Sapienza Università di Roma

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

Big Data Computing

Author:
Simone Lidonnici

17 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.

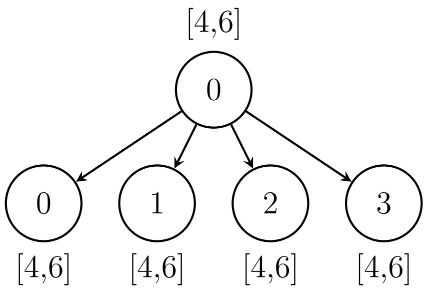
The cost model can be expressed with this formula:

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

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

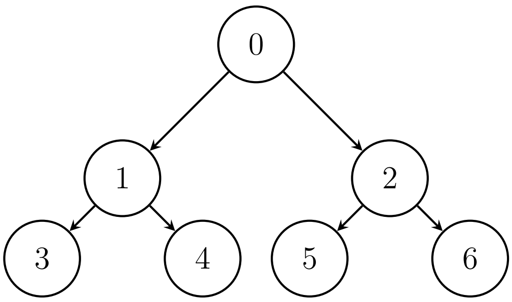
A simpler collective operation is the **Broadcast**, in which data from a process are copied to all the other processes.



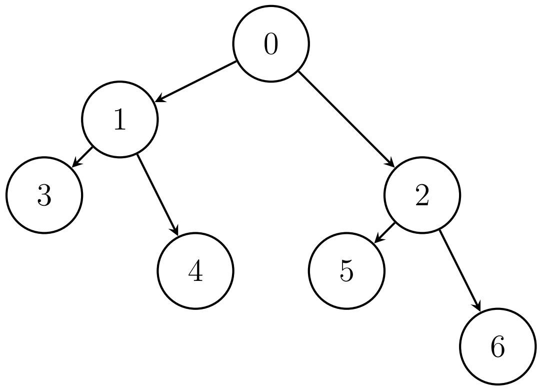
The basic algorithm to compute the Broadcast is the **chain algorithm**, in which at every step the last process to have recieved the data sends them to the successive. So, in step 0 the process 0 (who has the data at the start) sends them to the process 1, in te next step 1 will send them to process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

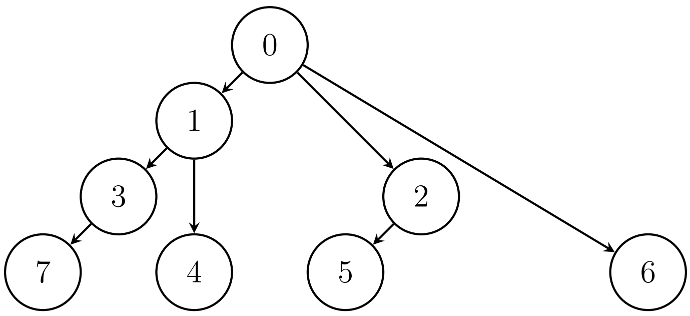


With concurrent communication:
$T = (\log(p+1) - 1)(\alpha + n\beta)$



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:

