**Unit 4: Cloud Programming Models (12 Hrs.)**

* Thread programming,
* Task programming,
* Map-reduce programming,
* Parallel efficiency of Map-Reduce,
* Enterprise batch processing using Map-Reduce,
* Comparisons between Thread, Task and Map reduce

**Thread programming,**

* With computer programming, a **thread** is a small set of instructions designed to be scheduled and executed by the CPU independently of the parent process.
* For example, a program may have an open thread waiting for a specific event to occur or running a separate job, allowing the main program to perform other tasks.

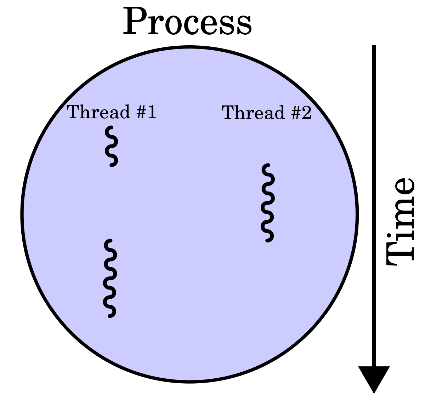
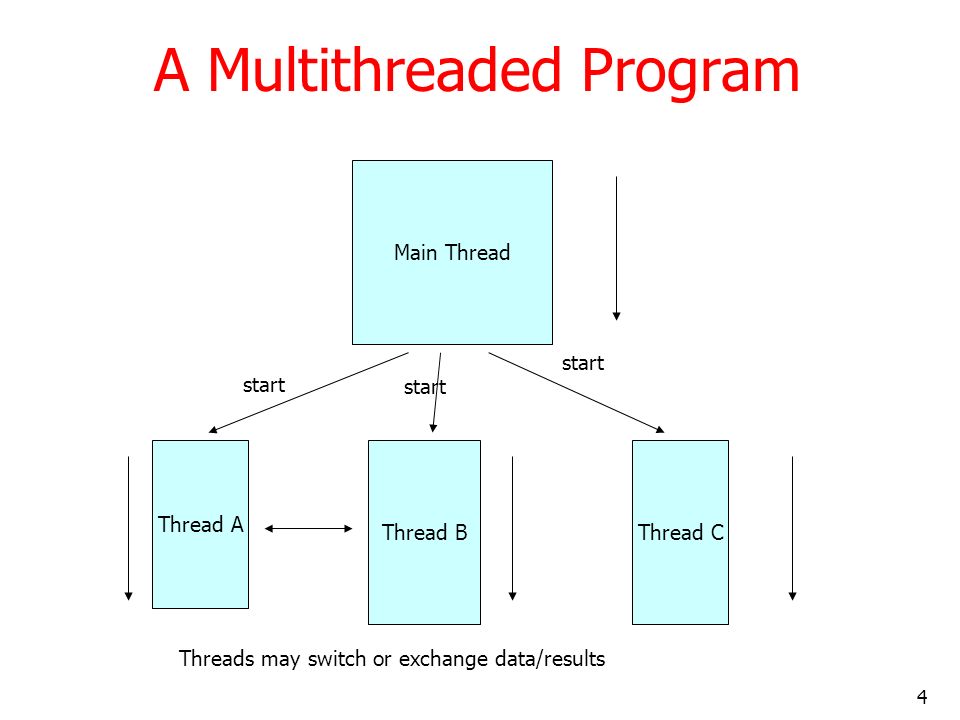
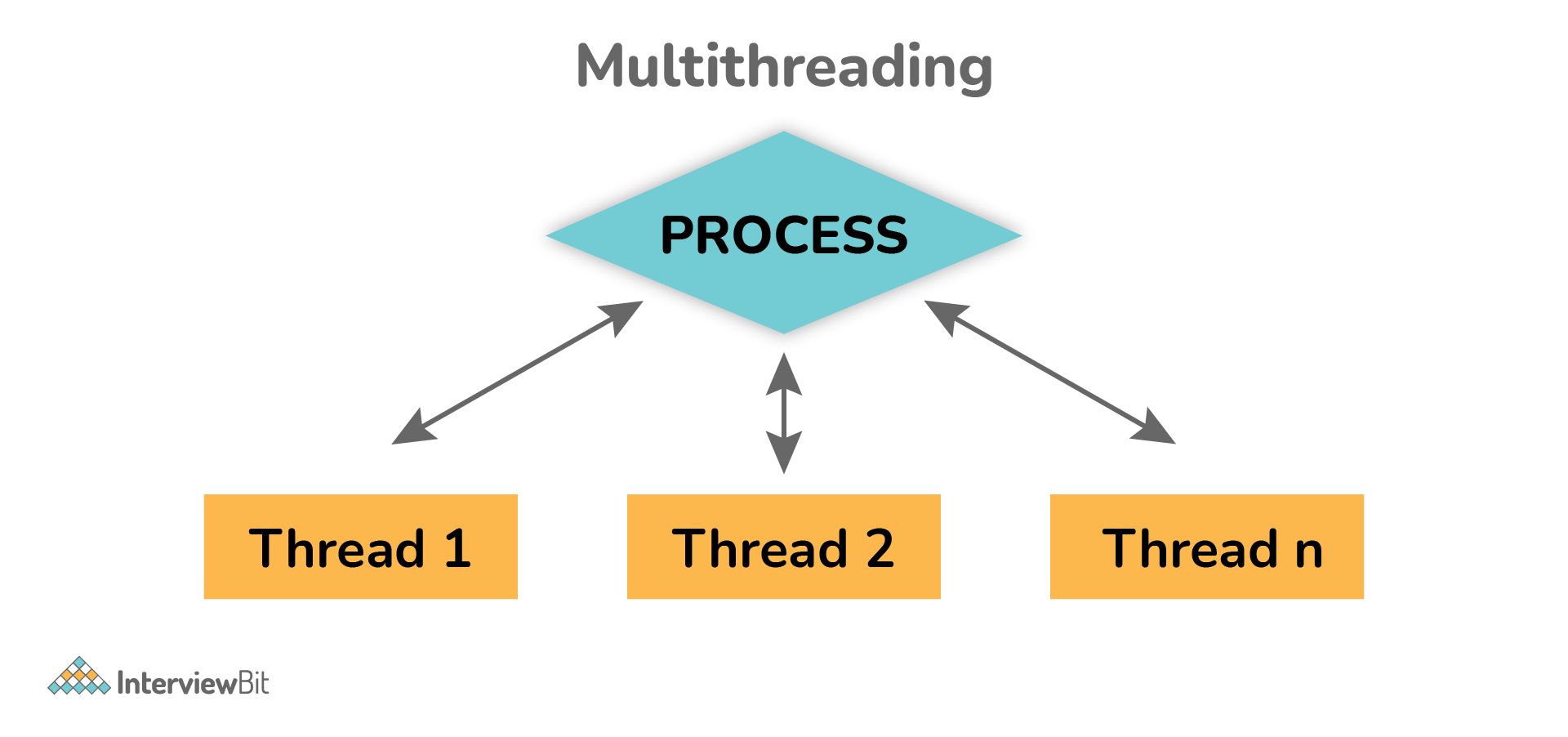


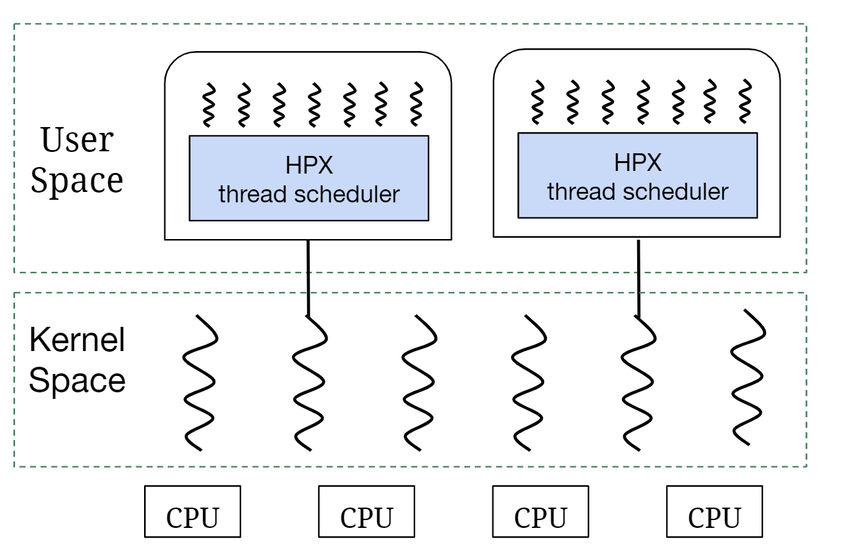
Fig 1. A process with two threads of execution, running on a single processor.

