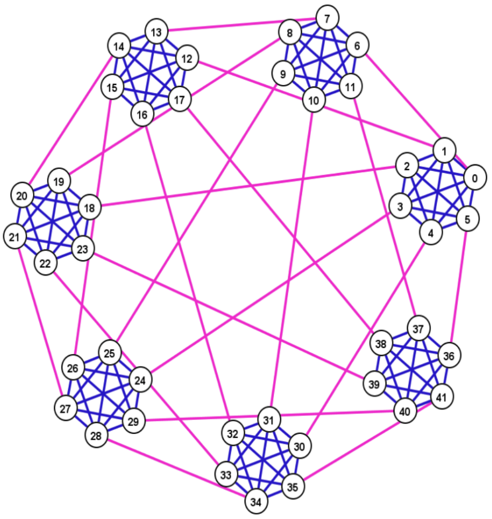
This network, also called **Folded CLOS** is the best that can be built in terms of performance but is also the most expendive one.

To reduce the cost we can make every switch have a ratio of 2:1 between links going down and links going up. This makes the topology become a blocking topology but saves money. The cost reduction is lineare with the ratio.

Fat-trees don't scale well because to add nodes we have to add layer an this increase the diameter.

3.3 Dragonfly

The **Dragonfly** topology is composed of fully connected groups of switches. The groups are also fully connected between them. This topology scales better than fat-tree and has a diameter of 3 (5 if the links between switch and server are counter) that is constant.

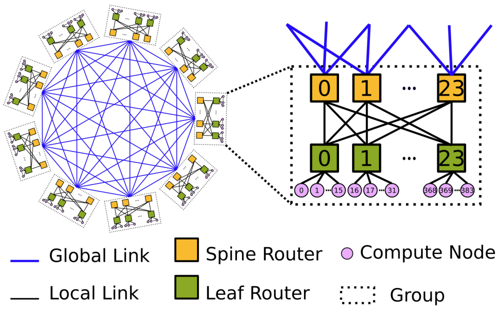


This topology has some disadvantages:

- Doesn't have full global bandwidth
- Adding new server is complicated.
- Load balancing is hard, because minimal and non-minimal path exists between two nodes.
- Can have loop, so deadlocks are possible.

3.3.1 Dragonfly+

The **Dragonfly+** is a modified version of the Dragonfly in which the groups are bipartite graphs.



The diameter is 3 like the dragonfly but can connect more server with the same switch radix.

3.4 Other topologies

- **Slimfly**: this topology is composed by two types of groups, each connected in a different way between itself. The good thing of this topology is the diameter equal to 2 but it works only for specific switch radix values.
- **HammingMesh**: this topology optimized o train Deep Learning models. Is a Mesh 2D divided into smaller meshes and in which every row and column is connected using a non blocking fat-tree.
- **Jellyfish**: this topology has no regular structure and uses a random graph so the is easy to add new servers. The problem are the management and configuration that are very hard.
- **Reconfigurable Data Center Network (RDCN)**; in this topology the links are not fixed but can change over time. This can improve the performance but it also has some problems like routing and when to change the links.

3.5 Load balancing

The **load balancing** (also known as **adaptive routing**) is when there are multiple path between a pair of node and the traffic must be balanced between the different paths. The algorithms can be divided in different categories:

- Centralized: there is an external controller that makes the decisions.
- Distributed: each server makes its own decisions. Distributed algorithm can be divided based on who takes the decision:
  - In-Network: decisions are made by switches.
  - Host-Based: decisions are made by the servers.

  They can also be divided based on if they choose path to improve the performance:
  - Congestion Oblivious: the path is changed randomly if there is congestion.
  - Congestion Aware: the path is changed to improve the performance.

3.5.1 In-Network Congestion Oblivious

The two main algorithm in this category are **Equal-cost Multi-path (ECMP)** and **Random packet spraying**.

ECMP uses an hash function to select a random output port for every message that arrives. This causes a lot of collision when multiple message get mapped to the same port.

Random packet spraying selects a random path for each packet (instead of each message). The packet might arrive out of order, that is a problem with TCP, but not with modern RDMA protocols.

3.5.2 In-Network Congestion Aware

This types of algorithms are based on **flowlet**, that are burst of packet created when the interval between two packets of the same flow is larger than a pre-set gap. This gap is set to be sure that the probability of packets arriving not in order is very low.