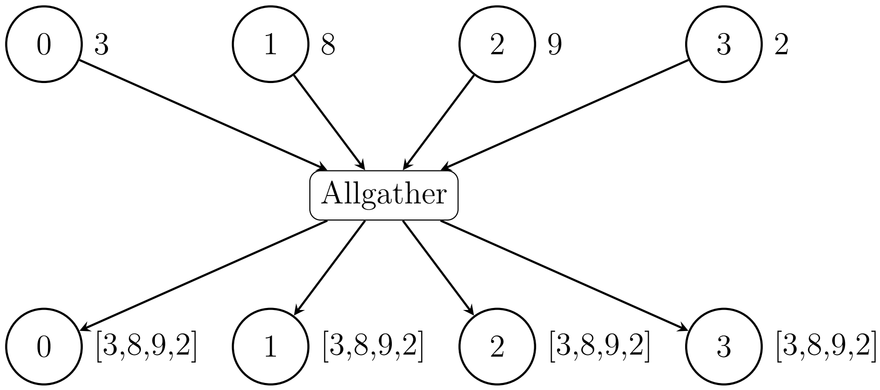
$$T = (\log p)\alpha + n\beta$$

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

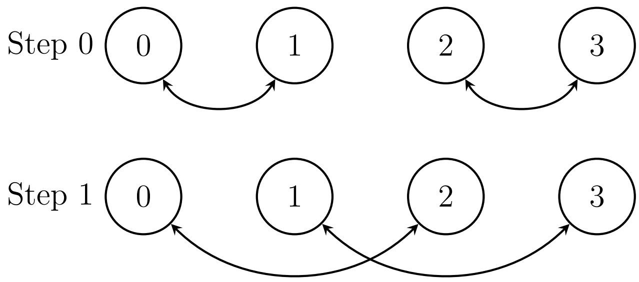
The **Allgather** is a collective operation that takes data from every process, combine them in a vector and copies the resulting vector to al the processes.



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.

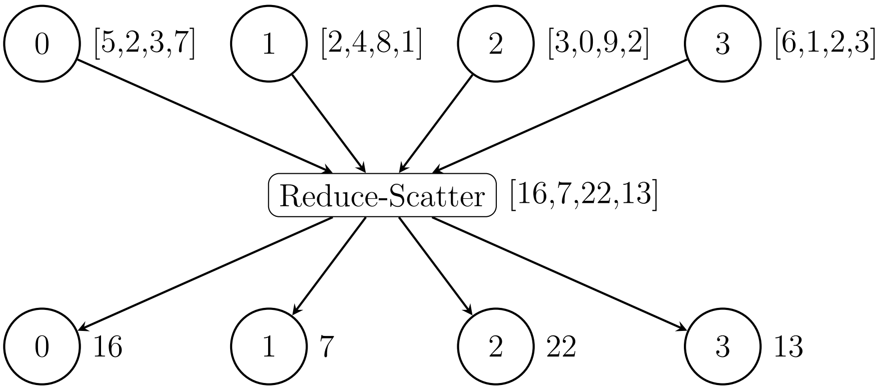


The cost is:

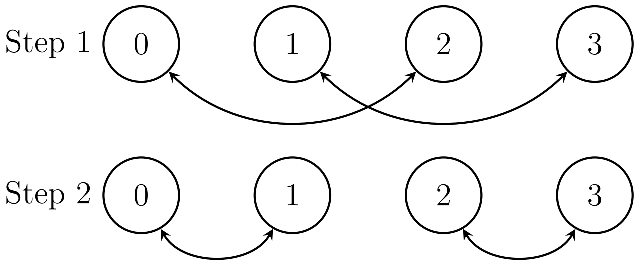
$$T = (\log p)\alpha + n\beta$$

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.

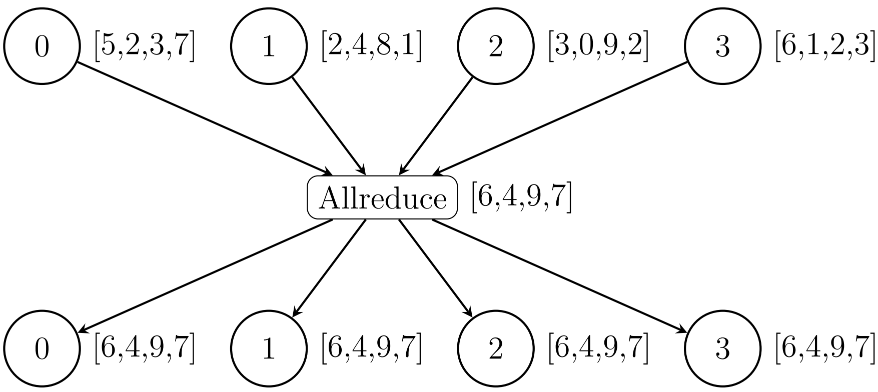


The cost is equal to the Recursive doubling algorithm of the Allgather:

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

2.3.4 Allreduce

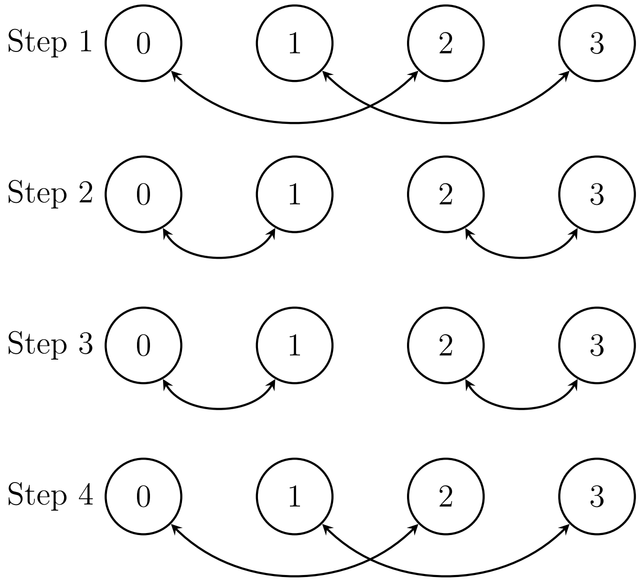
The **Allreduce** is a collective communication that takes a vector from every process, execute an operation on them and the resulting vector is copied in every process.



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.

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

$$\xrightarrow{\text{All-to-All}}$$

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

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.

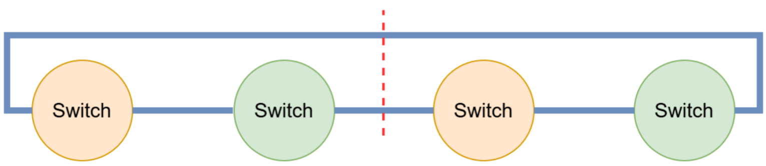
To start we need to explain some definitions:

- 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)

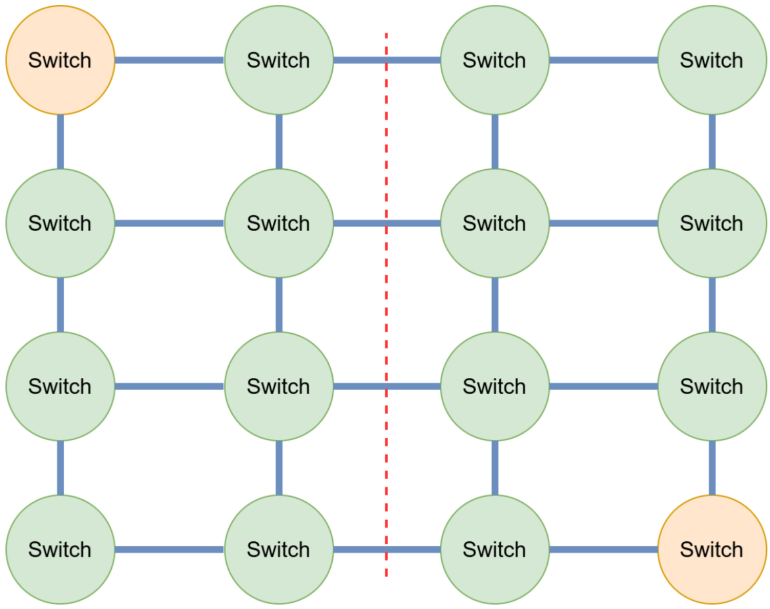
The **Ring** topology is a topology in which every server needs only two links, one to the previous server and one to the successive server.