* Multithreading is becoming increasingly
* important, both as a program structuring mechanism and
* to support efficient parallel computations. This paper
* surveys research in analysis for multithreaded programs,
* focusing on ways to improve the efficiency of analyzing
* interactions between threads and to detect data races. We
* try to compare the results of the sequential and threaded
* programming so that we can deploy these models to cloud
* platforms
* Multithreading is becoming increasingly important, both as a program structuring mechanism and to support efficient parallel computations. The research in analysis for multithreaded programs, focusing on ways to improve the efficiency of analysing interactions between threads and to detect data races. We try to compare the results of the sequential and threaded programming so that we can deploy these models to cloud platforms
* Multithreading is a widely used structuring technique for
* modern software. Programmers use multiple threads of
* control for a variety of reasons: to build responsive servers
* that interact with multiple clients, to run computations in
* parallel on a multiprocessor for performance, and as a
* structuring mechanism for implementing rich user
* interfaces. In general, threads are useful whenever the
* software needs to manage a set of tasks with varying
* interaction latencies, exploit multiple physical resources, or
* execute largely independent tasks in response to multiple
* external events. Developing parallel applications requires
* an understanding of the problem and its logical structure.
* Understanding the dependencies and the correlation of
* tasks within an application is fundamental for designing the
* right program structure and to introduce parallelism where
* appropriate. Decomposition is a useful technique that helps
* to understand whether a problem is divided into
* components (or tasks) that can be executed concurrently.
* The two main decomposition/partitioning techniques used
* area: domain and functional decomposition
* Multithreading is a widely used structuring technique for modern software. Programmers use multiple threads of control for a variety of reasons: to build responsive servers that interact with multiple clients, to run computations in parallel on a multiprocessor for performance, and as a structuring mechanism for implementing rich user interfaces.
* In general, threads are useful whenever the software needs to manage a set of tasks with varying interaction latencies, exploit multiple physical resources, or execute largely independent tasks in response to multiple external events.
* Developing parallel applications requires an understanding of the problem and its logical structure. Understanding the dependencies and the correlation of tasks within an application is fundamental for designing the right program structure and to introduce parallelism where appropriate. Decomposition is a useful technique that helps to understand whether a problem is divided into components (or tasks) that can be executed concurrently.

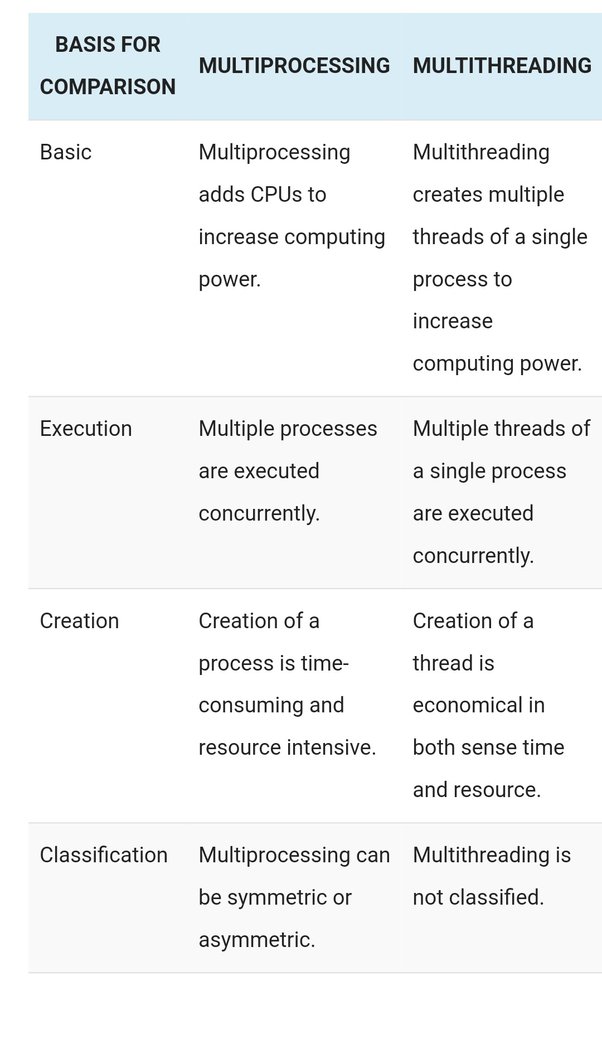








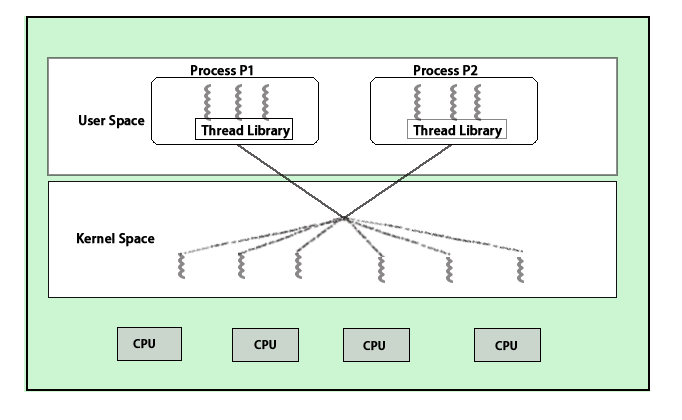
* **User-level thread**:
  + They are created and managed by users. They are used at the application level.
  + The [operating system](https://www.javatpoint.com/operating-system) does not recognize the **user-level thread**. User threads can be easily implemented and it is implemented by the user. If a user performs a user-level thread blocking operation, the whole process is blocked. The kernel level thread does not know nothing about the user level thread.
* **kernel thread**
  + The **kernel thread** recognizes the operating system.
  + There is a thread control block and process control block in the system for each thread and process in the kernel-level thread.
  + The kernel-level thread is implemented by the operating system. T
  + he kernel knows about all the threads and manages them.
  + The kernel-level thread offers a system call to create and manage the threads from user-space.
  + The implementation of kernel threads is more difficult than the user thread. Context switch time is longer in the kernel thread.



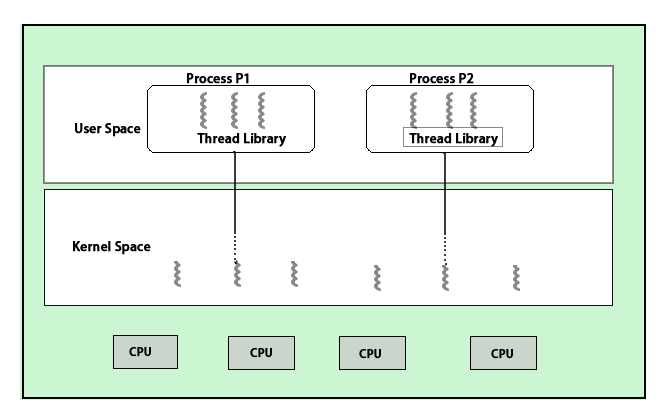
### **Multithreading Models**

These models are of three types

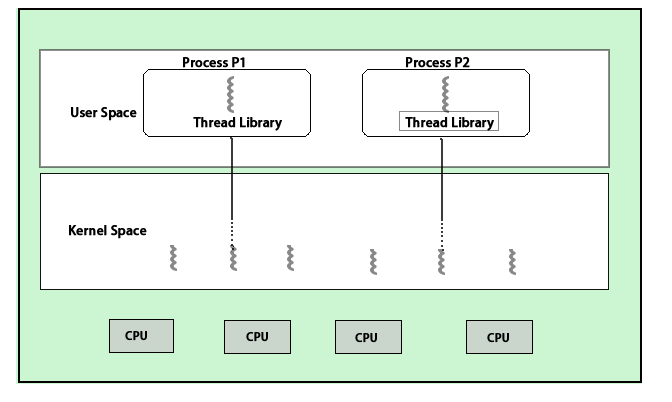
* Many to many
* Many to one
* One to one
* **Many to many**: Any number of user threads can interact with an equal or lesser number of kernel threads.



