Unit 4

Cloud Programming Models

 # Thread Programming

Programming applications with threads Modern applications perform multiple operations at the same time. Developers organize programs in terms of threads in order to express intra-process concurrency. The use of threads might be implicit or explicit.

Implicit threading happens when the underlying APIs use internal threads to perform specific tasks supporting the execution of applications such as graphical user interface (GUI) rendering, or garbage collection in the case of virtual machine-based languages.

Explicit threading is characterized by the use of threads within a program by application developers, who use this abstraction to introduce parallelism.

Common cases in which threads are explicitly used are I/O from devices and network connections, long computations, or the execution of background operations for which the outcome does not have specific time bounds. The use of threads was initially directed to allowing asynchronous operations—in particular, providing facilities for asynchronous I/O or long computations so that the user interface of applications did not block or became unresponsive.

 # What is a thread?

A thread identifies a single control flow, which is a logical sequence of instructions, within a process. By logical sequence of instructions, we mean a sequence of instructions that have been designed to be executed one after the other one. More commonly, a thread identifies a kind of yarn that is used for sewing, and the feeling of continuity that is expressed by the interlocked fibers of that yarn is used to recall the concept that the instructions of thread express a logically continuous sequence of operations.

Operating systems that support multithreading identify threads as the minimal building blocks for expressing running code. This means that, despite their explicit use by developers, any sequence of instruction that is executed by the operating system is within the context of a thread. As a consequence, each process contains at least one thread but, in several cases, is composed of many threads having variable lifetimes. Threads within the same process share the memory space and the execution context; besides this, there is no substantial difference between threads belonging to different processes

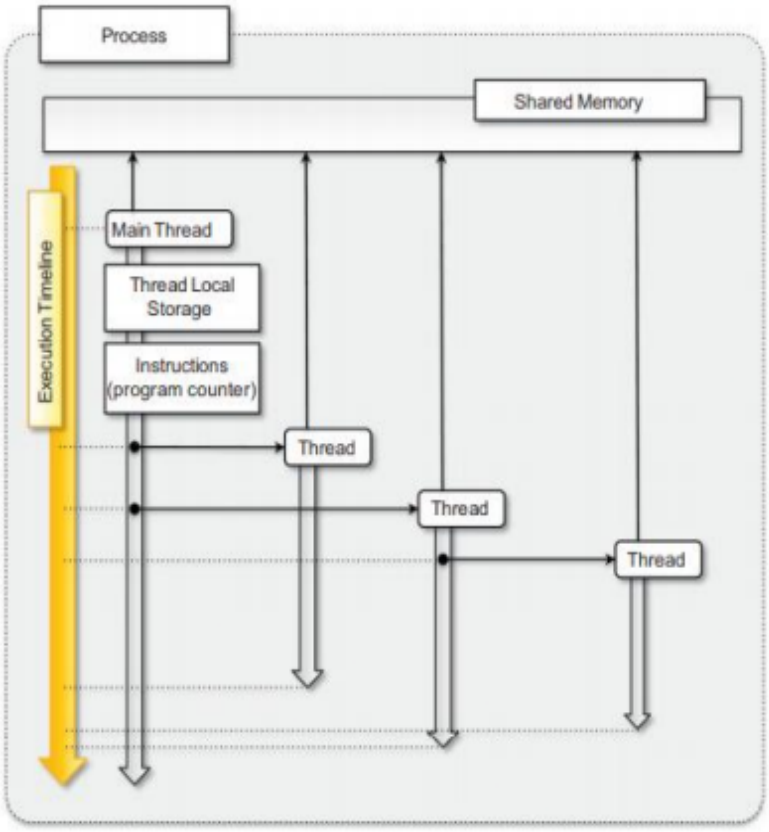


Fig: The relationship between processes and threads.

 # Thread APIs

Even though the support for multithreading varies according to the operating system and the specific programming languages that are used to develop applications, it is possible to identify a minimum set of features that are commonly available across all the implementations.

POSIX Threads

Portable Operating System Interface for Unix (POSIX) is a set of standards related to the application programming interfaces for a portable development of applications over the Unix operating system flavors.

The POSIX standard defines the following operations: creation of threads with attributes, termination of a thread, and waiting for thread completion (join operation)

The model proposed by POSIX has been taken as a reference for other implementations that might provide developers with a different interface but a similar behavior. What is important to remember from a programming point of view is the following:

• A thread identifies a logical sequence of instructions.

• A thread is mapped to a function that contains the sequence of instructions to execute.

• A thread can be created, terminated, or joined.

• A thread has a state that determines its current condition, whether it is executing, stopped, terminated, waiting for I/O, etc.

• The sequence of states that the thread undergoes is partly determined by the operating system scheduler and partly by the application developers.

• Threads share the memory of the process, and since they are executed concurrently, they need synchronization structures.

Threading support in java and .NET

Languages such as Java and C# provide a rich set of functionalities for multithreaded programming by using an object-oriented approach.

Both Java and .NET express the thread abstraction with the class Thread exposing the common operations performed on threads: start, stop, suspend, resume, abort, sleep, join, and interrupt. Start and stop/abort are used to control the lifetime of the thread instance.