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

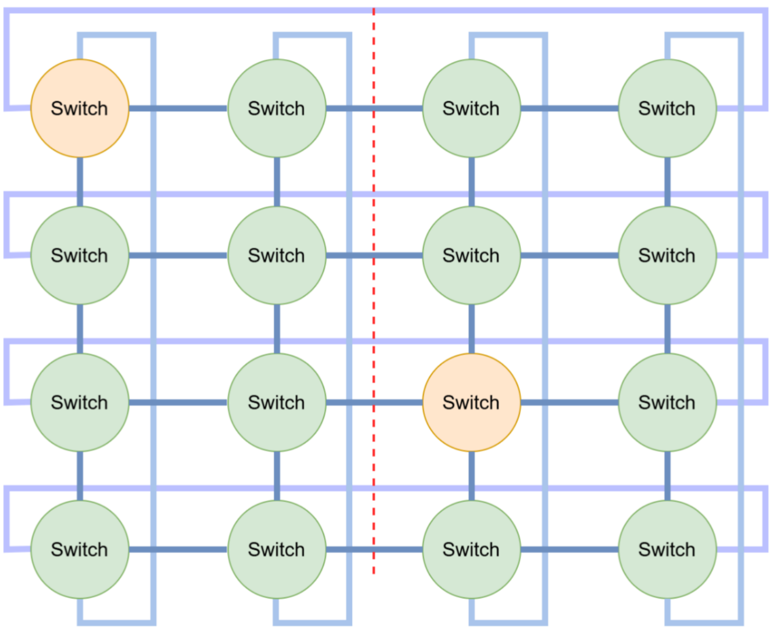
3.1.2 Mesh 2D

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



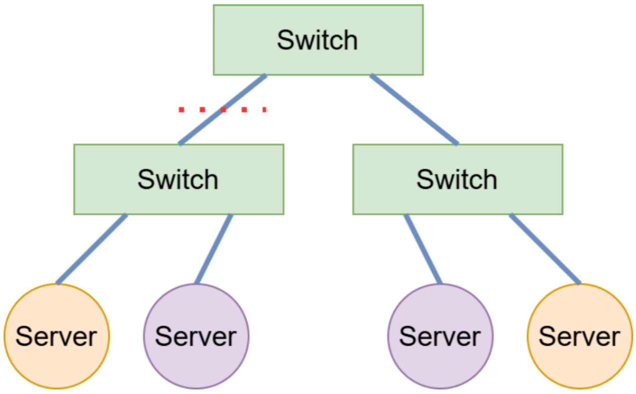
3.1.3 Torus 2D

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

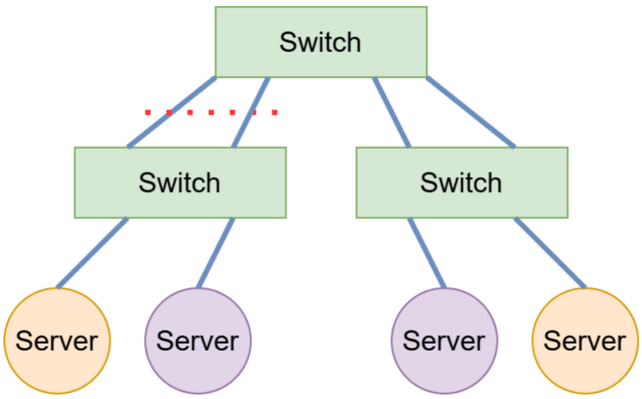
The tree topologies are based on the switch radix r. The basic tree has an height $h = \log_{r-1}(n)$, a diameter of $2h$ and a bisection cut of 1 (doesn't make much sense if the root has more than two child).



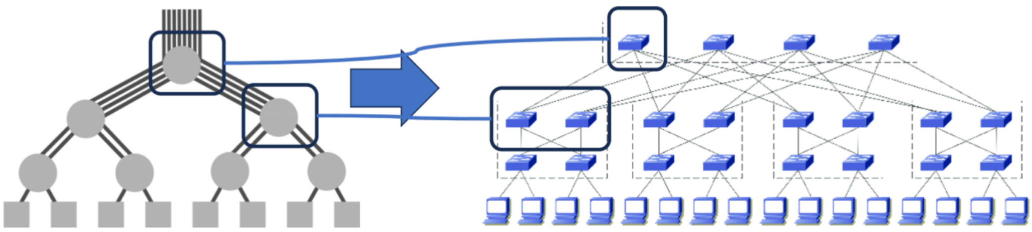
The basic tree is a blocking topology, because the switch have only one link going up.

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.

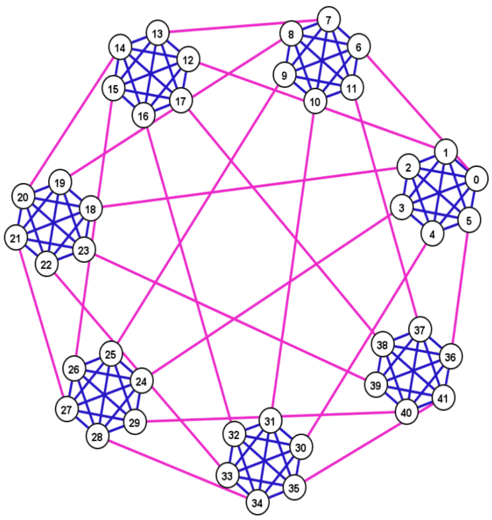


This network, also called **Folded CLOS** is the best that can be built in terms of performance but is also the most expendive one.

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.