* **Many to one**: It maps many user-level threads to one Kernel-level thread.



* **One to one**: Relationship between the user-level thread and the kernel-level thread is one to one.



### **Uses of Multithreading**

* It is a way to introduce parallelism in the system or program. So, you can use it anywhere you see parallel paths (where two threads are not dependent on the result of one another) to make it fast and easy.
* For example:
  + Processing of large data where it can be divided into parts and get it done using multiple threads.
  + Applications which involve mechanism like validate and save, produce and consume, read and validate are done in multiple threads. Few examples of such applications are online banking, recharges, etc.
  + It can be used to make games where different elements are running on different threads.
  + In Android, it is used to hit the APIs which are running in the background thread to save the application from stopping.
  + In web applications, it is used when you want your app to get asynchronous calls and perform asynchronously.

### **Advantages of Multithreading**

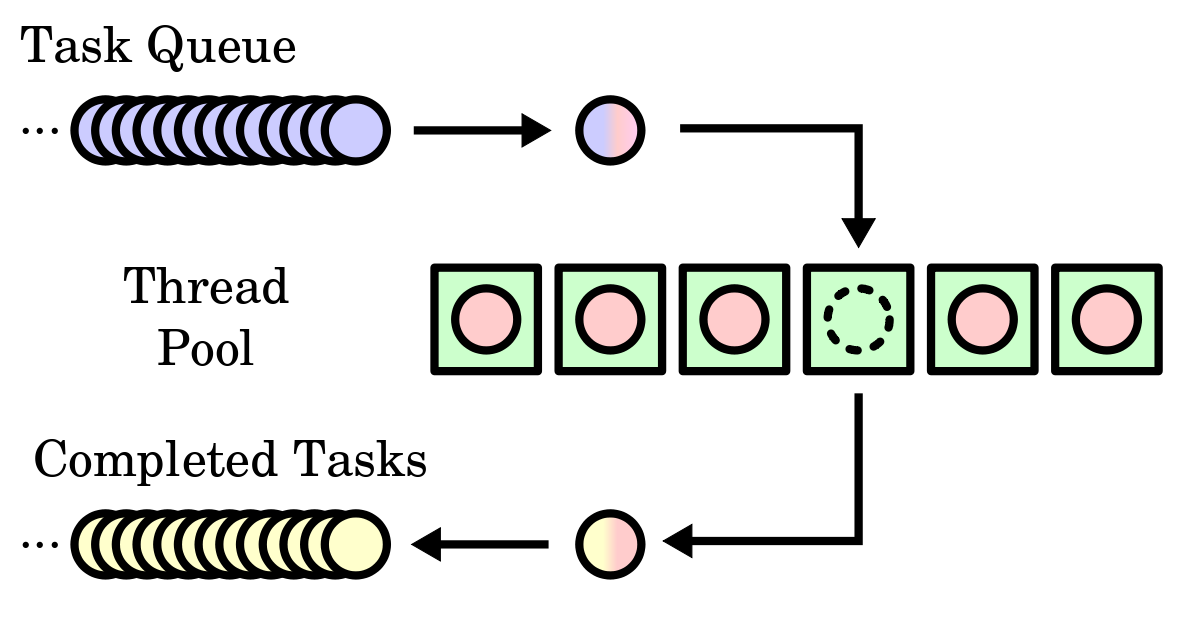
* Below are mentioned some of the advantages:
  + **Economical:** It is economical as they share the same processor resources. It takes lesser time to create threads.
  + **Resource sharing**: It allows the threads to share resources like data, memory, files, etc. Therefore, an application can have multiple threads within the same address space.
  + **Responsiveness**: It increases the responsiveness to the user as it allows the program to continue running even if a part of it is performing a lengthy operation or is blocked.
  + **Scalability**: It increases parallelism on multiple CPU machines. It enhances the performance of multi-processor machines.
  + It makes the usage of CPU resources better.

### **Why should we use Multithreading?**

* We should use this because of the following reasons:
  + To increase parallelism
  + To make most of the available CPU resources.
  + To improve application responsiveness and give better interaction with the user.

**Task Programming**

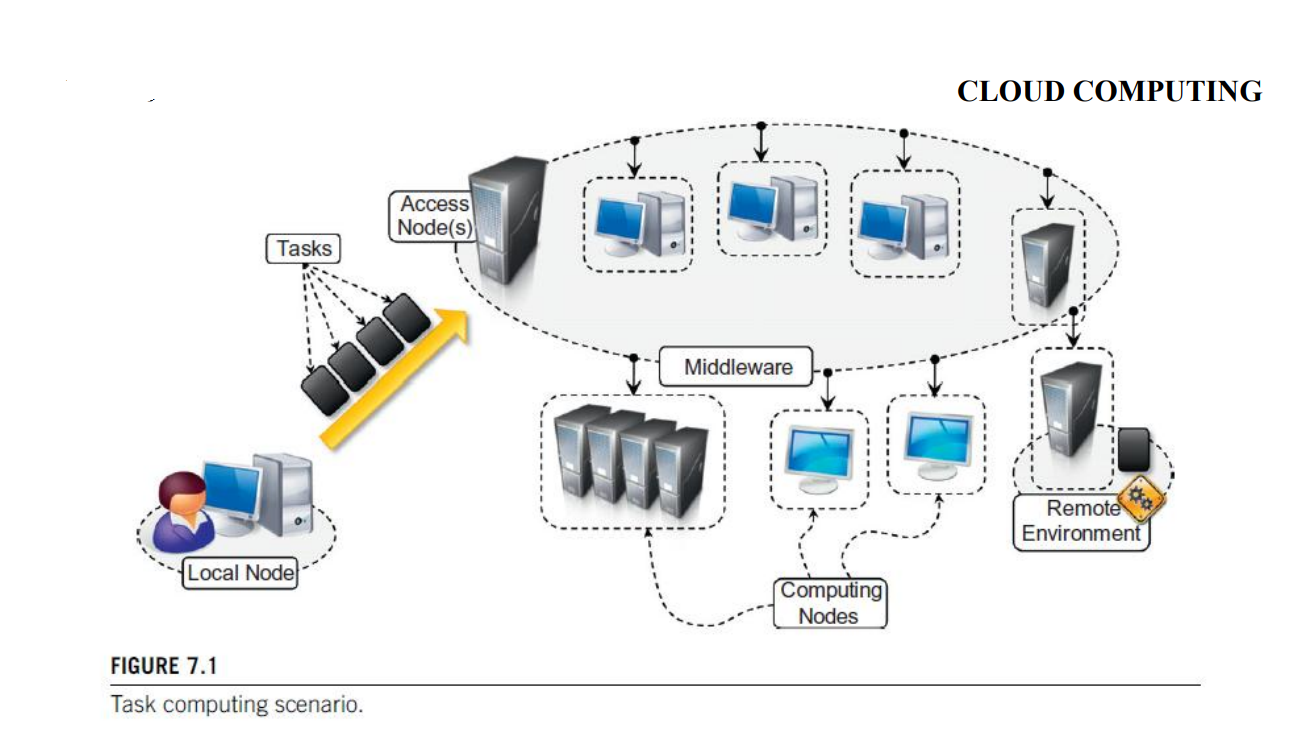
* Task computing is a wide area of distributed system programming encompassing several different models of architecting distributed applications,
* A task represents a program, which require input files and produce output files as a result of its execution. Applications are then constituted of a collection of tasks. These are submitted for execution and their output data are collected at the end of their execution. This chapter characterizes the abstraction of a task and provides a brief overview of the distributed application models that are based on the task abstraction.
* The **Aneka Task Programming Model** is taken as a reference implementation to illustrate the execution of **bag-of-tasks (BoT)** applications on a distributed infrastructure.
* In computer programming, a task is a basic unit of programming that an operating system controls. All of today's widely-used operating systems support multitasking, which allows multiple tasks to run concurrently, taking turns using the resources of the computer. Depending on how the operating system defines a task in its design, this unit of programming may be an entire program or each successive invocation of a program.
* In computing, a task is a unit of execution or a unit of work. The term is ambiguous; precise alternative terms include process, light-weight process, thread (for execution), step, request, or query (for work).



