

SPM - Distributed

Lorenzo Leuzzi

Message Passing Interface (MPI)	1
Introduction	1
Key Concepts	2
Commands	2
Collective Communication	3
Reduce and Allreduce	4
Groups and Communicators	5
Hadoop and its Filesystem (HDFS)	6
Introduction	6
MapReduce	6
Hadoop File System	7
Read	9
Write	9
Hadoop MapReduce	9
Examples	10
Apache Spark	11
Introduction	11
Resilient Distributed Dataset (RDD)	12
Shared Variables	13
GraphX	13
Actor Model	14
Introduction	14
CAP Theorem	15
PACEL	16
Types of Consistency	17
Akka	18
Why actor model?	18
Introduction	18
Actors in Akka	19

Message Passing Interface (MPI)

Introduction

Distributed computing is performed by different computers coordinating one another to reach altogether a common goal.

But this is not that easy:

- Work has to be coordinated: synchronization is needed and communications need to be managed.
- Data has to be consistent and accessible in a distributed context.
- Failures in Nodes and Networks need to be managed.

Key Concepts

A **communicator** in parallel computing defines a **group of processes** that can communicate with each other. Each **process** in the group is assigned a unique rank and communicates explicitly with others using these ranks. Communication involves send and receive operations, where a process sends a message by specifying the recipient's rank and a unique tag. The receiver can then handle the data accordingly. This type of communication, involving one sender and receiver, is called **point-to-point communication**.

Sometimes, processes need to communicate with all others in the group, such as when a manager process needs to **broadcast** information to all worker processes. Writing individual send and receive code for this would be cumbersome and suboptimal for network usage. MPI (Message Passing Interface) can efficiently handle various **collective communications** that involve all processes in the group. Combining point-to-point and collective communications allows the creation of complex parallel programs.

Commands

During **MPI_Init**, MPI initializes its global and internal variables. This process includes forming a communicator that encompasses all spawned processes and assigning unique ranks to each process.

MPI_Comm_size returns the size of a communicator.

MPI_Comm_rank returns the rank of a process in a communicator.

MPI_Get_processor_name obtains the actual name of the processor on which the process is executing.

MPI_Finalize is used to clean up the MPI environment.

```

MPI_Send (    void* data,
              int count,
              MPI_Datatype datatype,
              int destination,
              int tag,
              MPI_Comm communicator )

```

```

MPI_Recv (    void* data,
              int count,
              MPI_Datatype datatype,
              int source,
              int tag,
              MPI_Comm communicator,
              MPI_Status* status)

```

Almost every MPI call uses a similar syntax

- The first argument is the data buffer.
- The second and third arguments describe the count and type of elements that reside in the buffer. MPI_Send sends the exact count of elements, and MPI_Recv will receive at most the count of elements. MPI_Send and MPI_Recv functions utilize MPI Datatypes as a means to specify the structure of a message at a higher level.
- The fourth and fifth arguments specify the rank of the sending/receiving process and the tag of the message.
- The sixth argument specifies the communicator.
- The last (seventh) argument (for MPI_Recv only) provides information about the received message.

Example

```

#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);
    // Find out rank, size
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // We are assuming at least 2 processes for this task
    if (world_size < 2) {
        fprintf(stderr, "World size must be greater than 1 for %s\n", argv[0]);
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    int number;
    if (world_rank == 0) {
        // If we are rank 0, set the number to -1 and send it to process 1
        number = -1;
        MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    } else if (world_rank == 1) {
        MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 1 received number %d from process 0\n", number);
    }
    MPI_Finalize();
}

```

MPI_Barrier is used to synchronize all of the processes in an MPI communicator.

Collective Communication

There are ways to ease the task of communicating with a group of processes.

MPI_Bcast is a collective communication operation in MPI used to **broadcast** a message from one process (known as the root) to all other processes within a communicator. Every process in the communicator invokes `MPI_Bcast()`, but only the root process initiates the message broadcast. Non-root processes pause until they receive the message. The call only completes when all processes have received the message, including the root process. This ensures that all processes attain an identical final state after the operation.

`MPI_Bcast` is typically used to distribute the same data to all processes in a communicator, making it useful for scenarios where every process needs the same information.

```
MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
```

MPI_Gather is a collective communication operation in MPI where processes send messages to a root process within a communicator. Every process in the communicator invokes `MPI_Gather()`, but only one process can serve as the root. The root process collects messages from all other processes in the communicator.

`MPI_Gather` is suitable when you want to collect data from various processes and bring it together into a single process for further analysis or processing. The result will be an array of data at the root process, where each element corresponds to the data from a different process.

```
int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

MPI_Scatter sends data from one process to all other processes in a communicator.

In `MPI_Scatter`, all processes in the communicator participate, but only the root process provides the data to be scattered. The root process divides its data into equal-sized portions, sending each portion to the corresponding process in the communicator. Each process receives its designated portion, which is typically stored in a receive buffer. Once the `MPI_Scatter` operation is complete, each process has its own part of the data, allowing for independent computations or analysis on the received data.

```
int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

Reduce and Allreduce

Reduce is a way to merge values. Data reduction involves reducing a set of numbers into a smaller set of numbers via a function. MPI has a handy function called **MPI_Reduce** that will handle almost all of the common reductions that a programmer needs to do in a parallel application. `MPI_Reduce` is commonly used for operations like summation, multiplication, finding the minimum or maximum value, or performing other user-defined reductions.