Thread Programming in .Net

The c# Thread class has properties and methods for creating and controlling threads. It may be found in System.Namespace for threading

C# Thread Properties

Property	Description
CurrentThread	returns the instance of currently running thread.
IsAlive	checks whether the current thread is alive or not. It is used to find the execution status of the thread.
IsBackground	is used to get or set value whether current thread is in background or not.
ManagedThreadId	is used to get unique id for the current managed thread.
Name	is used to get or set the name of the current thread.
Priority	is used to get or set the priority of the current thread.
ThreadState	is used to return a value representing the thread state.

C# Thread Methods

A list of important methods of Thread class are given below:

Method	Description
Abort()	is used to terminate the thread. It raises ThreadAbortException.
Interrupt()	is used to interrupt a thread which is in *WaitSleepJoin* state.
Join()	is used to block all the calling threads until this thread terminates.
ResetAbort()	is used to cancel the Abort request for the current thread.
Resume()	is used to resume the suspended thread. It is obselete.
Sleep(Int32)	is used to suspend the current thread for the specified milliseconds.
Start()	changes the current state of the thread to Runnable.
Suspend()	suspends the current thread if it is not suspended. It is obselete.
Yield()	is used to yield the execution of current thread to another thread.

C# Main Thread Example

The first thread which is created inside a process is called Main thread. It starts first and ends at last.

```
using System;
using System.Threading;
public class ThreadExample
{
   public static void Main(string[] args)
   {
      Thread t = Thread.CurrentThread;
      t.Name = "MainThread";
      Console.WriteLine(t.Name);
   }
}
```

C# Threading Example

```
using System;
using System.Threading;
public class MyThread
{
   public static void Thread1()
   {
      for (int i = 0; i < 10; i++)
      {
         Console.WriteLine(i);
      }
   }
}
public class ThreadExample
{
   public static void Main()
   {
      Thread t1 = new Thread(new ThreadStart(MyThread.Thread1));
      Thread t2 = new Thread(new ThreadStart(MyThread.Thread1));
      t1.Start();
      t2.Start();
   }
}
```

C# Thread (Sleep() Method)

```
using System;
using System.Threading;
public class MyThread
{
   public void Thread1()
   {
      for (int i = 0; i < 10; i++)
      {
```

```
            Console.WriteLine(i);
            Thread.Sleep(200);
        }
    }
}
public class ThreadExample
{
    public static void Main()
    {
        MyThread mt = new MyThread();
        Thread t1 = new Thread(new ThreadStart(mt.Thread1));
        Thread t2 = new Thread(new ThreadStart(mt.Thread1));
        t1.Start();
        t2.Start();
    }
}
```

C# Thread (Abort() Method)

```
using System;
using System.Threading;
public class MyThread
{
    public void Thread1()
    {
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine(i);
            Thread.Sleep(200);
        }
    }
}
public class ThreadExample
{
    public static void Main()
```

```
    {
        Console.WriteLine("Start of Main");
        MyThread mt = new MyThread();
        Thread t1 = new Thread(new ThreadStart(mt.Thread1));
        Thread t2 = new Thread(new ThreadStart(mt.Thread1));

        t1.Start();
        t2.Start();
        try
        {
            t1.Abort();
            t2.Abort();
        }
        catch (ThreadAbortException tae)
        {
            Console.WriteLine(tae.ToString());
        }
        Console.WriteLine("End of Main");
    }
}
```

 # Parallel Computation with threads

Creating parallel applications needs a comprehensive understanding of the problem and its logical structure. Understanding the interdependence and correlations of task inside an application is critical for developing the optimal software structure and adding parallelism when necessary. A decomposition is a useful approach for determining if a problem can be broken down into components that can be done concurrently. Domain and functional decompositions are the two basic decomposition approaches.

 # Multithreading with ANEKA

- Aneka offers the capability of implementing multi-threaded applications over the Cloud by means of *Thread Programming Model*.

- The *Thread Programming Model* has been designed to transparently porting high-throughput multi-threaded parallel applications over a distributed infrastructure and provides the best advantage in the case of embarrassingly parallel applications.

#Aneka Environment

Aneka Threads

An AnekaThread is a piece of work that may be run on a remote computer. Unlike ordinary threads, each AnekaThread runs in its process on the distant system. An AnekaThread provides a comparable interface to a normal thread and may be started and aborted in the same way.