* The Task Programming Model is a **high-level multithreaded programming model**. It is designed to allow Maple code to be written that takes advantage of multiple processors, while avoiding much of the complexity of traditional multithreaded programming. No explicit threading, users create Tasks, not Threads.
* Task computing is a wide area of **distributed system programming** encompassing several different models of architecting distributed applications, which, eventually, are based on the same fundamental abstraction: the task. Applications are then constituted of a collection of tasks.

**Task computing**

* A task identifies one or more operations that produce a distinct output and that can be isolated as a single logical unit. In practice, a task is represented as a distinct unit of code, or a program, that can be separated and executed in a remote run time environment.
* Multithreaded programming is mainly concerned with providing a support for parallelism within a single machine. Task computing provides distribution by harnessing the compute power of several computing nodes. Hence, the presence of a distributed infrastructure is explicit in this model. Now clouds have emerged as an attractive solution to obtain a huge computing power on demand for the execution of distributed applications.
* To achieve it, suitable middleware is needed. A reference scenario for task computing is depicted in Figure 7.1.



* The middleware is a software layer that enables the coordinated use of multiple resources, which are drawn from a datacentre or geographically distributed networked computers.
* A user submits the collection of tasks to the access point(s) of the middleware, which will take care of scheduling and monitoring the execution of tasks.
* Each computing resource provides an appropriate runtime environment.
* Task submission is done using the APIs provided by the middleware, whether a Web or programming language interface. Appropriate APIs are also provided to monitor task status and collect their results upon completion. It is possible to identify a set of common operations that the middleware needs to support the creation and execution of task-based applications.
* These operations are:
  + Coordinating and scheduling tasks for execution on a set of remote nodes
  + Moving programs to remote nodes and managing their dependencies
  + Creating an environment for execution of tasks on the remote nodes
  + Monitoring each task’s execution and informing the user about its status
  + Access to the output produced by the task.

**Frameworks for task computing**

* Some popular software systems that support the task-computing framework are:

1. Condor 2. Globus Toolkit 3. Sun Grid Engine (SGE) 4. BOINC

* Architecture of all these systems is similar to the general reference architecture depicted in Figure 7.1.
* They consist of two main components: a scheduling node (one or more) and worker nodes.
* The organization of the system components may vary.

**1. Condor**

* Condor is the most widely used and long-lived middleware for managing clusters, idle workstations, and a collection of clusters.
* Condor supports features of batch-queuing systems along with the capability to checkpoint jobs and manage overload nodes.
* It provides a powerful job resource-matching mechanism, which schedules jobs only on resources that have the appropriate runtime environment.
* Condor can handle both serial and parallel jobs on a wide variety of resources.
* It is used by hundreds of organizations in industry, government, and academia to manage infrastructures

**2. Globus Toolkit**

* The Globus Toolkit is a collection of technologies that enable grid computing.
* It provides a comprehensive set of tools for sharing computing power, databases, and other services across corporate, institutional, and geographic boundaries.
* The toolkit features software services, libraries, and tools for resource monitoring, discovery, and management as well as security and file management.
* The toolkit defines a collection of interfaces and protocol for interoperation that enable different systems to integrate with each other and expose resources outside their boundaries.

**3. Sun Grid Engine (SGE)**

* Sun Grid Engine (SGE), now ***Oracle Grid Engine***, is middleware for workload and distributed resource management.
* Initially developed to support the execution of jobs on clusters, SGE integrated additional capabilities and now is able to manage heterogeneous resources and constitutes middleware for grid computing.
* It supports the execution of parallel, serial, interactive, and parametric jobs and features advanced scheduling capabilities such as budget-based and group- based scheduling, scheduling applications that have deadlines, custom policies, and advance reservation.

**4. BOINC**

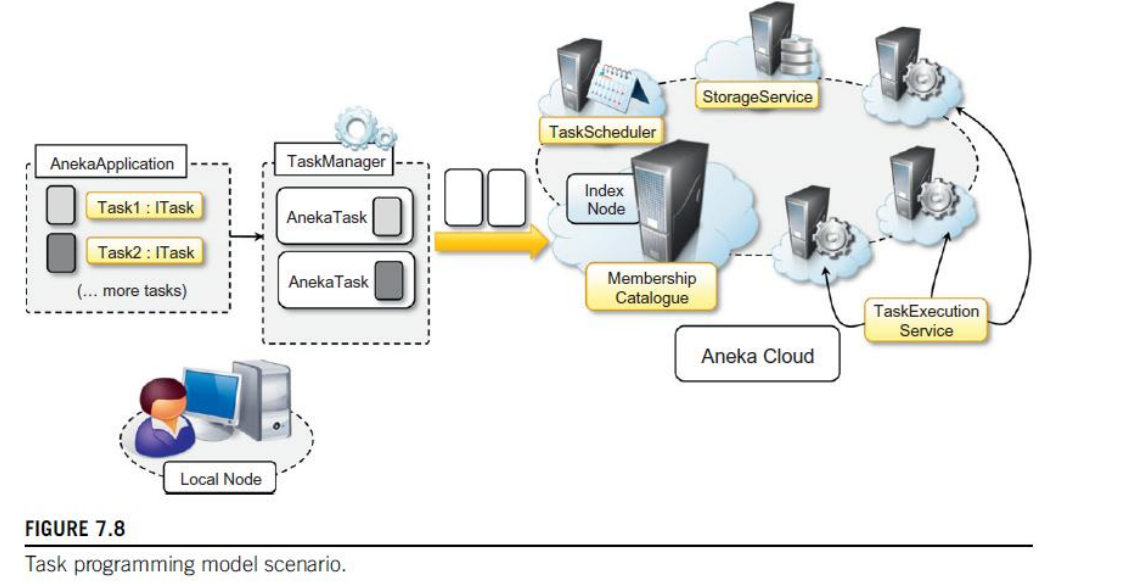
* **Berkeley Open Infrastructure for Network Computing (BOINC)** is framework for ***volunteer and grid computing.***
* It allows us to turn desktop machines into volunteer computing nodes that are leveraged to run jobs when such machines become inactive. BOINC supports job check pointing and duplication.
* BOINC is composed of two main components: the BOINC server and the BOINC client. The **BOINC server** is the central node that keeps track of all the available resources and scheduling jobs. The **BOINC client** is the software component that is deployed on desktop machines and that creates the BOINC execution environment for job submission. BOINC systems can be easily set up to provide more stable support for job execution by creating computing grids with dedicated machines.
* When installing BOINC clients, users can decide the application project to which they want to donate the CPU cycles of their computer. Currently several projects, ranging from medicine to astronomy and cryptography, are running on the BOINC infrastructure.

**Aneka task-based programming**

* Aneka is a software platform for developing cloud computing applications.
  + Aneka is a pure PaaS solution for cloud computing.
  + Aneka is a cloud middleware product that can be deployed on a heterogeneous set of resources: Like: a network of computers, a multi core server, data centres, virtual cloud infrastructures, or a mixture of all
  + The framework provides both middleware for managing and scaling distributed applications and an extensible set of APIs for developing them
* Aneka provides support for all the flavours of task-based programming by means of the Task Programming Model, which constitutes the basic support given by the framework for supporting the execution of **bag-of-tasks** applications.
* Task programming is realized through the abstraction of the Aneka.Tasks.ITask. By using this abstraction as a basis support for execution of legacy applications, parameter sweep applications and workflows have been integrated into the framework.