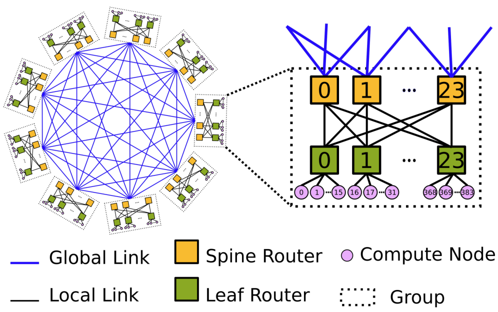


This topology has some disadvantages:

- 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

4.1 Storage

Generally with storage we mean data that can't be accessed directly by the CPU, instead memory is the data addressed from the CPU.
The storage used for Big Data can have various forms:

- Distributed File System (DFS): manage large files on multiple nodes.
- NoSQL data stores: non-relational data models like key-value or graph. Have orizontal scalability e fault tolerance.
- NewSQL databases: relational databases, but scalabale and with foult tolerance.

4.1.1 Distributed File System

A famous DFS is the **Google File System (GFS)**, that is built with the assumpion that there are hardware fails and multiple sequential writes of data.
Needs to store very large files, that are splitted in chunks (64 or 128 MB). The chunks are stored in chunks server and avery chunks is replicated for fault tolerance and availability. The chunks are managed by a master node that stores the file metadata and manages the operation on chunks. To guarantee availability, shadow masters take its place if the master fails.
GFS have other additional features such as:

- Data integrity: chunks are divided in 64KB blocks and fro every block a checksum is computed and checked every time data is read.
- Load balancing: chunks are balanced across chunk servers.
- API: other than traditional operation, supports two special one: snapshot (istantaneous copy) and record append (atomically append).

To write on a file GFS has a series of steps:

1. Application issues a write
2. Client translates the write position into a chunk index and sends the request to the master node
3. The master node sends the client the locations of the replicas (one of the replicas is the primary replica)

4. The client sends the data to be written to all the replicas
5. The client tells the primary replica to start the write
6. The primary replica decides in which order the data must be written in the chunk (there might be data coming from different clients in different order in the buffer)
7. The primary replica communicates the write order to the other replicas
8. The secondary replicas acknowledge the end of the write
9. The primary replica acknowledge the end of the write to the client

GFS appends data to the file at least once atomically (application must cope with possible duplicates).
Another DFS is the **Hadoop Distributed File System (HDFS)**, that is essentially a GFS clone with few differences. The most important one is the erasure coding, that replace the replicas, saving space.
Erasure coding consist in splitting the data in chunks and storing them together with m parity chunks that are used to recover the chunks in case of corruption.

4.1.2 SQL Databases

Relational Databases management promise ACID properties:

- Atomicity: either all the statements in a transaction are executed, or the transaction is aborted without affecting the databases
- Consistency: the database is consistent both before and after a transaction
- Isolation: the result of incomplete/in-progress transactions are not visible to other in-progress transactions
- Durability: Once a transaction is committeed, it will remain committed even in case of a system failure

The main problem with relation databases is the scalability, because they were not designed for a distributed system. They can scale by using replication (data replicated on multiple nodes) or sharding (dividing data on multiple nodes) but either of them impact the performance.

4.1.3 NoSQL Data stores

NoSQL data stores supports flexible scheme and horizontal scaling. They tradeoff reliability for higher performance. Instead of ACID properties use BASE properties:

- Basically Available: the system is available most of the time and there could exist a subsystem temporarily unavailable
- Soft state: the system might not always be consistent
- Eventually consistent: at some point, the system converges to a consistent state

NoSQL data stores are easy to scale than SQL ones and have and higher performance for large data.

4.1.4 RDMA

RDMA (Remote Direct Memory Access) is a way of handling data sends and recieves without using the kernel, that occupies a CPU core.

In RDMA systems the Network Stack is divided: one part is done by the software and one by the NIC. Also, the NIC resources can be accessed directly by the user-space, bypassing the kernel.

To bypass the kernel when sending data, instead of doing the control operations (done by the kernel) for every send, all the checks are done only once and the other times are skipped.

The most used protocol for RDMA is Infiniband/RoCE that in the last version has an Infiniband transport protocol, but also uses UDP/IP for the network layer and Ethernet for the link layer to be able to communicate with a system outside that doesn't uses Infiniband.

