USN | | | | | | | | | | |

Internal Assessment Test 2 – October 2019

Sub:	**Cloud Computing and its Applications**					Sub Code:	**15CS742**	Branch:	**CSE**	
Date:	12 / 10 / 2019	Duration:	90 min's	Max Marks:	50	Sem / Sec:	**VII**/ A,B & C		OBE	

Answer any FIVE FULL Questions

		MARKS	CO	RBT
1 (a)	Discuss in detail about threads	[05]	**CO2**	**L2**
(b)	Explain the relation between process and thread with the help of a relevant diagram	[05]	**CO2**	**L1**
2 (a)	Describe MPI program structure with a neat diagram	[05]	**CO2**	**L1**
(b)	Illustrate developing parameter sweep application on Aneka	[05]	**CO1**	**L2**
3 (a)	Differentiate Aneka threads with Common threads	[10]	**CO3**	**L2**
4 (a)	Distinguish between domain and functional decomposition techniques with illustrative examples	[10]	**CO3**	**L2**

CI **CCI** **HOD**

USN | | | | | | | | | | |

Internal Assessment Test 2 – October 2019

Sub:	**Cloud Computing and its Applications**					Sub Code:	**15CS742**	Branch:	**CSE**	
Date:	12 / 10 / 2019	Duration:	90 min's	Max Marks:	50	Sem / Sec:	**VII**/ A,B & C		OBE	

Answer any FIVE FULL Questions

		MARKS	CO	RBT
1 (a)	Discuss in detail about threads	[05]	**CO2**	**L2**
(b)	Explain the relation between process and thread with the help of a relevant diagram	[05]	**CO2**	**L1**
2 (a)	Describe MPI program structure with a neat diagram	[05]	**CO2**	**L1**
(b)	Illustrate developing parameter sweep application on Aneka	[05]	**CO1**	**L2**
3 (a)	Differentiate Aneka threads with Common threads	[10]	**CO3**	**L2**
4 (a)	Distinguish between domain and functional decomposition techniques with illustrative examples	[10]	**CO3**	**L3**

CI **CCI** **HOD**

5 (a)	Explain work flow with practical example	[06]	**CO2**	**L2**
(b)	Describe two work flow technologies	[04]	**CO1**	**L1**
6 (a)	Explain the importance of computation and communication with respect to the design of parallel and distributed applications.	[06]	**CO3**	**L3**
(b)	Discuss about POSIX threads	[04]	**CO3**	**L2**
7 (a)	Describe the different task based application models	[06]	**CO1**	**L1**
(b)	What is data- intensive computing? Describe the open challenges in data-intensive computing	[04]	**CO3**	**L3**

CI **CCI** **HOD**

5 (a)	Explain work flow with practical example	[06]	**CO2**	**L2**
(b)	Describe two work flow technologies	[04]	**CO1**	**L1**
6 (a)	Explain the importance of computation and communication with respect to the design of parallel and distributed applications.	[06]	**CO3**	**L3**
(b)	Discuss about POSIX threads	[04]	**CO3**	**L2**
7 (a)	Describe the different task based application models	[06]	**CO1**	**L1**
(b)	What is data- intensive computing? Describe the open challenges in data-intensive computing	[04]	**CO3**	**L2**

CI **CCI** **HOD**

CO PO Mapping

Course Outcomes		Modules covered	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3	PSO4
CO1	Explain cloud computing, virtualization and classify services of cloud computing	1,2	2	1	2	-	1	-	-	-	-	-	-	-	-	1	-	-
CO2	Illustrate architecture and programming in cloud	2,3,4	2	2	2	1	2	1	-	-	-	-	-	-	1	1	-	1
CO3	Describe the platforms for development of cloud applications	4,5	2	2	2	2	2	1	-	-	-	-	-	-	2	2	-	1
CO4	List the applications of cloud	4,5	2	2	2	2	2	1	-	-	-	-	-	-	2	2	-	1

COGNITIVE LEVEL	REVISED BLOOMS TAXONOMY KEYWORDS
L1	List, define, tell, describe, identify, show, label, collect, examine, tabulate, quote, name, who, when, where, etc.
L2	summarize, describe, interpret, contrast, predict, associate, distinguish, estimate, differentiate, discuss, extend
L3	Apply, demonstrate, calculate, complete, illustrate, show, solve, examine, modify, relate, change, classify, experiment, discover.
L4	Analyze, separate, order, explain, connect, classify, arrange, divide, compare, select, explain, infer.
L5	Assess, decide, rank, grade, test, measure, recommend, convince, select, judge, explain, discriminate, support, conclude, compare, summarize.