**Task programming model**

* The Task Programming Model provides a very intuitive abstraction for quickly developing distributed applications on top of Aneka.
* It provides a minimum set of APIs that are mostly centered on the Aneka.Tasks.ITask interface.
* Figure 7.8 provides an overall view of the components of the Task Programming Model and their roles during application execution.



* Developers create distributed applications in terms of **ITask instances**, the collective execution of which describes a running application. These tasks, together with all the required dependencies (data files and libraries), are grouped and managed through the Aneka Application class, which is specialized to support the execution of tasks.
* Two other components, **AnekaTask and Task Manager**, constitute the client-side view of a task-based application. The former constitutes the runtime wrapper Aneka uses to represent a task within the middleware; the latter is the underlying component that interacts with Aneka, submits the tasks, monitors their execution, and collects the results.
* In the middleware, four services coordinate their activities in order to execute task-based applications. These are: **MembershipCatalogue, TaskScheduler, ExecutionService, and StorageService**. **MembershipCatalogue** constitutes the main access point of the cloud and acts as a service directory to locate the **TaskScheduler** service that is incharge of managing the execution of task-based applications.
* Its main responsibility is to allocate task instances to resources featuring the Execution Service for task execution and for monitoring task state.

|  |  |
| --- | --- |
|  | **Advantages of using the Task Programming Model**  Here are a few advantages of using the Task Programming Model: |

|  |  |
| --- | --- |
| • | No explicit threading, users create Tasks, not Threads. |

|  |  |
| --- | --- |
| • | Maple schedules the Tasks to the processors so that the code scales to the number of available processors. |

|  |  |
| --- | --- |
| • | Multiple algorithms written using the Task Programming Model can run at the same time without significant performance impact. |

|  |  |
| --- | --- |
| • | Complex problems can be solved without requiring traditional synchronization tools such as Mutexes and Condition Variables. |

|  |  |
| --- | --- |
| • | If such synchronization tools are not used, the function cannot be deadlocked. |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| • | The [Task](https://www.maplesoft.com/support/help/maple/view.aspx?path=Threads%2fTask) functions are simple and the model mirrors conventional function calling.  **MapReduce programming,** MapReduce  * MapReduce is a processing technique and a program model for distributed computing based on high level programing language. * MapReduce is a programming model or pattern within the Hadoop framework that is used to access big data stored in the **Hadoop Distributed File System (HDFS).** It is a core component, integral to the functioning of the Hadoop framework. * Fig 1 below shows the basic block diagram of MapReduce.   MapReduce Algorithm   * The MapReduce algorithm contains two important tasks, namely **Map and Reduce**. **Map** takes *a set of data and converts it into another set of data*, where individual elements are broken down into tuples (key/value pairs). Secondly, **reduce** task, *which takes the output from a map as an input and combines those data tuples into a smaller set of tuples*. As the sequence of the name MapReduce implies, the reduce task is always performed after the map job. * The MapReduce framework operates on <key, value> pairs, that is, the framework views the input to the job as a set of <key, value> pairs and produces a set of <key, value> pairs as the output of the job, conceivably of different types. * The key and the value classes should be in serialized manner by the framework and hence, need to implement the Writable interface. Additionally, the key classes have to implement the Writable-Comparable interface to facilitate sorting by the framework. Input and Output types of a **MapReduce job** − (Input) <k1, v1> → map → <k2, v2> → reduce → <k3, v3>(Output).  |  |  |  | | --- | --- | --- | |  | **Input** | **Output** | | **Map** | <k1, v1> | list (<k2, v2>) | | **Reduce** | <k2, list(v2)> | list (<k3, v3>) |      * The major advantage of MapReduce is that it is easy to scale data processing over multiple computing nodes. Under the MapReduce model, the data processing primitives are called mappers and reducers. Decomposing a data processing application into *mappers* and *reducers* is sometimes nontrivial. * But, once we write an application in the MapReduce form, scaling the application to run over hundreds, thousands, or even tens of thousands of machines in a cluster is merely a configuration change. This simple scalability is what has attracted many programmers to use the MapReduce model. * The **MapReduce task** is *mainly divided into* ***two phases*** [**Map Phase**](https://www.geeksforgeeks.org/hadoop-mapper-in-mapreduce/) and [**Reduce Phase**](https://www.geeksforgeeks.org/hadoop-reducer-in-map-reduce/)as shown above.   **Components of MapReduce Architecture:**   * **Client:** * The MapReduce client is the one who brings the Job to the MapReduce for processing. * There can be multiple clients available that continuously send jobs for processing to the Hadoop MapReduce Manager. * **Job:** * The MapReduce Job is the actual work that the client wanted to do which is comprised of so many smaller tasks that the client wants to process or execute. * **Hadoop MapReduce Master:** * It divides the particular job into subsequent job-parts. * **Job-Parts:** * The task or sub-jobs that are obtained after dividing the main job. * The result of all the job-parts combined to produce the final output. * **Input Data:** * The data set that is fed to the MapReduce for processing. * **Output Data:** * The final result is obtained after the processing.     Fig Components of MapReduce Architecture   * In **MapReduce**, we have a client. The client will submit the job of a particular size to the Hadoop MapReduce Master. * Now, the MapReduce master will divide this job into further equivalent job-parts. These job-parts are then made available for the Map and Reduce Task. This Map and Reduce task will contain the program as per the requirement of the use-case that the particular company is solving. * The developer writes their logic to fulfil the requirement that the industry requires. * The input data which we are using is then fed to the Map Task and the Map will generate intermediate key-value pair as its output. The output of Map i.e. these key-value pairs are then fed to the Reducer and the final output is stored on the HDFS. * There can be n number of Map and Reduce tasks made available for processing the data as per the requirement. * The algorithm for Map and Reduce is made with a very optimized way such that the time complexity or space complexity is minimum. * Let’s discuss the MapReduce phases to get a better understanding of its architecture: * The MapReduce task is mainly divided into **2 phases** i.e. Map phase and Reduce phase. * **Map:** * As the name suggests its main use is to map the input data in key-value pairs. * The input to the map may be a key-value pair where the key can be the id of some kind of address and value is the actual value that it keeps. * The *Map()* function will be executed in its memory repository on each of these input key-value pairs and generates the intermediate key-value pair which works as input for the Reducer or *Reduce()* function.      * **Reduce:** * The intermediate key-value pairs that work as input for Reducer are shuffled and sort and send to the Reduce() function. * Reducer aggregate or group the data based on its key-value pair as per the reducer algorithm written by the developer. * How Job tracker and the task tracker deal with MapReduce: * **Job Tracker:** * The work of Job tracker is to manage all the resources and all the jobs across the cluster and also to schedule each map on the Task Tracker running on the same data node since there can be hundreds of data nodes available in the cluster. * **Task Tracker:** * The Task Tracker can be considered as the actual slaves that are working on the instruction given by the Job Tracker. * This Task Tracker is deployed on each of the nodes available in the cluster that executes the Map and Reduce task as instructed by Job Tracker.   **Inputs and Outputs**   * The Map function takes input from the disk as <key,value> pairs, processes them, and produces another set of intermediate <key,value> pairs as output. * The Reduce function also takes inputs as <key,value> pairs, and produces <key,value> pairs as output. |

* The computation model expressed by MapReduce is very straightforward and allows greater productivity for people who have to code the algorithms for processing huge quantities of data.
* In general, any computation that can be expressed in the form of two major stages can be represented in terms of MapReduce computation.
* These stages are:

**1. Analysis.**

* This phase operates directly on the data input file and corresponds to the operation performed by the map task. Moreover, the computation at this stage is expected to be embarrassingly parallel, since map tasks are executed without any sequencing or ordering.