To speed up the simple operation, like recieving a message, do a simple calculation and send it back, smartNIC are used. In SmartNIC a CPU is put in the NIC to perform the simple calculation without paying the cost of sending the data to the application and back.

4.2 MapReduce Programming Model

MapReduce is a programming model used for processing big datasets with parallel algorithm on a cluster. It uses a DFS to store data on multiple nodes and guarantee fault tolerance.

The MapReduce consist of steps:

1. Input: key-value pairs $\{(k_i, v_i)^*\}$
2. Map phase: $(k_i, v_i) \rightarrow \{(k_i', v_i')^*\}$

   Takes the input key-value pair and outputs a set of key-value pairs. The map function is called for every key-value pair in input.
3. Combine phase: $(k_i', \{v_i'^*\}) \rightarrow (k_i', v_i')$

   After the map task the output may contain many pair sharing the same key, if the operation is suitable a Combine phase can be added to pre-aggregate the value before the Shuffle phase.
4. Shuffle phase: $\{(k_i', v_i')^*\} \rightarrow \{(k_i', \{v_i'^*\})^*\}$

   Collect all the pairs with the same key and groups the values associated.
5. Reduce phase: $(k_i', \{v_i'^*\}) \rightarrow (k_i', v_i'')$

   The reduce function is called for every key-value pair at the end of the shuffle phase.

The MapReduce is good for problems that require man sequential data access and large batch jobs, but is not good for problems with random access to data and indipendent data. Also, not every application can be expressed with a MapReduce.

In MapReduce input and output are stored on the DFS, instead intermediate results of map and reduce function are stored in the local filesystem of each node.

The master node takes care of coordination, propagating informations on finished tasks between mappers and reducers. It also pings nodes to detect failures and reschedule the task on other nodes if the fail occurred.

To decide how key-value pair are shuffled and sent to reducers a Partition Function is used. With R reducer nodes a key k is sent to reducer r with a function:

$$r = \text{hash}(k)\%R$$

4.3 Data-Flow Systems: Spark

In MapReduce the tasks have only two ranks: one for map and one for reduce. **Data-Flow** systems allow any number of ranks and other function than map and reduce.
The most popular Data-Flow System is **Spark**, a computing framework that provides a lot of functions, fast data sharing and execution graphs. It's also compatible with Hadoop. Spark is a fault-tolerant system with in-memory caching that provides efficient execution of multi-round algorithms.
The driver process (master in MapReduce) runs the application from a node in the cluster and its job is to maintain information about the application, respond to user input and manage the works across executors. The executor process (worker in MapReduce) run the code assigned and reports the state back to the driver. The cluster manager controls physical machines and allocates resources.

4.3.1 Resilient Distributed Datasets (RDD)

The fundamental abstraction of Spark are RDDs, a collection of elements of the same type. The works in Spark are expressed as transformation and operations on RDDs.
Each RDD is split in partitions distributed across the nodes, the program can specify the number of partition. Tipically is 2 or 3 times the number of cores to be sure every core is used but the partitions are not too small.
RDDs are immutable, for multiple purpose:

- Parallelism: the data that a process is reading can't change
- Caching: there is no need for consistency
- Fault tolerance: RDD maintain a trace of the transformation done and an RDD can be re-calculated if the node fails

RDDs can be created from data storage or from the transformation of other RDDs. With the property of being recalculable the intermediate RDDs could not be saved if they are used few times. If they are used multiple times is better to save them.
On an RDD A three types of operation are possible:

- Transformation: generate another RDD B from the data of RDD A
- Action: computation on the RDD A that returns a value to the application
- Persistence: save the RDD A for later

The transformations on RDDs can require to shuffle data across nodes:

- Narrow transformation: each partition of an RDD A contributes to one partition of the RDD B, so there is no need to shuffle data.

- Wide transformation: each partition of A contributes to multiple partition of B, so a shuffle of data is needed.

Spark uses the **lazy evaluation**, means that nothing is computed until is used in an action. When an action is triggered on an RDD, Spark builds a graph of operations that need to be exeuted to compute the action.

4.3.2 DataFrame and Dataset

Spark provides two interfaces to operate on structured data: DataFrame API and Dataset API.
A DataFrame is a distributed collection of data organized in named columns, that allow higher abstraction than RDD. Like RDDs, DataFrames are immutable and the operation executable on DataFrames are transformations and actions. Datasets API is a more general extension of DataFrame API.
DataFrames have higher performance than RDDs due to optimizations of the query plan.