PROGRAM OUTCOMES (PO), PROGRAM SPECIFIC OUTCOMES (PSO)				CORRELATION LEVELS	
PO1	Engineering knowledge	PO7	Environment and sustainability	0	No Correlation
PO2	Problem analysis	PO8	Ethics	1	Slight/Low
PO3	Design/development of solutions	PO9	Individual and team work	2	Moderate/ Medium
PO4	Conduct investigations of complex problems	PO10	Communication	3	Substantial/ High
PO5	Modern tool usage	PO11	Project management and finance		
PO6	The Engineer and society	PO12	Life-long learning		
PSO1	Develop applications using different stacks of web and programming technologies				
PSO2	Design and develop secure, parallel, distributed, networked, and digital systems				
PSO3	Apply software engineering methods to design, develop, test and manage software systems.				
PSO4	Develop intelligent applications for business and industry				

Scheme Of Evaluation
Internal Assessment Test 2 – Oct 2019

Sub:	Cloud Computing and its Applications							Code:	15CS742
Date:	12/10/2019	Duration:	90mins	Max Marks:	50	Sem:	VII	Branch:	A, B, C

Note: Answer Any Five Questions

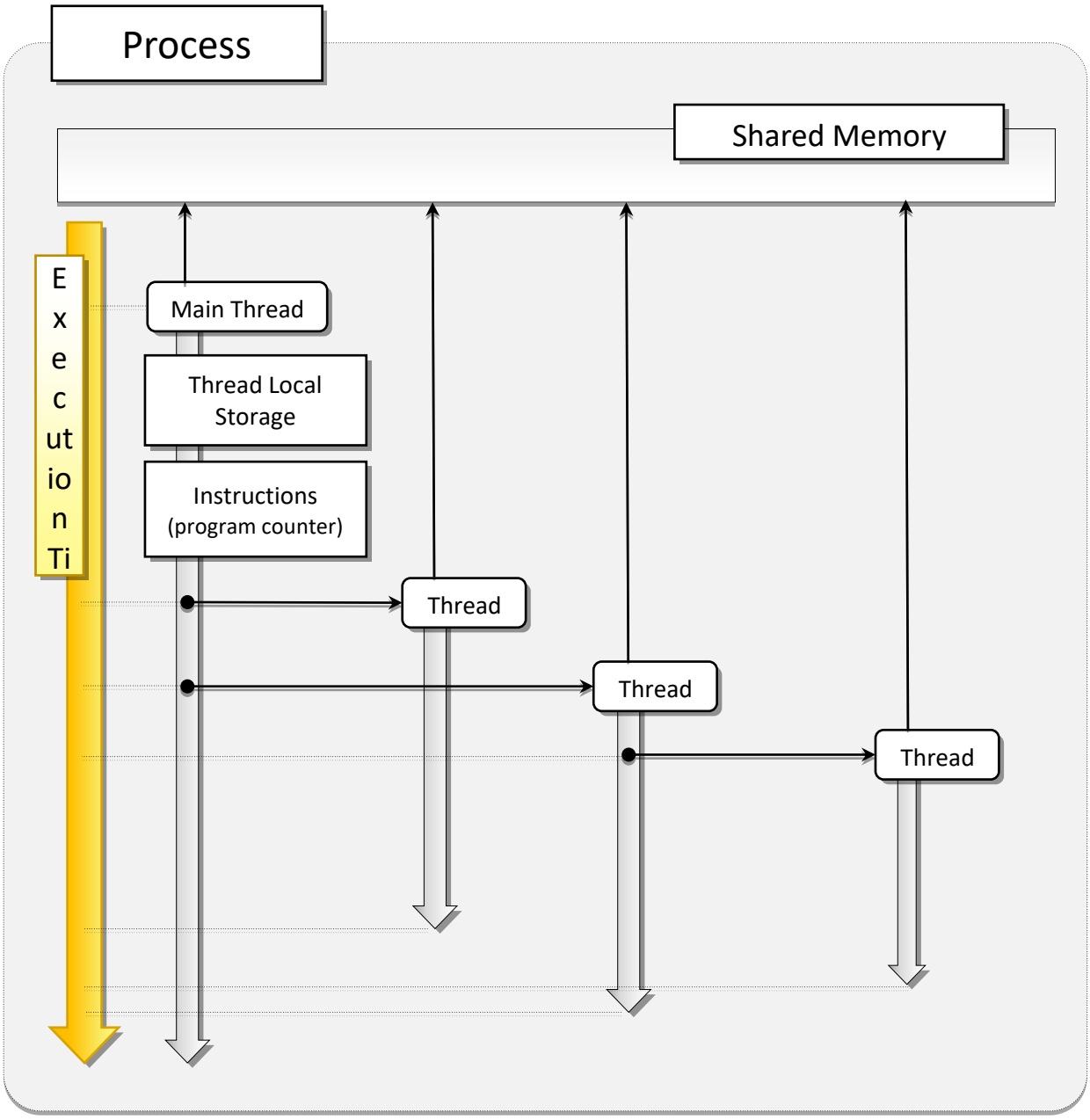
Question #		Description	Marks Distribution		Max Marks
1	a)	**Discuss in detail about threads** ♦ What is a thread? ♦ Operating System support for thread ♦ Context switching ♦ Implicit and explicit threading	1 M 1 M 2 M 1 M	5 M	10 M
	b)	**Explain the relation between process and thread with the help of a relevant diagram** ♦ Pictorial representation ♦ What is main thread and how it relates to execution ♦ Any example or code.	2 M 2 M 1 M	5 M	
2	a)	**Describe MPI program structure with neat diagram** ♦ Need of MPI ♦ Set of routines supported by programming language ♦ MPI program structure	1 M 2M 2M	5 M	10 M
	b)	**Illustrate developing parameter sweep application on Aneka** ♦ Explain PSM ♦ What is Namespace ♦ 3 Namespaces: o Aneka.PSM.Core o Aneka.PSM.Workbench o Aneka.PSM.Console	1 M 1M 3M	5 M	
3		**Differentiate Aneka threads with Common threads** ♦ Interface Compatibility ♦ Thread Life Cycle	2 M 2 M	10 M	10 M