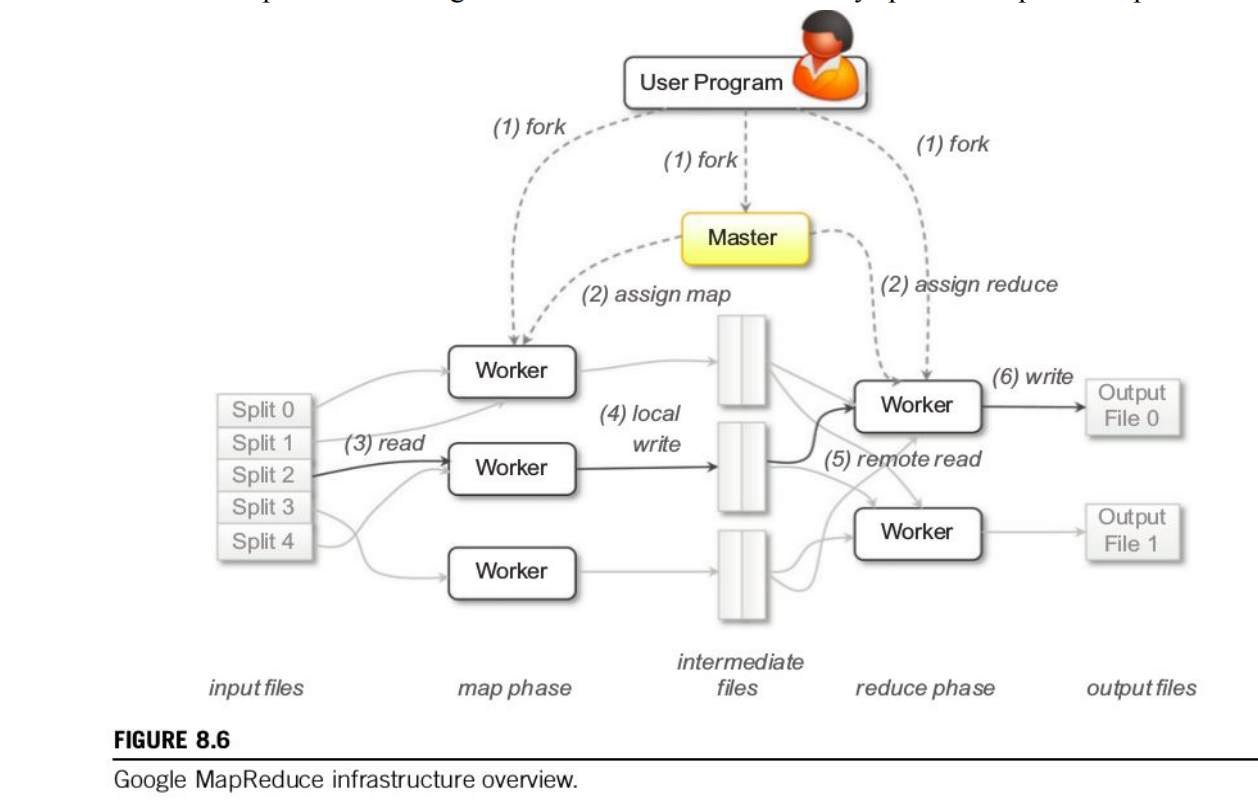
**2. Aggregation.**

* This phase operates on the intermediate results and is characterized by operations that are aimed at aggregating, summing, and/or elaborating the data obtained at the previous stage to present the data in their final form. This is the task performed by the reduce function.
* Figure 8.6 gives a more complete overview of a MapReduce infrastructure, according to the implementation proposed by Google.
* As depicted, the user submits the execution of MapReduce jobs by using the client libraries that are in charge of submitting the input data files, registering the map and reduce functions, and returning control to the user once the job is completed. A generic distributed infrastructure (i.e., a cluster) equipped with job-scheduling capabilities and distributed storage can be used to run MapReduce applications.
* Two different kinds of processes are run on the distributed infrastructure:

**a. Master​ ​process** and

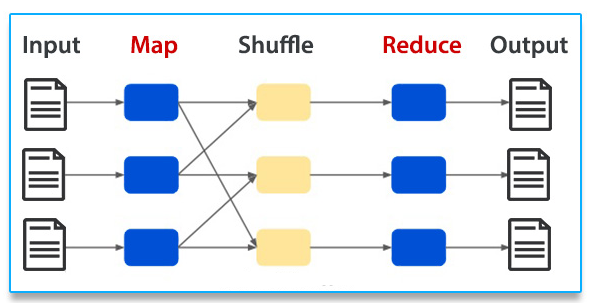
**b.​ ​worker​ ​process.**

* The **master process** is in charge of controlling the execution of map and reduce tasks, partitioning, and reorganizing the intermediate output produced by the map task in order to feed the reduce tasks. The master process generates the map tasks and assigns input splits to each of them by balancing the load.
* The **worker processes** are used to host the execution of map and reduce tasks and provide basic I/O facilities that are used to interface the map and reduce tasks with input and output files. Worker processes have input and output buffers that are used to optimize the performance of map and reduce tasks. In particular, output buffers for map tasks are periodically dumped to disk to create intermediate files. Intermediate files are partitioned using a user-defined function to evenly split the output of map tasks.



**Parallel efficiency of Map-Reduce**

* **MapReduce is an attractive model for parallel data processing in high- performance cluster computing environments**.
* The scalability of MapReduce is proven to be high, because a job in the MapReduce model is partitioned into numerous small tasks running on multiple machines in a large-scale cluster.
* MapReduce allows for the distributed processing of the map and reduction operations.
* **Maps can be performed in parallel, provided that each mapping operation is independent of the others**; in practice, this is limited by the number of independent data sources and/or the number of CPUs near each source.
* Parallel computing is **a type of computing architecture in which several processors simultaneously execute multiple, smaller calculations broken down from an overall larger, complex problem**
* The MapReduce framework performs two steps for each job requested: The first step is to divide the requested job into several mapping tasks and assign them to different computing nodes. The original input processing data of the mapping task is the input file.



* The Map invocations are distributed across multiple machines by **automatically partitioning the input data into a set of M splits or shards**. The input shards can be processed in parallel on different machines.

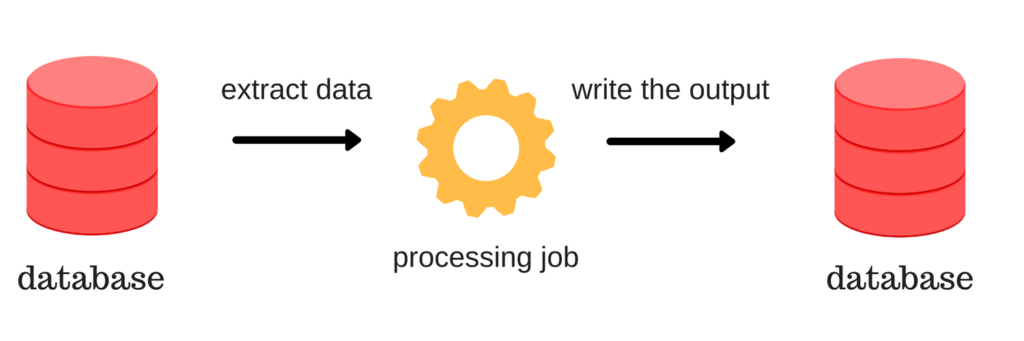
**Enterprise batch processing using Map-Reduce**

**Batch processing – MapReduce**

* MapReduce is a framework that allows the user to write code that is executed on multiple nodes without having to worry about fault tolerance, reliability, synchronization or availability.
* Examples of batch processing are **transactions of credit cards, generation of bills, processing of input and output in the operating system** etc. Examples of real-time processing are bank ATM transactions, customer services, radar system, weather forecasts, temperature measurement etc.