4.3.3 Spark components

Spark has different components to work with differnt types of datas:

- Spark SQL: component for working with structured data like relational DB, CSV and JSON. Can execute queries and create SQL DataFrames.
- GraphX: component for working with graphs.
- Streaming: component that enables scalable, high-throughput, fault-tolerant stream processing of live data streams.

Spark Streaming divides data streams into micro-batches created at regular time intervals and process every batch like an RDD. The output is also pushed out in batches. The basic abstraction to work with data streams is **Discretized Stream (DStream)** that is implemented as a sequence of RDDs (so is immutable), each one containing data from a certain interval. The DStream API is very similar to the RDD one and also the transformations are very similar.
The transformations can be stateless, if the process of each batch is indipendent to the previous ones, or stateful, if uses data from previous batches.
The output operations push out the DStream data to external system (like actions for RDDs, they trigger the computation).
An important type of operation when working with data streams are window operations, that apply a transformation over a sliding window of data. Two parameter are specified: the window length (duration of the window) and the sliding interval (interval at which the operation is performed). The parameters must be multiples of the batch interval.

5 Similarity

5.1 Introduction

5.1.1 Search problem

A common problem that occurs is to find a value x in a list of n items. This can be done in two ways:

- List unsorted: $O(n)$
- Sort list and binary search: $O(n \log n) + O(\log n)$

The best case scenario depends on n and on the number of search that we need to do.
Let m be the number of search, the cost of the two approach is:

- List unsorted: $O(mn)$
- Sort list and binary search: $O(n \log n) + O(m \log n)$

So, is convenient to sort if:

$$n \log n + m \log n < mn \implies m \geq \left\lceil \frac{n \log n}{n - \log n} \right\rceil$$

5.1.2 Vertical and Horizontal scaling

There are two ways in which the data can scale:

- Vertical scaling: the number of items n is very large, couldn't even fit in memory. We will assume that this happens by design.
- Horizontal scaling: a single value of the list is complex to represent, for example images. The problems shift to find the most similar image to the input one.

5.2 Similarity meaning

Similar can mean different things depending on the domain and the representation we are using. We assume data lives in a d-dimensional Euclidean space. Similarity can be compute with different metrics.

Metric and Metric Space

Given a set X, δ is a **metric** if is function $\delta : X \times X \to [0, \infty)$ where:

1. $\delta(x, y) \geq 0$ (non-negativity)
2. $\delta(x, y) = 0 \iff x = y$ (identity of indiscernibles)
3. $\delta(x, y) = \delta(y, x)$ (simmetry)
4. $\delta(x, y) \leq \delta(x, z) + \delta(z, y)$ (triangle inequality)

X is the **metric space**.

5.2.1 Euclidean Space

Given $X = \mathbb{R}^d$, for two points $x, y \in \mathbb{R}^d$ the Euclidean distance is defined as:

$$\delta(x, y) = \sqrt{\sum_{i=1}^{d} (x_i - y_i)^2}$$

The Euclidean distance is also referred to as the L^2 norm of the displacement vector:

$$||x - y||_2 = \sqrt{(x - y)(x - y)}$$

The Euclidean distance can e generalized by the **Minkowski distance** (L^p norm), that is defined as:

$$\delta_p(x, y) = \left(\sum_{i=1}^{d} ||x_i - y_i||^p \right)^{\frac{1}{p}}$$

The important value of p for this formula are:

- $p = 1$ (Manhattan distance):

$$\delta_1(x, y) = \sum_{i=1}^{d} ||x_i - y_i||$$

- $p = 2$ (Euclidean distance):

$$\delta_2(x, y) = \sqrt{\sum_{i=1}^{d} ||x_i - y_i||^2}$$

- $p \to \infty$ (Chebyshev distance):

$$\delta_\infty(x, y) = \max_{i \in [1,d]} (||x_i - y_i||)$$

5.2.2 Cosine Similarity

Measures the angles between two vectors, the output has range $[-1, 1]$. It represents the orientation difference, not the magnitude.
Is defined as:

$$\cos(\theta) = \frac{xy}{||x||_2 ||y||_2}$$

where θ is the angle between the two vectors.