It works in this way:

1. All processes within the communicator call `MPI_Reduce`, providing their own local data values to be combined. These data values are often referred to as "operands."
2. The root process is specified as part of the `MPI_Reduce` call, and this process is the one that will receive the final result of the reduction operation.
3. `MPI_Reduce` performs the reduction operation on the operands from all processes. The type of reduction operation (e.g., addition, multiplication) is specified in the `MPI_Reduce` call.
4. Once the reduction is completed, the result is stored at the root process. Other processes receive no result value.

```
MPI_Reduce(void* send_data, void* recv_data, int count, MPI_Datatype  
datatype, MPI_Op op, int root, MPI_Comm communicator)
```

Many parallel applications will require accessing the reduced results across all processes rather than the root process. Unlike `MPI_Reduce`, which reduces data to a single value at the root process, `MPI_Allreduce` ensures that all processes in the communicator receive the final result of the reduction operation.

```
MPI_Allreduce(void* send_data, void* recv_data, int count,  
MPI_Datatype datatype, MPI_Op op, MPI_Comm communicator)
```

Groups and Communicators

Since now, `MPI_COMM_WORLD` communicator has been used, suitable for smaller applications where all processes communicate with each other or with one process at a time. However, as applications grow, communicating with all processes simultaneously becomes less practical. To address this, new communicators can be created to interact with specific subsets of the original process group, enabling more targeted communication.

`MPI_Comm_split` creates new communicators by "splitting" a communicator into smaller, more specialized communicators based on the input values `color` and `key`. This function is particularly useful when you want to group processes with similar characteristics or tasks together, enabling more focused communication within subsets of the original process group. It's important to note here that the original communicator doesn't go away, but a new communicator is created on each process.

```
MPI_Comm_split( MPI_Comm comm, int color, int key, MPI_Comm*  
newcomm)
```

```

// Get the rank and size in the original communicator
int world_rank, world_size;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int color = world_rank / 4; // Determine color based on row

// Split the communicator based on the color and use the
// original rank for ordering
MPI_Comm row_comm;
MPI_Comm_split(MPI_COMM_WORLD, color, world_rank, &row_comm);

int row_rank, row_size;
MPI_Comm_rank(row_comm, &row_rank);
MPI_Comm_size(row_comm, &row_size);

printf("WORLD RANK/SIZE: %d/%d \t ROW RANK/SIZE: %d/%d\n",
world_rank, world_size, row_rank, row_size);

MPI_Comm_free(&row_comm);

```

While `MPI_Comm_split` is the simplest way to create a new communicator, it isn't the only way to do so. There are more flexible ways to create communicators, they use `MPI_Group`. MPI uses these groups in the same way that set theory generally works. An `MPI_Group` is an ordered set of processes, and it represents a logical collection of processes in an MPI communicator.

`MPI_Comm_group` is a function that is used to create an `MPI_Group` object from an existing communicator. This function allows you to extract the group of processes associated with a communicator.

```
MPI_Comm_group( MPI_Comm comm, MPI_Group* group)
```

There are two kinds of operations when working on groups and communicators.

Local Operations: Operations on groups are considered local because they do not involve communication with other ranks or processes. When you create, manipulate, or work with a group, it is specific to the local process and does not require coordination or agreement with other processes. You can perform these operations independently on each process without the need for communication.

Remote Operations: In contrast, operations on communicators, such as creating a new communicator, are considered remote operations. These operations require communication and coordination among all processes involved because communicators define a communication context that all processes within that communicator share. All processes must agree on the same communicator, and this involves communication and synchronization among them.

You can do union and intersection of two groups generating a new one.

```
MPI_Group_union( MPI_Group group1, MPI_Group group2, MPI_Group*
newgroup)
MPI_Group_intersection( MPI_Group group1, MPI_Group group2,
MPI_Group* newgroup)
```

`MPI_Comm_create_group` creates a new communicator from a group.

```
MPI_Comm_create_group( MPI_Comm comm, MPI_Group group, int tag,
MPI_Comm* newcomm)
```

`MPI_Group_incl` is a function that allows you to pick specific ranks in a group and construct a new group containing only those ranks.

```
MPI_Group_incl(MPI_Group group, int n, const int ranks[], MPI_Group* newgroup)
```

Hadoop and its Filesystem (HDFS)

Introduction

In 2005, the rapidly increasing volume of data from the internet and other sources posed challenges for organizations in terms of storage, processing, and analysis. Traditional file systems and databases were ill-suited to handle this massive and complex data, often being both costly and hard to manage. Thus the motivation behind the development of HDFS was to provide a scalable, reliable, and cost-effective solution for storing and managing large datasets, and to make these capabilities accessible to a broader range of organizations and individuals. The key idea is to split your huge task into many smaller ones, have them execute on many machines in parallel and aggregate the data appropriately.

MapReduce

MapReduce is a programming model and processing framework that was popularized by Google for processing and generating large-scale data sets in a parallel and distributed manner. It is designed to handle massive amounts of data across a cluster of computers, enabling the processing of large datasets efficiently and reliably.