**Batch processing**

* There are a lot of use cases for a system described in the introduction, but the focus of this post will be on data processing – more specifically, batch processing.
* Batch processing is an automated job that does some computation, usually done as a periodical job. It runs the processing code on a set of inputs, called a batch. Usually, the job will read the batch data from a database and store the result in the same or different database.
* An example of a batch processing job could be reading all the sale logs from an online shop for a single day and aggregating it into statistics for that day (number of users per country, the average spent amount, etc.). Doing this as a daily job could give insights into customer trends.

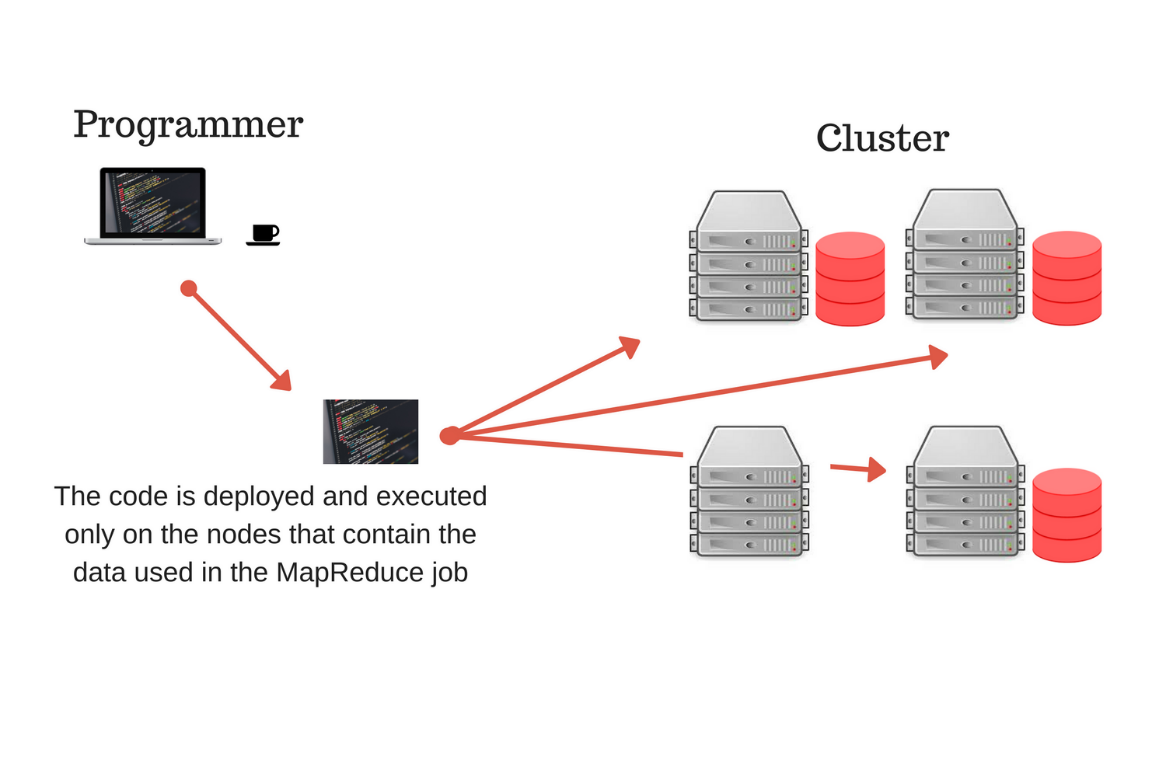


### **MapReduce**

* MapReduce is a programming model that was introduced in a [white paper](https://static.googleusercontent.com/media/research.google.com/en/archive/mapreduce-osdi04.pdf) by Google in 2004. Today, it is implemented in various data processing and storing systems ([Hadoop](https://hadoop.apache.org/), [Spark](https://spark.apache.org/), [MongoDB](https://www.mongodb.com/) etc.) and it is a foundational building block of most big data batch processing systems.
* For MapReduce to be able to do computation on large amounts of data, it has to be a distributed model that executes its code on multiple nodes. This allows the computation to handle larger amounts of data by adding more machines – **horizontal scaling**. This is different from **vertical scaling**, which implies increasing the performance of a single machine.

### **Execution**

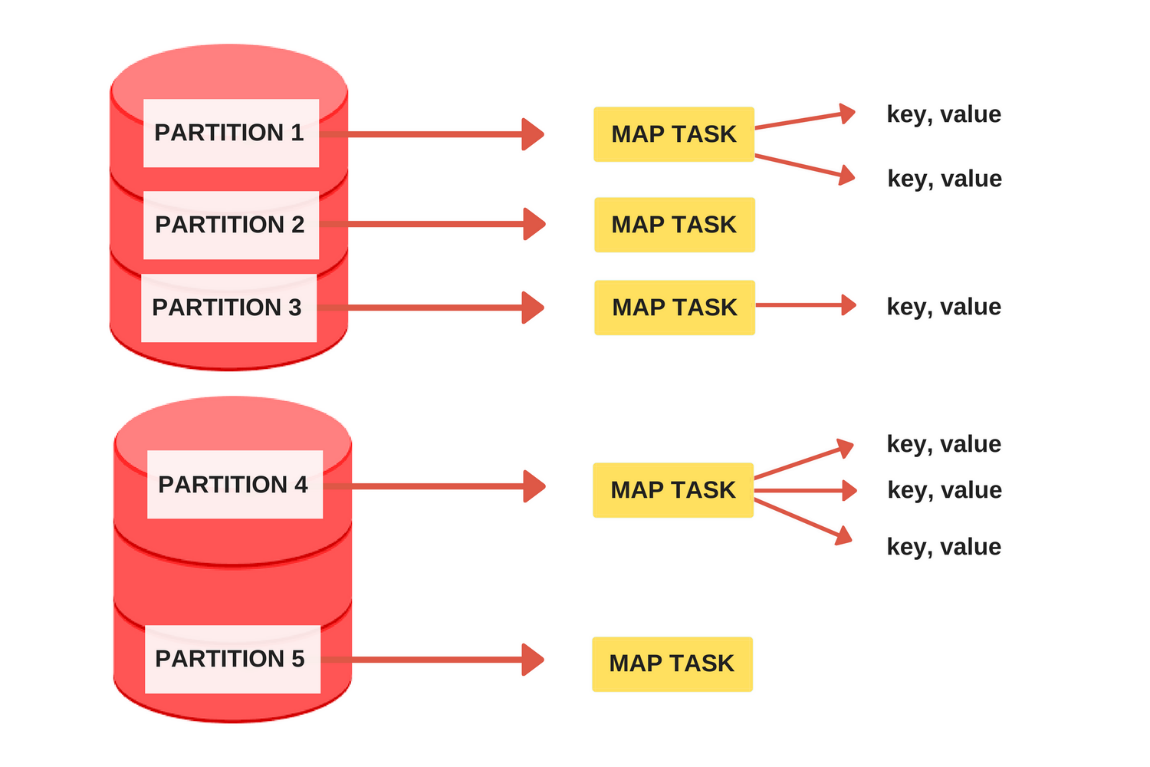
* In order to decrease the duration of our distributed computation, MapReduce tries to reduce **shuffling** (moving) the data from one node to another by distributing the computation so that it is done on the same node where the data is stored. This way, the data stays on the same node, but the code is moved via the network. This is ideal because the code is much smaller than the data.
* To run a MapReduce job, the user has to implement two functions, **map** and **reduce**, and those implemented functions are distributed to nodes that contain the data by the MapReduce framework. Each node runs (executes) the given functions on the data it has in order the minimize network traffic (shuffling data).



* The computation performance of MapReduce comes at the cost of its expressivity. When writing a MapReduce job we have to follow the strict interface (return and input data structure) of the map and the reduce functions. The map phase generates key-value data pairs from the input data (partitions), which are then grouped by key and used in the reduce phase by the reduce task. Everything except the interface of the functions is programmable by the user.