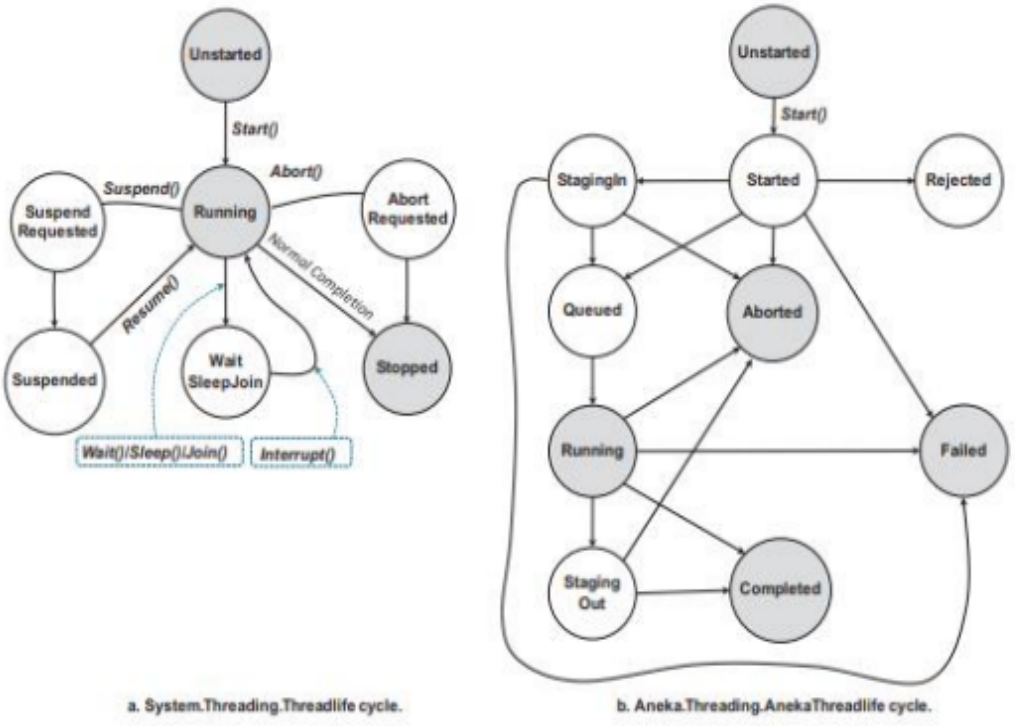
Standard .Net Threads in comparison with AnekaThreads

Aneka thread vs. common threads

Table 6.1 Thread API Comparison

.Net Threading API	Aneka Threading API
System.Threading	Aneka.Threading
Thread	AnekaThread
Thread.ManagedThreadId (int)	AnekaThread.Id (string)
Thread.Name	AnekaThread.Name
Thread.ThreadState (ThreadState)	AnekaThread.State
Thread.IsAlive	AnekaThread.IsAlive
Thread.IsRunning	AnekaThread.IsRunning
Thread.IsBackground	AnekaThread.IsBackground[false]
Thread.Priority	AnekaThread.Priority[ThreadPriority.Normal]
Thread.IsThreadPoolThread	AnekaThread.IsThreadPoolThread [false]
Thread.Start	AnekaThread.Start
Thread.Abort	AnekaThread.Abort
Thread.Sleep	[Not provided]
Thread.Interrupt	[Not provided]
Thread.Suspend	[Not provided]
Thread.Resume	[Not provided]
Thread.Join	AnekaThread.Join

Thread Life Cycle



Thread Synchronization

The .Net framework includes many synchronization primitives for regulating local thread interactions and preventing race conditions, such as locking and signaling. The Aneka threading library only offers a single synchronization technique through the AnekaThread.Join method.

Invoking AnekaThread's join method causes the main application thread to block until the AnekaThread quits, either successfully or unsuccessfully.

Thread Priorities

The System.Threading.Thread class supports thread priorities, where the scheduling priority can be one selected from one of the values of the ThreadPriority enumeration: Highest, AboveNormal, Normal, BelowNormal, or Lowest. However, operating systems are not required to honor the priority of a thread, and the current version of Aneka does not support thread priorities

Techniques for parallel computation with threads

Decomposition is a useful technique that aids in understanding whether a problem is divided into components (or tasks) that can be executed concurrently. The two main decomposition/partitioning techniques are domain and functional decompositions.

Domain decomposition

Domain decomposition is the process of identifying patterns of functionally repetitive, but independent, computation on data.

. The master-slave model is a quite common organization for these scenarios:

• The system is divided into two major code segments.

• One code segment contains the decomposition and coordination logic.

• Another code segment contains the repetitive computation to perform.

• A master thread executes the first code segment.

• As a result of the master thread execution, as many slave threads as needed are created to execute the repetitive computation.

• The collection of the results from each of the slave threads and an eventual composition of the final result are performed by the master thread.