Here's how MapReduce works:

1. **Map Phase:** In this phase, the input data is divided into smaller chunks and distributed to multiple worker nodes in a cluster. Each worker node independently processes its portion of the data and applies a user-defined "map" function to it. This function transforms the data into a set of key-value pairs.
2. **Shuffle and Sort:** After the mapping phase, the MapReduce framework automatically groups all key-value pairs with the same key together and sorts them. This step is crucial as it prepares the data for the next phase.
3. **Reduce Phase:** In the reduce phase, another user-defined "reduce" function is applied to the sorted and grouped key-value pairs. The reduce function takes in a key and the associated list of values and performs some computation or aggregation on them to produce the final output.

MapReduce is fault-tolerant, meaning it can handle node failures and data recovery automatically. It is highly parallelizable and can scale to process extremely large datasets by distributing the workload across a cluster of computers. While it was initially developed by Google, the concept of MapReduce has been widely adopted in the big data ecosystem, and various implementations and tools, such as Apache Hadoop, have been created to facilitate large-scale data processing using the MapReduce model.

MapReduce, while once a widely used and effective data processing model, is considered "legacy" in contemporary data processing contexts. It has some limitations that can be

problematic in today's data processing environments. One significant issue is related to its batch processing nature. MapReduce processes data in batches or jobs. This means that if you're running a MapReduce job, and it fails for any reason (e.g., a hardware failure, a software bug, or data corruption), you'll need to restart the entire job from the beginning. This batch-oriented approach can be inefficient and time-consuming, especially when dealing with large datasets. If a job takes a long time to complete, the risk of failure increases, and restarting the entire process can be a cumbersome and resource-intensive task. Additionally, in real-time or near-real-time data processing scenarios, the batch processing model may not be suitable because it doesn't provide immediate results. As a result, modern data processing frameworks and models have emerged that address these limitations, such as stream processing frameworks like Apache Kafka, Apache Flink, and Apache Spark Streaming, which can handle data in smaller, more continuous chunks and provide fault tolerance and resilience without the need to restart from scratch in case of failures. These newer approaches are better suited to handle the dynamic and real-time nature of today's data processing needs.

Hadoop File System

Hadoop File System, commonly referred to as HDFS (Hadoop Distributed File System), is a distributed file storage system designed to store and manage large volumes of data across clusters of commodity hardware. It is a fundamental component of the Apache Hadoop framework, which is widely used for distributed data processing and big data analytics.

The Hadoop Distributed File System (HDFS) has a distributed and hierarchical architecture designed to provide high availability, fault tolerance, and scalability for storing and managing large datasets. Here's an overview of the key components and their roles in the architecture of HDFS:

NameNode: The NameNode is the master server in the HDFS architecture and acts as the central directory and metadata repository. It keeps track of the structure and organization of the file system, including information about file names, directories, permissions, and the location of data blocks. The NameNode does not store the actual data content of files; it only manages metadata. It is a single point of failure in the system, so its availability and reliability are crucial.

DataNodes: DataNodes are worker nodes in the HDFS cluster that store the actual data blocks. Each DataNode periodically sends heartbeat signals to the NameNode to confirm its health and availability. DataNodes report block information and perform block replication as directed by the NameNode to maintain data redundancy.

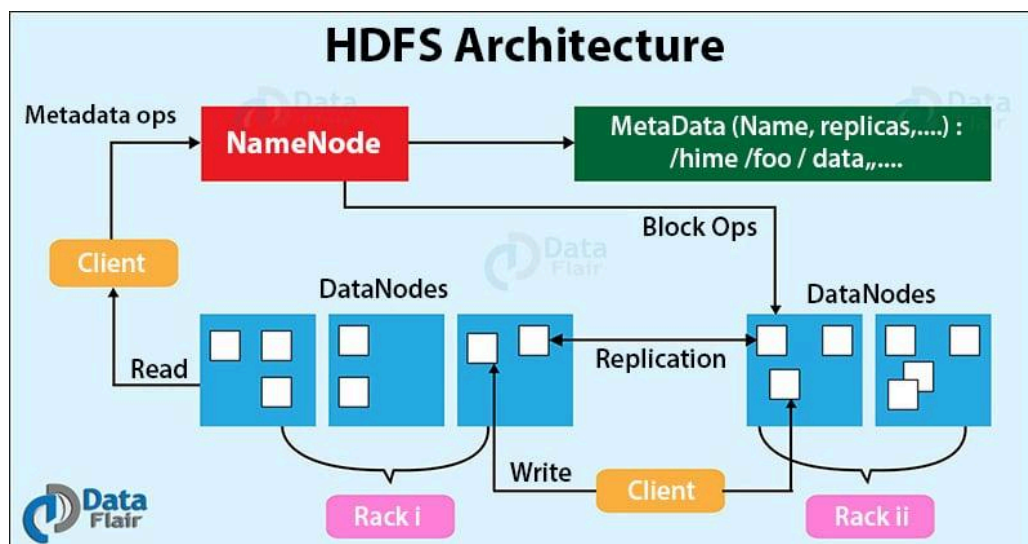
Block: Data in HDFS is divided into fixed-size blocks (e.g., 128MB or 256MB). These blocks are distributed across the DataNodes in the cluster. HDFS replicates each block to multiple DataNodes (usually three by default) to ensure fault tolerance. Block is of fixed size, easy to manage and manipulate. Because of blocks, working with many small files in Hadoop is messy.