5.2.3 Jaccard index

Measures the similarity between two finite sets. Is defined as:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

The output has range $[0, 1]$. The Jaccard distance is the complementary of the index:

$$\delta_J(A, B) = 1 - J(A, B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|}$$

5.3 Clustering

Clustering is the procedure to group a set of objects into classes of similar objects. Is an unsupervised learning technique because there is no external influence other than the data. Is a method of data exploration, a way to look for patterns in data.
Given a set of data points and a notion of distance, groups data in such a way that:

- Members of a cluster are similar to each other
- Members of different clusters are dissimilar to each other

The problem with clustering is with hih-dimensional spaces, where almost all pairs are at the same distance.

5.3.1 Document clustering

To group document on the same topic we work in the space of words. Document with similar words probability talk about the same topic.
There are different ways to represent document in the space of words:

1. Set of words: we only count if a word appear or not in the document
2. Bag of words: we count the multiplicity of words in the document
3. Bag of n-grams: we count the group of n consecutive words that appear

In the set of words method two document are simply described as the set of all the words that appear in the document.
In the bag of words method a vocabulary is used, that contains the list of all the possible words contained in all documents. Each document is a vector in which the index i indicates the occurrences of the i-th word in the vocabulary.
If we define:

- $D = \{d_1, \dots, d_n\}$: the collection of n documents
- $V = \{w_1, \dots, w_m\}$: the vocabulary of m words
- $d_i = (f(w_1, i), \dots, f(W_m, i))$: the vector representing d_i

- $f : V \times D \to \mathbb{R}$: the function that maps every word to a value

We can define the function f in different ways:

- One-Hot binary weighting scheme:

$$f(w_j, i) = \begin{cases} 1 & w_j \text{ appears in } d_i \\ 0 & \text{otherwise} \end{cases}$$

- Term-frequency weighting scheme:

$$f(w_j, i) = tf(w_j, i)$$

  in which tf calculates the number of times w_j appear in d_i.

- TF-IDF weighting scheme:

$$f(w_j, i) = tf(w_j, i) \cdot idf(w_j)$$

  in which idf is defined as:

$$idf(w_j) = \log\left(\frac{n+1}{n_j+1}\right)$$

  in which n_j is the number of documents containing the word w_j.

Depending on the representation used the similarity method that can be used is different:

- Set of words: Jaccard distance
- One-Hot bag of words: Euclidean distance
- TF or TF-IDF bag of words: Cosine similarity

5.3.2 High-Dimensional Spaces

Is easy to reach a very high-dimensional space, where only few dimensions are non zero. Data tends to be more sparser as the number of dimension grows. For example, the Euclidean distance of two points is high if they are far in at least one dimension (one of the dimension $x_i - y_i$ dominates the formula). The higher the number of dimensions, the higher the probability that this occurs.
We take an unit-length (length 1 in every dimension) hypercube H with d dimensions and N points randomly in the cube. We choose a specific point p and we want to create a smaller cube h centered in p that contains the k nearest point to p. The cube has length l in every dimension and every points has distance smaller than $\frac{l}{2}\sqrt{d}$ (the diagonal of the small cube).
The volume of the smaller hypercube is $V_h = l^d$ and contains $\frac{k}{N}$ points, so:

$$l^d \simeq \frac{k}{N} \implies l \simeq \left(\frac{k}{N}\right)^{\frac{1}{d}}$$

With this formula we can clearly see that the length of the hypercube h grows exponentially as the number of dimension grows.

This is caused by the edges of the hypercube, with more dimensions the probability to not be on the edge in at least one dimension is low. Let ϵ define the edge, the probability of any point to not be on the edge with 1 dimension is:

$$P(\text{not edge}) = (1 - 2\epsilon)$$

With more dimensions the probability become smaller and with d dimensions:

$$P(\text{not edge}) = (1 - 2\epsilon)^d$$

This problem can be resolved by assuming that in the real world the data is not random, but have a pattern underneath. The **Manifold hypothesis** suggests that high-dimensional data lie on low-dimensional subspace (manifold) embedded in the high-dimensional space. For example, for a digit recognition on 20×20 images taking in account all 400 dimensions model every random patter of pixel on the image. In reality the digits cover a small fraction of all the space and can be modelled in a much simpler way.