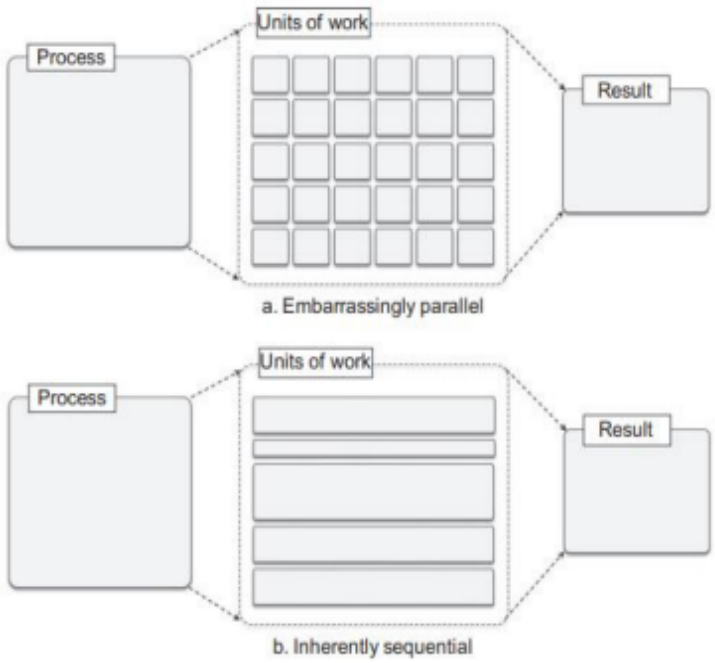
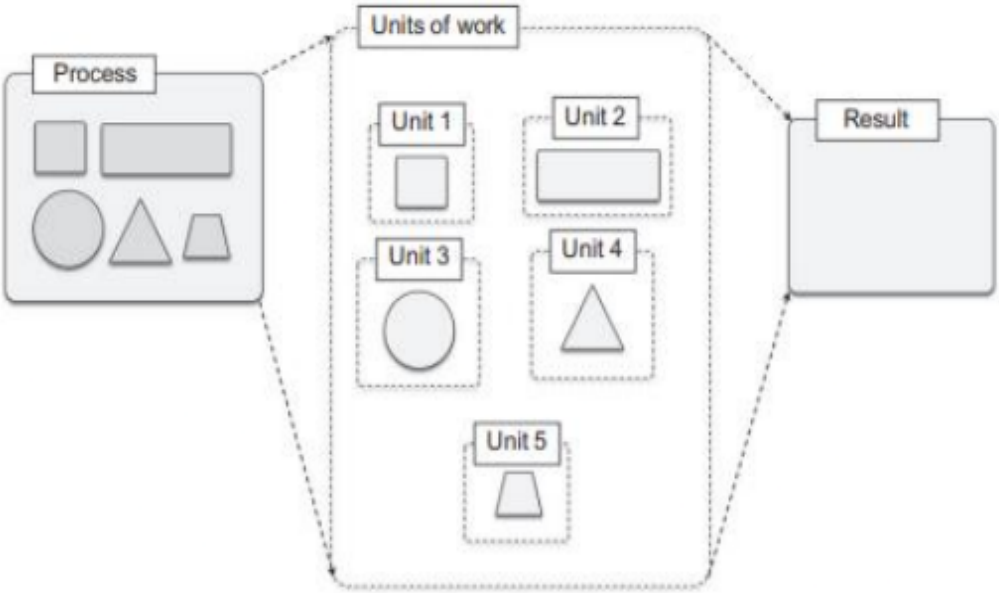


Fig: Domain Decomposition Technique

Functional decomposition

- Functional decomposition is the process of identifying functionally distinct but independent computations.
- The focus here is on the type of computation rather than on the data manipulated by the computation.

- This kind of decomposition is less common and does not lead to the creation of a large number of threads, since the different computations that are performed by a single program are limited.



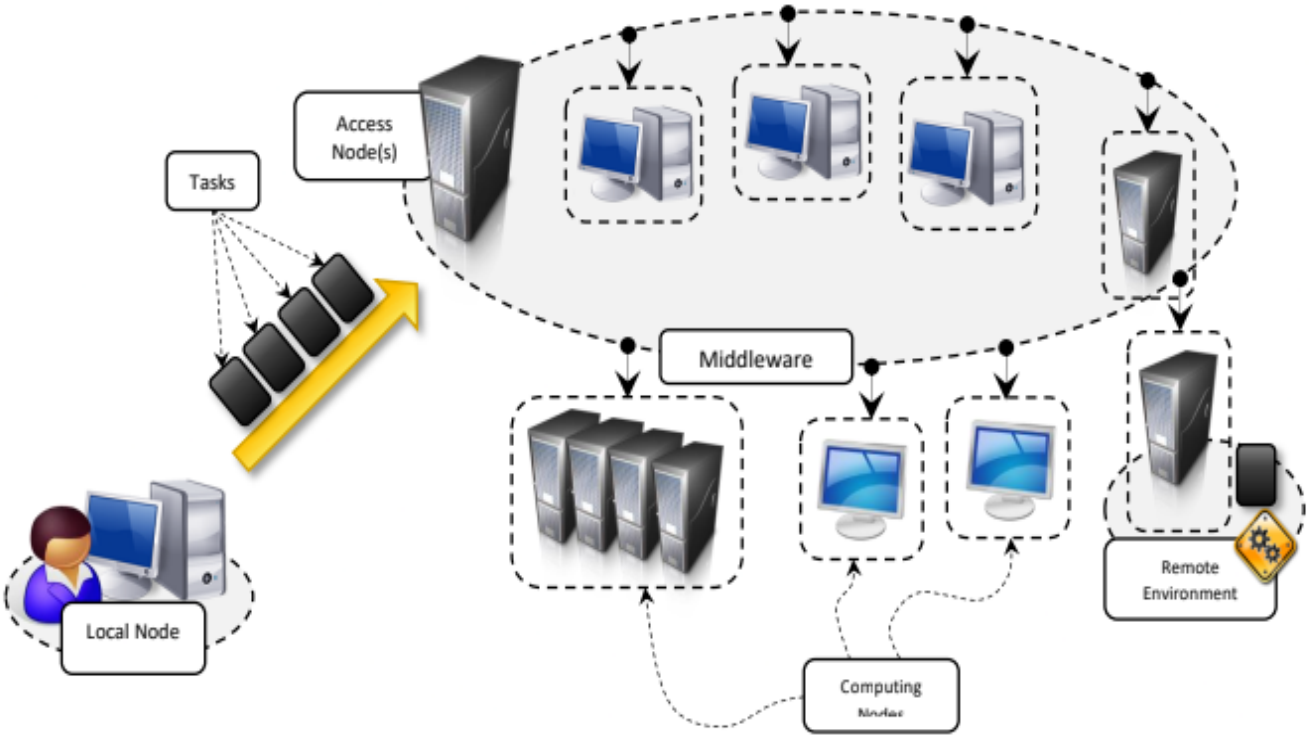
 # Task Programming

- A task generally represents a program, which might require input files and produce output files as a result of its execution.
- Applications are then constituted by a collection of tasks.
- These are submitted for execution and their output data is collected at the end of their execution.
- The way in which tasks are generated, the order in which they are executed, or whether they need to exchange data, differentiate the application models falling under the umbrella of task programming.
- Organizing an application in terms of tasks is the most intuitive and common practice for developing parallel and distributed computing applications.
- 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 runtime environment.