### **Map**

* [Hadoop](https://hadoop.apache.org/), along with its many other features, had the first open-source implementation of MapReduce. It also has its own distributed file storage called [HDFS](https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html). In Hadoop, the typical input into a MapReduce job is a directory in HDFS.
* In order to increase parallelization, each directory is made up of smaller units called partitions and each partition can be processed separately by a map task (the process that executes the map function). This is hidden from the user, but it is important to be aware of it because the number of partitions can affect the speed of execution.



* The map task (mapper) is called once for every input partition and its job is to extract key-value pairs from the input partition. The mapper can generate any number of key-value pairs from a single input (including zero, see the figure above). The user only needs to define the code inside the mapper. Below, we see an example of a simple mapper that takes the input partition and outputs each word as a key with value 1.

# Map function, is applied on a partition

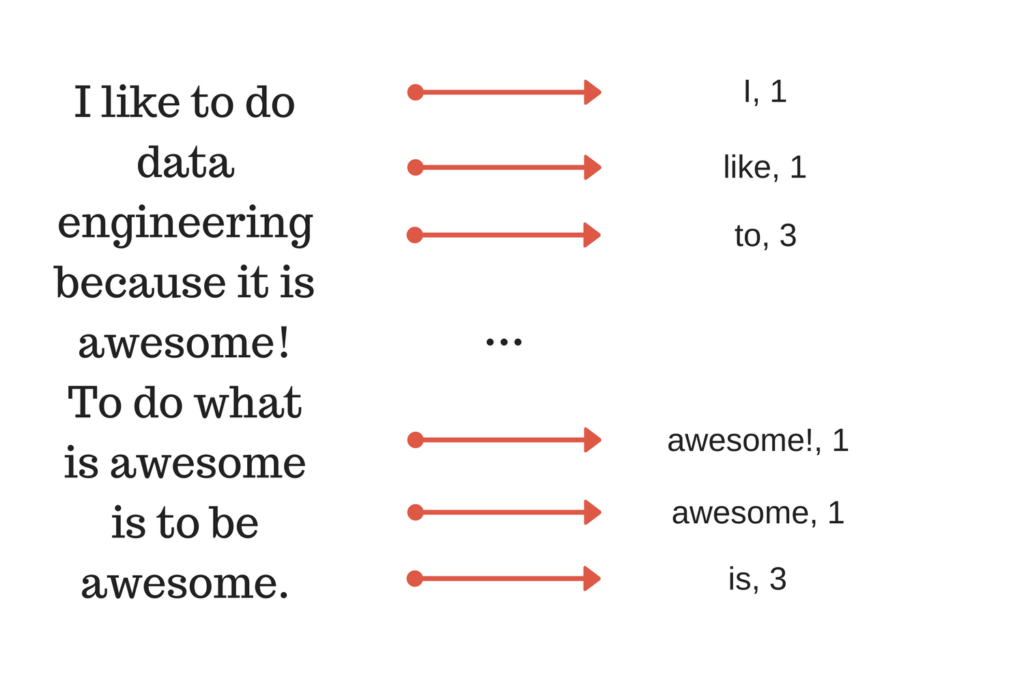
def mapper(key, value):

# Split the text into words and yield word,1 as a pair

for word in value.split():

normalized\_word = world.lower()

yield normalized\_word, 1



### **Reduce**

* The MapReduce framework collects all the key-value pairs produced by the mappers, arranges them into groups with the same key and applies the reduce function. All the grouped values entering the reducers are sorted by the framework. The reducer can produce output files which can serve as input into another MapReduce job, thus enabling multiple MapReduce jobs to chain into a more complex data processing pipeline.

# Reduce function, applied to a group of values with the same key

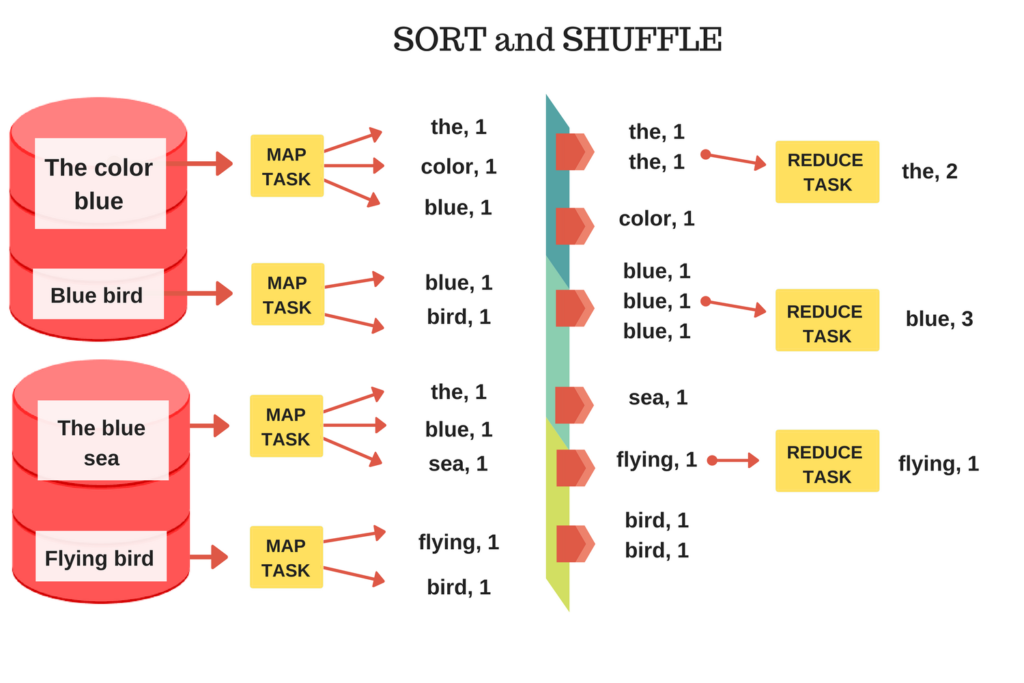
def reducer(key, values):

# Sum all the values with the same key

result = sum(values)

return result

* The mapper yielded key-value pairs with the word as the key and the number 1 as the value. The reducer can be called on all the values with the same key (word), to create a distributed word counting pipeline. In the image below, we see that not every sorted group has a reduce task. This happens because the user needs to define the number of reducers, which is 3 in our case. After a reducer is done with its task, it takes another group if there is one that was not processed.



**Comparisons between Thread, Task and Map reduce**

**Thread**

* Thread is light weight, taking lesser resources than a process
* While one thread is blocked and waiting, a second thread in the same task can run.
* All threads can share same set of open files, child processes.
* Thread switching does not need to interact with operating system.
* The benefits of multi-threaded programming can be broken down into four major categories:
* Responsiveness
* Resource Sharing
* Economy
* Scalability

**Advantages of Thread**

* Threads minimize the context switching time.
* Use of threads provides concurrency within a process.
* Efficient communication.
* It is more economical to create and context switch threads.
* Threads allow utilization of multiprocessor architectures to a greater scale and efficiency.

**Task programming**

* Task programming is **the process by which domain experts create decoder tasks for trajectory self-supervised learning**. This process consists of selecting attributes from trajectory data, writing programs, and creating decoder tasks based on the programs
* Task computing is a **computation** meant to fill the gap between tasks (what the user wants to be done) and services (functionalities that are available to the user).
* Task computing seeks to redefine how users interact with and use computing environments.
* It is built on pervasive computing.

**Advantages of Task programming**

* Timesharing:
* Handle multiple users:
* Protected memory:
* Programs can run in the background:
* Increase reliability:
* The user can use multiple programs:
* Best use of computer resources:

**Map-reduce**

* Map-reduce concept helps the developer from the overhead of handling thread synchronisation, deadlock, shared data.
* Threading offers facilities to partition a task into several subtasks,
* Map Reduce is a parallel and a distributed computing framework used to process datasets that have large scale nature on a cluster.

**Advantages of MapReduce programming**

* Scalability. Hadoop is a platform that is highly scalable.
* Cost-effective solution.
* Flexibility.
* Fast.
* Security and Authentication.
* Parallel processing.
* Availability and resilient nature.
* Simple model of programming.