The first algorithm of this type is **CONGA**, in which every leaf switch keep track of the congestion on all the paths towards other leaft switches. When a flowlet arrive to the switch, it is sent to the least congested path.

Another algorithm is **Let It Flow**, in which every time a flowlet arrive to the switch, it is sent in a random path. If the path is congested, the flowlet will be smaller and will be rerouted.

3.5.3 Host-Based Congestion Aware

The algorithm of this type is **Flowbender**, that uses the ECMP hashing function to force every flow on a different path, when it detect congestion.

3.6 Congestion Control

The congestion control is needed for some problems that can't be resolved by load balancing. For example if the link congested is the one connected to an host or if a switch has two flow in input that want to exit on the same port.

Congestion control is based on detecting the congestion and adjusting the sending rate accordingly. The congestion can be detected by the sender, the switch or the reciever, but only the sender can adjust the rate.

3.6.1 Random Early Detection (RED)

The **RED** algorithm notify the congestion by dropping the packets (TCP will timeout), but instead of dropping it when the queue is full, two threshold are set, a minimum and a maximum threshold. When the average queue lenght is higher than the minimum threshold, the packets are dropped with a probability, if it exceed the maximum threshold the packets are always dropped.

3.6.2 Explicit Congestion Notification (ECN)

The **ECN** algorithm, uses two bits in the pakcet header. It follows the same logic as RED but instead of dropping the packets, it marks them.

When a packet is marked, the reciever will mark the corresponding Ack to communicate to the sender that there is congestion. The sender will then adjust the sending rate by 50%.

To avoid reporting outdated information a packet is merked when exiting the queue and not when entering.

3.6.3 Data Center TCP (DCTCP)

DCTCP works similarly to ECN but the sending rate is adjusted based on the amount of packets that are marked, if all the packets are marked the rate is reduced by 50%, but in all the other cases is reduced by a minor amount. Also, the marks control is done by using the istantaneous queue lenght and not the average.

3.6.4 Priority-Based Flow Control (PFC)

This algorithm assumes that the network is lossless (common in PC networks), so the switches don't drop packets.

It is based on the priority queue inside a switch, when one queue bcomes too full, a message is sent to all the application sending packets on that queue to make them stop.

There are two major problems with this algorithm, the first one is deadlock, because a cycle of queues could stop each other. The second is congestion spreading, when stopping a switch stops all the one before it and so on.

3.6.5 High Precision Congestion Control (HPCC)

The goal of the algorithm is to converge to the correct sending rate by using precise congestion signal. It works like ECN but instead of using two bits it attaches to te packet header more information: timestamp, queue lenght, trasmitted bytes and link bandwidth. The reciever also copies these informations on the corresponding Ack.

3.7 In-Network compute

The switch can be used, in addition to simply forwarding the packets from input to output, to do complex operation.

The existing programmable switches are based o **Reconfigurable Match-Action Table (RMT)** in which the programmer declares how packets are processed.

This in-network compute can be used to do simple tasks:

- In-network telemetry: to have more details on the status of the network.
- New traffic load balancing algorithm.
- Better support for congestion control.

It also can be used to perform more complex tasks:

- In-network Map-Reduce
- In-network Allreduce

3.7.1 Map-Reduce

Map-Reduce is an approach in which the input data is splitted, mapped, shuffled and reduced to final data.

Every switch does the partial aggregation of the data and sends the result to the successive switch. At the end the last switch does the reduction on the data of only two switches. This reduce the amount of data transfered and speed ups a lot the computation.

If a switch fills its memory it sends the packets without doing the partial aggregation, in the worst case is the same as doing the Map-Reduce without in-network computation.

3.7.2 Allreduce

To do an in-network Allreduce we build a reduction tree in which the switches do the reduction. So every switch does the partial reduction of the data and sends only the results. This is now a single phase algorithm with a cost of:

$$T = \alpha + n\beta$$

It has some issues:

- Load balancing
- Reproducibility: the result could be different every run because floating point sum is not associative.
- Lack of floating-point units: floating-points are expensive in terms of area and latency so is better to use fixed-point that are composed by integer bits and fraction bits.
- Packet losses and switch failures: if an already aggregated data is lost the retrasmission mechanism must be a lot complex to recover the data.

4
Software Infrastructure