- Programs are the most common option for representing tasks, especially in the field of scientific computing, which has leveraged distributed computing for its computational needs.

 # Task Computing Scenario

- Multi-threaded programming is mainly concerned with providing a support for parallelism within a single machine.
- Task computing provides distribution by utilizing the compute power of several computing nodes.
- Now Clouds have emerged as an attractive solution to obtain a huge computing power on-demand for the execution of distributed applications.
- In order to achieve it, a suitable middleware is needed.



- The middleware is a software layer that enables the coordinated use of multiple resources, which are drawn from a Data Center 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, which may vary from implementation to implementation (a simple shell, a sandboxed environment, or a virtual machine).
- Task submission is normally done by using the APIs provided by the middleware, whether they are a web or a programming language interface.
- Appropriate APIs are also provided to monitor task status and collect their results upon the completion.

 # Computing Categories

- According to the specific nature of the problem, different categories for task computing have been proposed over time.
- These categories do not enforce any specific application model but provide an overall view of the characteristics of the problems.
- They implicitly impose requirements on the infrastructure and the middleware.
- Applications falling in this category are:
  - *High-Performance Computing (HPC)*
  - *High-Throughput Computing (HTC)*
  - *Many Tasks Computing (MTC)*.

High Performance Computing

- High Performance Computing is the use of distributed computing facilities for solving problems that need large computing power.
- Historically, supercomputers and clusters are specifically designed to support HPC applications that are developed to solve grand challenging problems in science and engineering.
- The general profile of HPC applications is constituted by a large collection of compute intensive tasks which needs to be processed in a short period of time.
- It is common to have parallel and tightly coupled tasks, which require low latency interconnection network to minimize the data exchange time.

High Throughput Computing

- High Throughput Computing is the use of distributed computing facilities for applications requiring large computing power over a long period of time.
- HTC systems need to be robust and reliably operate over a long time scale.
- Traditionally, computing Grids composed of heterogeneous resources (clusters, workstations, and volunteer desktop machines), have been used to support HTC.

- The general profile of HTC applications is that are made up of a large number of tasks, whose execution can last for a considerable amount of time (i.e. weeks or months).
- Classical examples of such applications are scientific simulations or statistical analyses.
- It is quite common to have independent tasks that can be scheduled in distributed resources as they do not need to communicate.
- HTC systems measure their performance in terms of jobs completed per month.

Many Task Computing

- Many Tasks Computing model started receiving attention recently and it covers a wide variety of applications.
- It aims to bridge the gap between HPC and HTC.
- MTC is similar to High Throughput Computing, but it concentrates on the use of many computing resources over a short period of time to accomplish many computational tasks.
- In brief, MTC denotes high-performance computations comprising multiple distinct activities, coupled via file system operations.

 # Frameworks for Task Computing

Several frameworks are available to allow the execution of task-based applications on distributed computing resources such as clouds. Some popular software systems supporting task computing framework are:

a. *Globus Toolkit*
b. *Sun Grid Engine (SGE)*
c. *BOINC (The Berkeley Open Infrastructure for Networking Computing)*
d. *Nimrod/G*
e. *Aneka*.

 # Task Based Application Models

- There exist several models that are based on the concept of task as the fundamental unit for composing distributed applications.
- What makes them different is the way in which tasks are generated, the relations they have with each other, the presence of dependencies, or other conditions.
- for example a specific set of services in the runtime environment, that have to be met.
- Here, we quickly review the most common and popular models based on the concept of task.

- Embarrassingly Parallel Applications
- Parameter Sweep Applications
- MPI Applications

Embarrassingly Parallel Applications

- Embarrassingly parallel applications constitute the most simple and intuitive category of distributed applications.
- As already discussed in the previous chapter, embarrassingly parallel applications are constituted by a collection of tasks that are independent from each other and that can be executed in any order.
- The tasks might be of the same type or of different nature and they do not need to communicate among themselves.
- This category of applications is supported by the majority of the frameworks for distributed computing.
- Since tasks do not need to communicate, there is a lot of freedom for the way in which they are scheduled.
- Tasks can be executed in any order and there is no specific requirement for tasks to be executed at the same time.
- Therefore, scheduling of these applications is simplified and mostly concerned with the optimal mapping of tasks to available resources.
- Frameworks and tools supporting embarrassingly parallel applications are the *Globus Toolkit*, *BOINC*, and ***Aneka.***

Parameter Sweep Applications

- Parameter sweep applications are a type of application in which the same code is executed numerous times with different sets of input parameter values.
- This comprises altering one parameter throughout a range of values or adjusting numerous parameters throughout a vast multidimensional space.