Client: Clients interact with HDFS to read and write data. When a client wants to read or write a file, it communicates with the NameNode to obtain metadata (e.g., file structure) and then interacts directly with the appropriate DataNodes to read or write the data blocks.

Secondary NameNode: Despite its name, the Secondary NameNode is not a backup for the NameNode. It periodically performs checkpoints by merging the edit logs (changes to the file system metadata) with the file system image (snapshot of metadata) to create a new file system image. This operation helps reduce the recovery time in case of NameNode failure.

Rack Awareness: HDFS is designed to be aware of the physical network topology of the cluster. It tries to place replicas of data blocks on different racks to improve data reliability and availability in the event of rack failures.

Block Replication: HDFS replicates data blocks across multiple DataNodes to ensure fault tolerance. The replication factor is typically set to three, meaning each data block is stored on three different DataNodes.



Read

1. Client connects to NameNode (NN) to read data
2. NN tells client where to find the data blocks
3. Client reads blocks directly from data nodes (without going through NN)
4. In case of node failures, client connects to another node that serves the missing block

Not asking clients to read blocks through NN but directly from DN:










- prevents NN from being the bottleneck of the cluster
- allows HDFS to scale to a large number of concurrent clients
- spreads the data traffic across the cluster.

Write

1. Client connects to NN to write data
2. NN tells client write these data nodes

3. Client writes blocks directly to data nodes with desired replication factor
4. In case of node failures, NN will figure it out and replicate the missing blocks

Replication Strategy Tradeoffs

	Reliability	Write Bandwidth	Read Bandwidth
Put all replicas on one node			
Put all replicas on different racks			
HDFS: First -> same node as client Second -> a node on different rack Third -> a different node on the same rack as 2			

Hadoop MapReduce

At the core of Hadoop's architecture is the MapReduce Engine, which is responsible for processing large-scale data tasks. Here's how it works:

Job Submission: The MapReduce Engine consists of a central component called the JobTracker. Client applications submit their MapReduce jobs to this JobTracker.

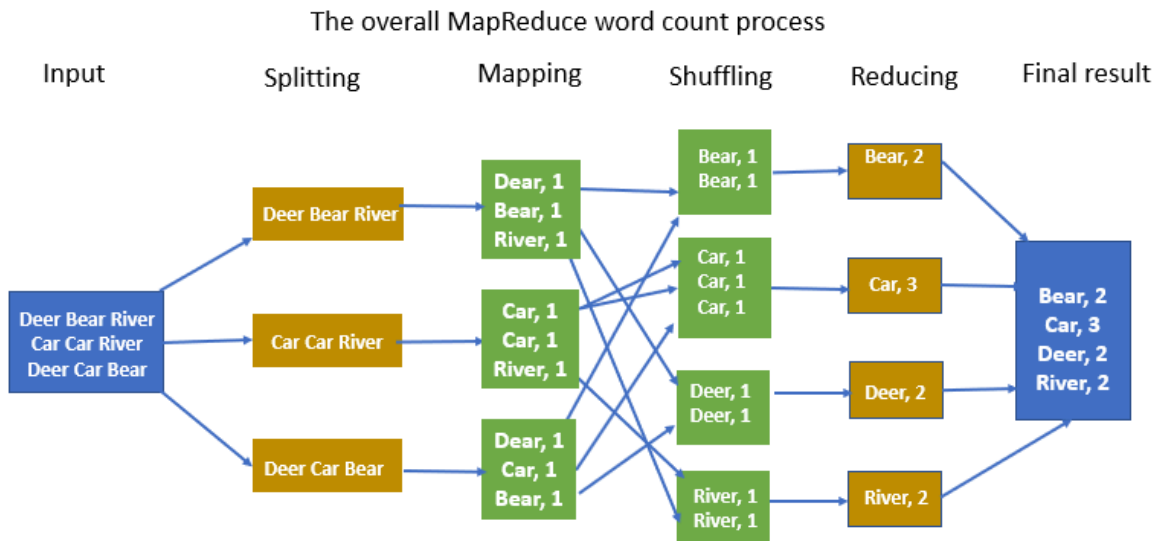
Work Distribution: The JobTracker's role is to distribute the work to available TaskTracker nodes within the Hadoop cluster. It aims to keep the processing tasks as close as possible to the data they need to work on.

Data Locality: Hadoop assumes a rack-aware file system, meaning it is aware of the physical network topology of the cluster. The JobTracker knows which node contains the required data. If the data isn't on the same node where the computation is running, it prioritizes nodes in the same rack to minimize data transfer over the network.

Fault Tolerance: If a TaskTracker node fails or takes too long to complete a task, the job is rescheduled on another available node. Each TaskTracker node runs tasks in separate Java virtual machine (JVM) processes, preventing a single task failure from affecting the entire node.

Heartbeats: TaskTracker nodes send periodic heartbeats to the JobTracker to keep it informed of their status. This communication ensures that the JobTracker can track the health and progress of tasks.