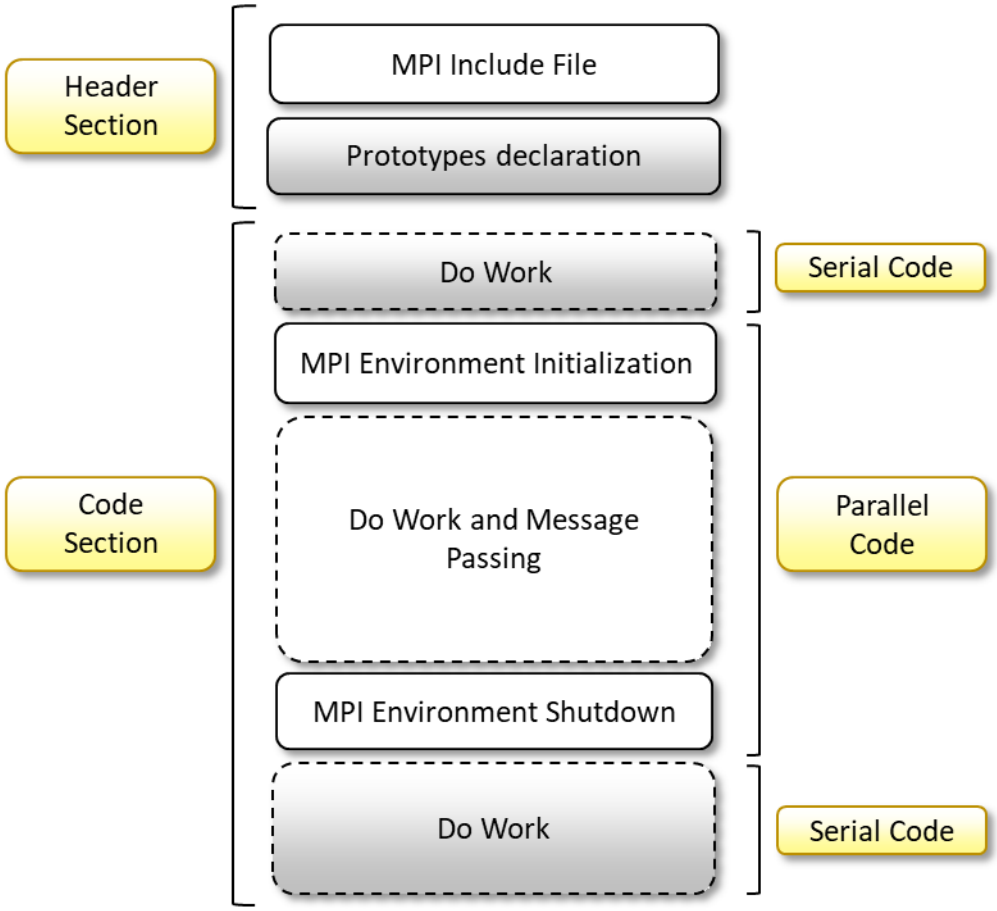
		♦ Thread Synchronization ♦ Thread Priorities ♦ Type Serialization	2 M 2 M 2 M		
4		Distinguish between domain and functional decomposition techniques with illustrative examples ♦ Definition of Domain Decomposition ♦ Use Cases ♦ Pictorial Representation ♦ Definition of Functional Decomposition ♦ Use Cases ♦ Pictorial Representation	2M 2 M 1M 2M 2 M 1M	10 M	10 M
5	a)	**Explain workflow with practical example** ♦ What is workflow ♦ Scientific workflow ♦ Directed Acyclic Graph (DAG) ♦ Satellite image processing (Example)	1 M 1 M 1 M 3 M	6 M	10 M
	b)	**Describe two workflow technologies** Explain any of two items from the following: ♦ Kepler ♦ DAGMan ♦ Cloudbus Workflow Management System ♦ Offspring	2 M 2 M	4 M	
6	a)	**Explain the importance of computation and communication with respect to the design of parallel and distributed applications** ♦ Two assumptions on the computations ♦ Queuing techniques for threads ♦ Why communication between threads are required	2 M 2 M 2 M	6 M	10 M
	b)	**Discuss about POSIX threads** ♦ Abbreviation and which OS it is used for ♦ Points to be remembered from programming perspective such as: o Logical sequence o Life of thread o Status of thread o Synchronization structure ♦ Programming Language compatibility	1 M 2M 1 M	4 M	
7	a)	**Describe the different task-based application models** Briefly discuss about following three: ♦ Embarrassingly parallel application ♦ Parameter sweep application ♦ MPI applications	2 M 2M 2 M	6 M	10 M