The pseudo-code explains the use of parameter sweeping for the execution of a generic evolutionary algorithm.

```
individuals= {100,200,300,500,1000}

generations= {50,100,200,400}

foreach indiv in individuals do

foreach generation in generations do
```

```
Task=generate_task(indiv,generation)
Submit_task(Task)
```

The example defines a bidimensional domain comprised of discrete variables which are iterated over each combination of two variables – individuals and generations. This will generate 20 task composing the application.

Message Passing Interface

- MPI provides developers with a set of routines that:
  - Manage the distributed environment where MPI programs are executed.
  - Provide facilities for point to point communication.
  - Provide facilities for group communication.
  - Provide support for data structure definition and memory allocation.
  - Provide basic support for synchronization with blocking calls.
- The general reference architecture is depicted. A distributed application in MPI is composed by a collection of MPI processes which are executed in parallel in a distributed infrastructure supporting MPI (most likely a cluster or nodes leased form Clouds).

 # Data Intensive Computing

Data intensive computing is concerned with the generation, manipulation, and analysis of massive amounts of data ranging from hundreds of megabytes (MB) to petabytes (PB) and beyond. The term dataset refers to a collection of information pieces that are useful to one or more applications. Datasets are frequently stored in repositories, which are infrastructures that allow for the storage, retrieval and indexing of enormous volumes of data.

 # Technologies for Data Intensive Computing

The creation of applications that are primarily focused on processing massive amounts of data is referred to as data intensive computing.

Storage Systems

Because of the growth of unstructured data in the form of blogs, Web pages, software logs, and sensor readings, the relational model in its original formulation does not appear to be the preferable approach for enabling large-scale data analytics. Various fields such as scientific computing, enterprise applications, media entertainment, natural language processing and social network analysis generate a very large quantity of data which is considered a business asset as businesses

are realizing the importance of data analytics. So, such volume of data requires more efficient data management techniques.

File systems such as Lustre, IBM General Parallel File System (FPFS), Google File System (GFS), Sector/Sphere, Amazon Simple Storage Service (S3) are widely used for high performance distributed file systems and storage clouds.

a. **Lustre**: Lustre file system is an open source, parallel file system that meets many of the needs of high performance computing settings. Lustre is meant to enable access to petabytes (PBs) of storage while serving thousangs of clients at hundreds s of gigabytes per second.
b. **General Parallel File System (GPFS):** This is the high performance distributed file system developed by IBM that provides support for the RS/6000 super computer and linux computing clusters. It is a cluster file system that allows numerous nodes to access a single file system or collection of file systems at the same time.
c. **Google File System (GFS):** It is a storage architecture that allows distributed application to run on Google's computing cloud. GFS is designed to run on multiple, standard x86-based servers and is intended to be fault tolerant, highly available distributed file systems, such as scalability, performance, dependability and resilience.
d. **Amazon Simple Storage Service (Amazon S3):** It is storage for the internet which is designed to make web-scale computing easier. Despite the internal specification is not disclosed, the system is said to enable high availability, dependability, scalability, security, high performance, limitless storage, and minimal latency at cheap cost.

NoSQL systems

After the file system, NoSQL systems also play a vital role in the field of intensive data processing. NoSQL is a database type that stores and retrieves data without previously defining its structure- an alternative to more strict relational databases. Some of the implementations that enable data-intensive applications are:

Apache CouchDB: it is a NoSQL document database that is open source that gathers and stores data in JSON-based document formats.

MongoDB: It is a major NoSQL database and an open-source document database developed in C++. It is a document-oriented NoSQL database that is used for large-scale data storage.

Amazon DynamoDB: It is a key-value and document database that promises performance in single digit milliseconds at any size. DynamoDB can handle more than 10 trillion request per day, with peaks of more tan 20 million request per second.