Monitoring: Users and administrators can monitor the status and information of both the JobTracker and TaskTracker nodes through a web browser interface, providing visibility into the processing tasks and the health of the Hadoop cluster.



There are some well-known limitations associated with this approach:

The process of allocating work to TaskTrackers is relatively straightforward. Each TaskTracker has a predefined number of available slots. Every active map or reduce task occupies one of these slots. The Job Tracker assigns tasks to the TaskTracker closest to the data that has an available slot.

This allocation method lacks consideration for the current workload on the allocated machine, which may impact its availability.

In cases where one TaskTracker operates at a notably slower pace, it has the potential to significantly delay the entire MapReduce job. This slowdown can occur because all tasks may end up waiting for the slowest one to complete. However, with speculative execution enabled, a single task can be executed on multiple slave nodes simultaneously, helping mitigate this issue by ensuring that the job continues to progress even if one instance is slow.

Examples

PageRank:

- PageRank is an algorithm used to rank web pages in search engine results based on their importance and popularity.
- In Hadoop MapReduce, you typically start by representing web pages and their links as a graph structure.
- The Map phase processes the graph, and each node redistributes its importance score to its neighboring nodes.
- The Reduce phase aggregates the importance scores from the Map phase and calculates the new PageRank for each web page.
- This process iterates until convergence, with each iteration being a new MapReduce job.

k-Nearest Neighbors (k-NN):

- k-NN is a machine learning algorithm used for classification or regression based on the proximity of data points.

- In Hadoop MapReduce, you start with a dataset containing data points and their features.
- The Map phase involves calculating distances between a test data point and all other data points in the dataset.
- The Reduce phase selects the k-nearest neighbors based on the calculated distances and makes a prediction or performs a classification.

Apache Spark

Introduction

Hadoop has been the first widely used solution. It is written in Java, easy to use with a lot of documentation but it's quite outdated so far. Alternative solutions exist nowadays: Apache Spark.

Apache Spark is an open-source, distributed data processing framework designed for big data processing and analytics. It is part of the Apache Software Foundation and has gained widespread popularity due to its speed, ease of use, and versatility. Spark provides a powerful and unified platform for various data processing tasks, including batch processing, real-time streaming, machine learning, and graph processing.

Key features and characteristics of Apache Spark include:

Speed: Spark is known for its speed and high performance. It achieves this through in-memory processing, which reduces the need to write data to disk between processing stages. This makes Spark significantly faster than traditional MapReduce-based frameworks like Hadoop.

In-Memory Computation: Spark leverages in-memory data caching, allowing it to store intermediate data in memory and reuse it efficiently across multiple stages of computation. This enhances performance for iterative algorithms and interactive queries.

Ease of Use: Spark offers APIs in multiple programming languages, including Scala, Java, Python, and R, making it accessible to a wide range of developers. Its high-level APIs simplify complex data processing tasks.

Versatility and Rich Ecosystem: Spark is a versatile framework that supports various data processing workloads, including batch processing (Spark Core), real-time stream processing (Spark Streaming), interactive querying (Spark SQL), machine learning (MLlib), and graph processing (GraphX). This allows users to build end-to-end data pipelines within a single framework.

Community and Support: Being an open-source project, Spark has a vibrant and active community, providing support, documentation, and a growing number of third-party packages and connectors.

Integration: Spark can seamlessly integrate with various data sources and storage systems, including Hadoop Distributed File System (HDFS), Apache Cassandra, Apache HBase, and more.

Apache Spark has become a preferred choice for organizations dealing with big data and complex data processing tasks. Its ability to handle a wide range of workloads and its performance advantages have made it a powerful tool in the field of data analytics and processing.

Resilient Distributed Dataset (RDD)

In Apache Spark, a Resilient Distributed Dataset (RDD) is a fundamental and immutable distributed data structure. RDDs are at the core of Spark's programming model and are designed to provide fault tolerance, parallel processing, and efficient data manipulation in a distributed computing environment. Formally, an RDD is a read-only, partitioned collection of records.

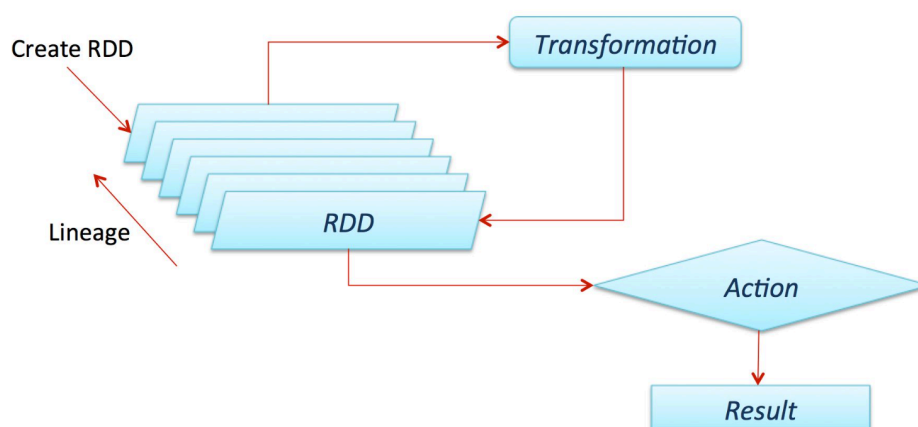
Resilient: RDDs are resilient because they automatically recover from node failures. If a partition of an RDD is lost due to a node failure, Spark can recompute that partition using the original data and the transformation operations applied to it.