	b)	**What is data- intensive computing? Describe the open challenges in data-intensive computing** ♦ Justify the model with Peta Bytes of Data and rate is more ♦ Write any 3 from the below open challenges: o Scaling of dataset o In memory data structure handling o Data signature generation techniques o Hybrid interconnection architectures o Distributed file systems	1 M 3 M	4 M	

Internal Assessment Test 1 – March 2019										
Sub:	Cloud Computing and its Applications					Sub Code:	15CS742	Branch:	CSE	
Date:	12/10/19	Duration:	90 mins	Max Marks:	50	Sem / Sec:	VII A/B/C		OBE	

	Answer any FIVE FULL Questions	MARKS	CO	RBT
1 (a)	Discuss in detail about threads **Answer:** 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. Consequently, 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	[05]	CO2	L2
1(b)	Explain the relation between process and thread with the help of a relevant diagram **Answer:** Figure below provides an overview of the relation between threads and processes and a simplified representation of the runtime execution of a multithreaded application. A running program is identified by a process, which contains at least one thread, also called the main thread. Such a thread is implicitly created by the compiler or the runtime environment executing the program. This thread is likely to last for the entire lifetime of the process and be the origin of other threads, which in general exhibit a shorter duration. As main threads, these threads can spawn other threads. There is no difference between the main thread and other threads created during the process lifetime. Each of them has its own local storage and a sequence of instructions to execute, and they all share the memory space allocated for the entire process. The execution of the process is considered terminated when all the threads are completed.	[05]	CO2	L1