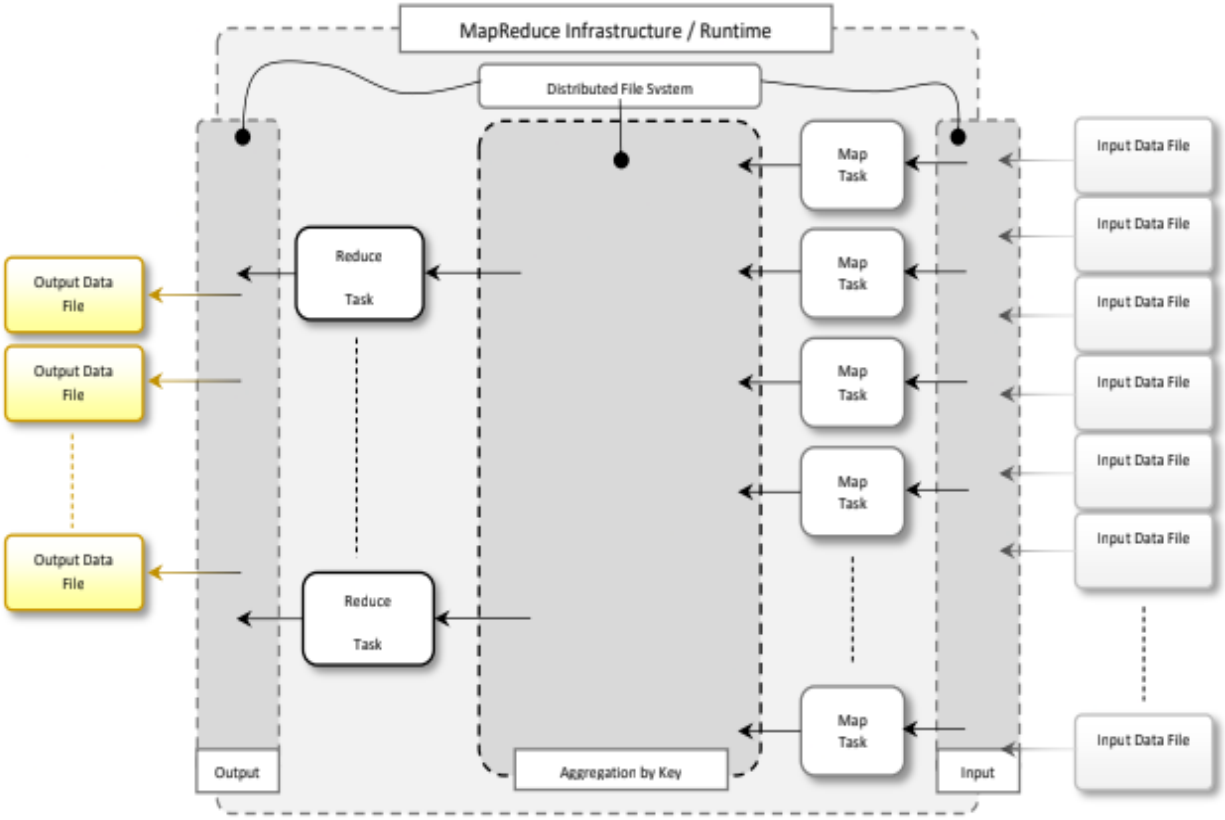
Google Bigtable: It is a distributed, column-oriented data stored developed by google to manage massive volume of structured data related to the company's Internet search and web services operations. It is fully managed, scalable, NoSQL database service for large analytical and operational workloads with up to 99.99 % availability.

HBase: It is a non-relational column-oriented database management system that works on the top of Hadoop distributed file system (HDFS). HBase was inspired by Google Bigtable in its design. HBase provides a fault-tolerant method of storing sparse data sets, which are common in many big data applications. It's ideal for real time data processing or random read/write access to enormous amounts of data.

Map-Reduce Programming

- MapReduce is a programming platform introduced by Google for processing large quantities of data.
- It expresses the computation logic of an application into two simple functions: *map* and *reduce*.
- Data transfer and management is completely handled by the distributed storage infrastructure (i.e. the Google File System), which is in charge of providing access to data, replicate files, and eventually move them where needed.
- Therefore, developers do not have to handle anymore these issues and are provided with an interface that presents data at a higher level: as a collection of key-value pairs.
- The computation of MapReduce applications is then organized in a workflow of map and reduce operations that is entirely controlled by the runtime system and developers have only to specify how the map and reduce functions operate on the key value pairs.
- More precisely, the model is expressed in the form of two functions, which are defined as follows:
- *map (k1,v1)* → *list(k2,v2)*
- *reduce(k2,list(v2))* → *list(v2)*
- The *map* function reads a key-value pair and produces a list of key-value pairs of different types.
- The *reduce* function reads a pair composed by a key and a list of values and produces a list of values of the same type.
- The types (*k1,v1,k2,kv2*) used in the expression of the two functions provide hints on how these two functions are connected and are executed to carry out the computation of a MapReduce job: the output of map tasks is aggregated together by grouping the values according to their corresponding keys and constitute the input of reduce tasks that, for each of the keys found, reduces the list of attached values to a single value.
- Therefore, the input of a MapReduce computation is expressed as a collection of key-value pairs <*k1,v1*> and the final output is represented by a list values: *list(v2)*.