Distributed: RDDs are distributed across a cluster of nodes in a Spark cluster. Each node processes a portion of the RDD in parallel, enabling scalable data processing.

Dataset: RDDs are a distributed collection of data elements. They can contain data of various types, including structured and unstructured data.

RDDs are **immutable**, which means that once created, their content cannot be modified. Instead, you can create new RDDs through transformations.

RDDs support two types of **operations**: transformations and actions. **Transformations** are operations that create a new RDD from an existing one. Examples of transformations include map, filter, and reduceByKey. Transformations on RDDs are lazily evaluated, meaning they are not executed immediately. Instead, Spark records the transformation operations and applies them only when an action is triggered. This lazy evaluation helps optimize the execution plan. Think of RDD as a set of instructions on how to compute the data via transformations. **Actions** are operations that trigger the execution of transformations and return a result to the driver program or write data to an external storage system. Examples of actions include count, collect, and saveAsTextFile.



You can create Resilient Distributed Datasets (RDDs) in Apache Spark using two methods. You can create an RDD by **parallelizing an existing collection** (e.g., a list or an array)

within your Spark driver program. This means you take data that's already in memory in your program and distribute it across the Spark cluster as an RDD for parallel processing. RDDs can also be created by **referencing datasets** stored in **external** storage systems, such as shared file systems, Hadoop Distributed File System (HDFS), HBase, or any data source that offers a Hadoop Input Format. Instead of loading data into your driver program, you simply reference the data source, and Spark will distribute the data across the cluster as an RDD.

Shared Variables

Shared variable abstractions are mechanisms that allow multiple tasks running on different nodes of a distributed cluster to share data efficiently and in a controlled manner. These shared variables are particularly useful when you need to perform certain types of operations that require coordination and data sharing among different stages or tasks of a Spark application. Spark supports two types of shared variables:

- Broadcast variables, which can be used to cache a value in memory on all nodes
- Accumulators, which are variables that are only “added” to, such as counters and sums.

GraphX

Important Big Data distributed applications deal with graphs. GraphX is a distributed graph processing framework and library that is part of the Apache Spark ecosystem. It is designed for large-scale graph processing and analytics, making it well-suited for applications involving social networks, recommendation systems, network analysis, and more. GraphX extends the capabilities of Apache Spark to handle graph data structures and graph-based computations efficiently.

GraphX introduces a vertex-centric programming model, where you define computation logic at the level of individual vertices and their neighboring vertices. This simplifies the development of graph algorithms. Each vertex maintains its state (i.e., its label and attribute) and can send messages to its neighboring vertices based on its state and the state of them. The messages are then received by the neighboring vertices, which can update their state accordingly.

A vertex RDD can be created from an RDD of tuples where the first element is the vertex ID and the second element is the vertex attribute.

An edge RDD can be created from an RDD of tuples where the first two elements are the source and destination vertex IDs and the third element is the edge attribute.

GraphX leverages both the Actor Model and the Bulk Synchronous Parallel (BSP).
Actor Model.

GraphX employs the actor model to portray graph vertices and conduct distributed graph processing tasks. Each individual vertex takes on the role of an actor, facilitating communication with other actors through message exchanges. This actor-based approach in GraphX provides a programming model for distributed graph processing that revolves around vertex-centric computations.

The computation in GraphX unfolds through a sequence of supersteps, where each superstep involves message passing and vertex computations. In each superstep

(Concurrent computation, Communication, Barrier synchronization), vertices receive messages from their neighbors and update their states accordingly. Subsequently, vertices exchange messages, ushering in the next superstep of computation.

Just as RDDs have basic operations like map, filter, and reduceByKey, graphs also have a collection of basic operators that take user defined functions and produce new graphs with transformed properties and structure. The core operators that have optimized implementations are defined in Graph and convenient operators that are expressed as a composition of the core operators are defined in GraphOps.

In GraphX, property operators are functions or operations applied to the properties or attributes associated with the vertices or edges of a graph, the graph structure is unaffected.

Actor Model

Introduction

The Actor Model is a mathematical and conceptual framework used in computer science and parallel computing for designing and reasoning about concurrent and distributed systems. It was first introduced by Carl Hewitt in the 1970s. Actors are based on “**behavior**” as opposed to the “class” concept of object-oriented programming

Actors are fundamental units of computation and concurrency. They are **autonomous entities** that can perform tasks independently. They communicate by sending **messages** to one another. These messages can contain data and instructions for the receiving actor. Each actor encapsulates its state and behavior. Other actors can only interact with it through message passing, and they have no direct access to the actor's internal state. An Actor can only communicate with the Actors to which it is connected. It can directly obtain information only from other Actors to which it is directly connected. Message passing is **asynchronous**, meaning an actor does not wait for a response when sending a message. It can continue processing other messages.

Actors can run **concurrently** and in **parallel** by definition. They can process messages concurrently, which makes the model suitable for building highly concurrent and distributed systems. The Actor Model is location-transparent, meaning actors can be distributed across different physical machines or run on a single machine without affecting the communication patterns.