MapReduce Computation Workflow



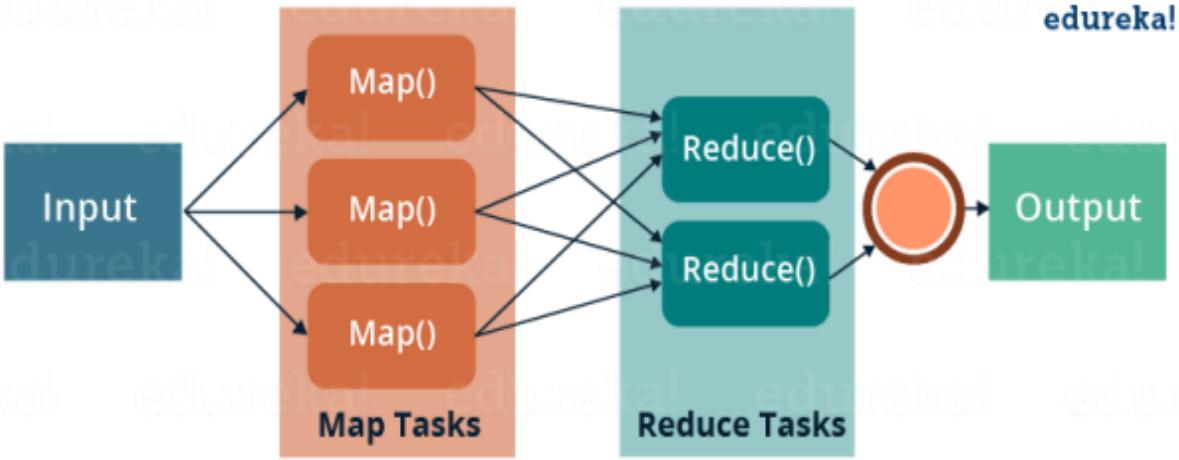
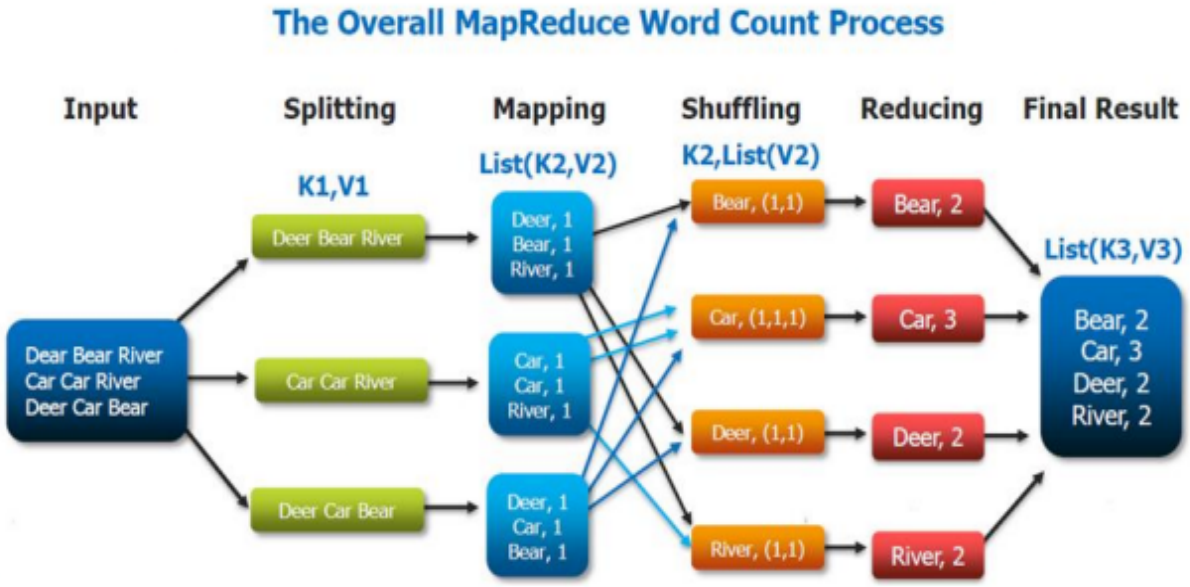


Fig: MapReduce



Benefits of MapReduce

a. It is fault-tolerant; during the middle of a map-reduce job, if a machine carrying a few data blocks fails, the architecture handles the failure.
b. By splitting the jobs, MapReduce tasks may process several portions of the same dataset concurrently. This has the advantage of completing tasks in less time.
c. Multiple copies of same data are delivered to various network nodes. As a result, in the event of a failure, alternative copies are quickly available for processing with no loss.
d. Each node sends a status update to the master node regularly. If a slave node fails to transmit its notice, the master node reassigns the slave node's current job to another available node in the cluster.

Enterprise Batch Processing using MapReduce

Data is the new money in the contemporary world. The data generated by today's enterprises has been increasing at exponential rates in size from the most recent couple of years. Data-intensive calculations are widespread in many application areas. Computational science is one of the most well-known. People who conduct scientific simulations and experiments are often eager to generate, review, and analyze large volumes of data. Other IT business fields, in addition to scientific computing, also require help from the data-intensive computation.

With such a large data volume, it will be difficult for a single server- or node- to handle. As a result, code that runs on several nodes is required. Because writing distributed systems present an infinite number of challenges. MapReduce is a framework that lets users develop code that runs on numerous nodes without worrying about fault tolerance, dependability, synchronization, or availability.

Batch processing, in a nutshell, is a way of waiting and performing everything periodically such as at the end of the day, week, or month. In the enterprise, during the specified period, the cumulative data will be large. So, to handle such big data, distributed computing environment, and the MapReduce technique can play a vital role.

Comparison Between Thread, Task and Map-Reduce

A thread is a fundamental unit of CPU utilization that consists of a program counter, a stack, and a collection of registers. Threads have their program and memory areas. A thread of execution is the shortest series of programmed instructions that a scheduler can handle separately. Threads are a built-in feature of operating system. The thread class provided by different programming languages such as .Net or Java provides a method for creating and managing threads.

A task is anything that you want to be completed that is a higher-level abstraction on top of threads. It is a collection of software instructions stored in memory. When a software instruction is placed into memory, it is referred to as a process or task. The task can inform you if it has been completed and whether the procedure has produced a result.

MapReduce is a framework that allows us to design programs that can process massive volumes of data in parallel on vast clusters of commodity hardware in a dependable manner. MapReduce is a programming architecture for distributed programming. The MapReduce method consists of two key tasks: Map and Reduce. Map translates one collection of data into another, where individual pieces are split down into tuples (key/value pairs). Second, there is the reduction job, which takes the result of a map as an input and merges those data tuples into a smaller collection of tuples. The reduction work is always executed after the map job, as the name MapReduce indicates.