The model naturally supports **fault tolerance** because each actor's state is isolated. If one actor fails, it does not impact the others.

The Actor model provides a **scalable** approach to building systems by allowing actors to be dynamically created and distributed across multiple machines.

This enables systems to scale horizontally by adding more machines to handle increasing loads.

The Actor model is **indeterminate** because the order in which messages are processed by actors is not fixed. When multiple messages are waiting in an actor's mailbox, the order in which they are processed is determined by the actor's scheduling policy, which may be non-deterministic. This means that the behavior of the system may depend on the order in which

messages are processed, and may therefore be unpredictable. The Actor model does provide guarantees around message delivery and processing. In particular, the model guarantees that messages will be delivered and processed **atomically** within an actor, which helps to avoid race conditions and other concurrency issues.

The Actor model is **quasi-commutative** because the order in which messages are sent between actors does not affect the final outcome of the system. This means that the Actor model can be used to build systems that are resilient to message delays, losses, and reordering.

However, it's important to note that quasi-commutativity is not a guarantee that the system will behave correctly in all circumstances. If messages must be processed in a particular order to ensure correct behavior, it may be necessary to introduce additional synchronization mechanisms to enforce that order.

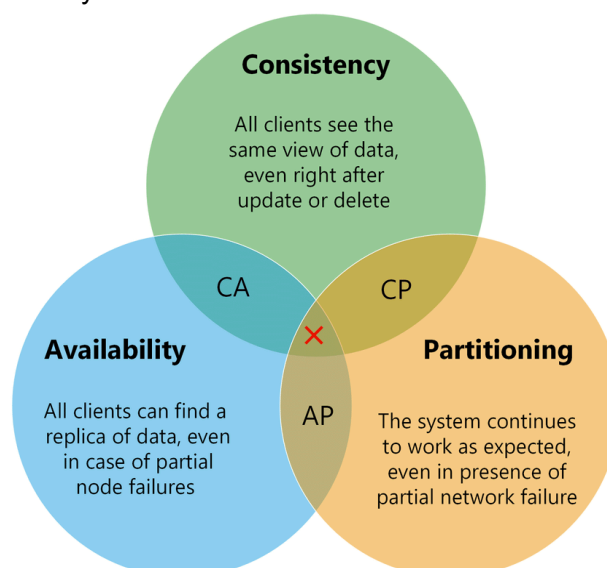
CAP Theorem

The CAP theorem, also known as Brewer's theorem, is a fundamental principle in distributed systems that was formulated by computer scientist Eric Brewer in 2000. The CAP theorem describes the trade-offs and constraints that exist when designing distributed systems, particularly in the context of data consistency, availability, and network partition tolerance. The theorem states that in a distributed system, it is impossible to simultaneously achieve all three of the following guarantees:

Consistency (C): Consistency implies that all nodes in a distributed system see the same data at the same time. In a consistent system, when a write operation is acknowledged, any subsequent read operation will return the most recent write's value.

Availability (A): Availability means that every request (read or write) made to the distributed system receives a response without guaranteeing that it contains the most recent data. An available system remains responsive to user requests even in the presence of network failures or node crashes.

Partition Tolerance (P): Partition tolerance deals with the system's ability to function even when network partitions occur, leading to communication failures between nodes. A partition-tolerant system can continue to operate and make progress despite network splits or delays in message delivery.



When a network partition failure happens should we decide to:

- cancel the operation and thus decrease availability but ensure consistency or
- proceed with the operation and thus provide availability but risk consistency.

The CAP theorem asserts that you can have at most two out of these three properties in a distributed system, but you cannot have all three simultaneously. This leads to the following trade-off scenarios:

CA: A system that prioritizes Consistency and Availability may sacrifice Partition Tolerance. Such systems aim to maintain strong data consistency and high availability but may not function properly in the presence of network partitions.

CP: A system that prioritizes Consistency and Partition Tolerance may sacrifice Availability. These systems ensure data consistency and can withstand network partitions but may become temporarily unavailable during partition events.

AP: A system that prioritizes Availability and Partition Tolerance may sacrifice Consistency. These systems aim to provide continuous availability and handle network partitions but may return potentially stale or conflicting data.

But it's really just two because a distributed system has to be tolerant of partitions but...

Consistency and Availability are not strict binary choices; trade-offs are possible. AP systems prioritize availability while accepting some degree of relaxed consistency, but they are not inherently inconsistent. Conversely, CP systems prioritize strong consistency but may experience temporary unavailability.

This implies that both AP and CP systems can strike a balance, offering a certain level of both consistency and availability, while also maintaining partition tolerance as a fundamental requirement.

PACELC

PACELC is an extension of the CAP theorem, introduced by computer scientist Daniel Abadi in 2010. It further refines the trade-offs in distributed systems by considering two additional factors: Latency and Consistency. The PACELC theorem takes into account the practical realities of distributed systems where network latency and communication costs play a significant role.

Partition Tolerance (P): Similar to the CAP theorem, this indicates whether the system continues to operate despite network partitions. P is always present as it is assumed that network partitions are unavoidable.

Availability (A): the ability of the system to respond to read and write requests at all times, regardless of network conditions.

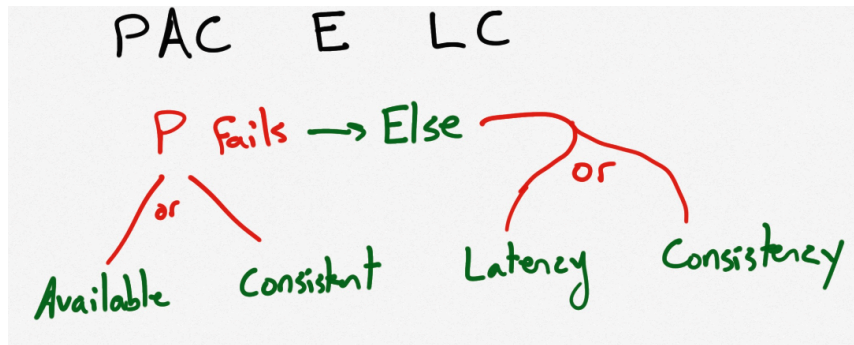
Consistency (C): strong data consistency, where all nodes in the distributed system have a consistent and up-to-date view of the data.

Else (E): even when the system is running normally in the absence of partitions.

Latency (L): L focuses on minimizing communication latency or delays between nodes in the system. Low latency is desirable in many real-time and interactive applications.

Consistency (C).

The PACELC states that if we have partitioning (P) we must choose between availability (A) or consistency (C), else (in normal scenarios) there's a tradeoff between latency (L) and consistency (C).



Types of Consistency

Strong Consistency: Once an update operation is finished, all subsequent accesses to the data will consistently return the same updated value.

Weak Consistency: There is no assurance that subsequent accesses will consistently return the updated value after an update operation.

Eventual Consistency: specific form of weak consistency, allows for temporary data inconsistencies in a distributed system, with the guarantee that if no new updates are made, all replicas of the data will eventually converge to a consistent state.

Eventual consistency can be of different types:

Causal Consistency: processes that share a causal relationship will always observe consistent data.

Read-your-write consistency: guarantees that a process, after performing an update operation on a data item, will never observe an older value when subsequently accessing that data item.

Session consistency: offers read-your-write consistency within a session's context. As long as a session exists, the system guarantees that a process will observe its own writes immediately after performing them. This guarantee applies as long as sessions do not overlap with each other.

Monotonic read consistency: if a process has seen a particular value of data item, any subsequent processes will never return any previous values.

Monotonic write consistency: the system guarantees to serialize the writes by the same process.

Akka

Why actor model?

Event-driven model: actors perform work in response to messages. Communication between Actors is asynchronous, allowing Actors to send messages and continue their own work without blocking waiting for a reply.

Strong isolation principles: unlike regular objects in Java, an Actor does not have a public API in terms of methods that you can invoke. Instead, its public API is defined through messages that the actor handles. This prevents any sharing of state between Actors; the only way to observe another actor's state is by sending it a message asking for it.

Location transparency: the system constructs Actors from a factory and returns references to the instances. Because location doesn't matter, Actor instances can start, stop, move, and restart to scale up and down as well as recover from unexpected failures.

Lightweight: each instance consumes only a few hundred bytes, which realistically allows millions of concurrent Actors to exist in a single application.

Introduction

Akka is a comprehensive set of libraries tailored for the development of scalable and resilient systems that operate concurrently and in a distributed manner. It simplifies the complexities associated with ensuring reliable behavior, fault tolerance, and high-performance in distributed applications. To succeed in building distributed systems, Akka addresses the challenges posed by environments where components can abruptly crash without responding, messages may vanish without leaving a trace on the network, and network latency tends to fluctuate unpredictably. These issues are common occurrences in distributed setups, and Akka provides valuable tools to mitigate their impact and build robust distributed systems.

Akka offers a multi-threaded behavior that eliminates the need for dealing with low-level concurrency constructs such as atomics or locks. This approach frees developers from the complexities of addressing memory visibility issues. Additionally, Akka facilitates transparent remote communication between systems and their components, sparing developers from the challenges of crafting and managing error-prone networking code. Furthermore, Akka boasts a clustered, high-availability architecture that exhibits elasticity, enabling seamless scaling in or out as needed. This elasticity empowers the creation of responsive and adaptable systems that can react dynamically to varying demands.

Actors in Akka

In the Akka framework, each actor is responsible for defining the type of messages it can receive. To ensure message immutability and support pattern matching, case classes and case objects are commonly employed for message representation. This approach is leveraged within the Actor to efficiently match and process the messages it receives.

In Akka, messages serve as the public API for actors, and it is advisable to define messages with clear, meaningful names and with semantic and domain-specific significance, even when they essentially wrap underlying data types. Ensuring that messages are immutable is essential since they are shared among different threads. Organizing an actor's associated messages within its object is a good practice as it enhances clarity regarding the types of messages the actor anticipates and manages. Lastly, an effective practice is to establish an actor's initial behavior within the object's apply method, facilitating a comprehensive understanding of the actor's functionality right from its inception.

In the context of the "HelloWorld" actors, three distinct message types are utilized: "Greet," which serves as a command directed to the Greeter actor to initiate a greeting; "Greeted," a reply message from the Greeter actor confirming the completion of the greeting; and "SayHello," a command signaling the GreeterMain to commence the greeting process.