Process
Shared Memory
Execution Ti
Main Thread
Thread Local Storage
Instructions (program counter)
Thread
Thread
Thread

2 (a)	Describe MPI program structure with a neat diagram **Answer:** Message Passing Interface (MPI) is a specification for developing parallel programs that communicate by exchanging messages. Compared to other models of task computing, MPI introduces the constraint of communication that involves MPI tasks that need to run at the same time. MPI has originated as an attempt to create common ground from the several distributed shared memory and message-passing infrastructures available for distributed computing. Nowadays, MPI has become a de facto standard for developing portable and efficient message passing HPC applications. Interface specifications have been defined and implemented for C/C11 and Fortran. To create an MPI application it is necessary to define the code for the MPI process that will be executed in parallel. This program has, in general, the structure described in Figure below. The section of code that is executed in parallel is clearly identified by two operations that set up the MPI environment and shut it down, respectively. In the code section defined within these two operations, it is possible to use all the MPI functions to send or receive messages in either asynchronous or synchronous mode.  	[05]	CO2	L1

2(b)	Illustrate developing parameter sweep application on Aneka **Answer:** Aneka integrates support for parameter-sweeping applications on top of the task model by means of a collection of client components that allow developers to quickly prototype applications through either programming APIs or graphical user interfaces (GUIs). The set of abstractions and tools supporting the development of parameter sweep applications constitutes the Parameter Sweep Model (PSM). The PSM is organized into several namespaces under the common root Aneka.PSM. More precisely: • Aneka.PSM.Core (Aneka.PSM.Core.dll) contains the base classes for defining a template task and the client components managing the generation of tasks, given the set of parameters. • Aneka.PSM.Workbench (Aneka.PSM.Workbench.exe) and Aneka.PSM.Wizard (Aneka.PSM. Wizard.dll) contain the user interface support for designing and monitoring parameter sweep applications. Mostly they contain the classes and components required by the Design Explorer, which is the main GUI for developing parameter sweep applications. • Aneka.PSM.Console (Aneka.PSM.Console.exe) contains the components and classes supporting the execution of parameter sweep applications in console mode. These namespaces define the support for developing and controlling parameter sweep applications on top of Aneka.	[05]	CO1	L2

3 (a)	Differentiate Aneka threads with Common threads **Answer:** To efficiently run on a distributed infrastructure, Aneka threads have certain limitations compared to local threads. These limitations relate to the communication and synchronization strategies that are normally used in multithreaded applications. **Distinction based on Interface compatibility** The Aneka.Threading.AnekaThread class exposes almost the same interface as the System.Threading.Thread class with the exception of a few operations that are not supported. The reference namespace that defines all the types referring to the support for threading is Aneka.Threading rather than System.Threading. The basic control operations for local threads such as Start and Abort have a direct mapping, whereas operations that involve the temporary interruption of the thread execution have not been supported. The reasons for such a design decision are twofold. First, the use of the Suspend/Resume operations is generally a deprecated practice, even for local threads, since Suspend abruptly interrupts the execution state of the thread. Second, thread suspension in a distributed environment leads to an ineffective use of the infrastructure, where resources are shared among different tenants and applications. This is also the reason that the Sleep operation is not supported. Therefore, there is no need to support the Interrupt operation, which forcibly resumes the thread from a waiting or a sleeping state. To support synchronization among threads, a corresponding implementation of the Join operation has been provided. Besides the basic thread control operations, the most relevant properties have been implemented, such as name, unique identifier, and state. Whereas the name can be freely assigned, the identifier is generated by Aneka, and it represents a globally unique identifier (GUID) in its string form rather than an integer. Properties such as IsBackground, Priority, and IsThreadPoolThread have been provided for interface compatibility but actually do not have any effect on thread scheduling. Other properties concerning the state of the thread, such as IsAlive and IsRunning, exhibit the expected behavior, whereas a slightly different behavior has been implemented for the ThreadState property that is mapped to the State property. The remaining methods of the System.Threading.Thread class (.NET 2.0) are not supported.	[10]	CO3	L2

Finally, it is important to note differences in thread creation. Local threads implicitly belong to the hosting process and their range of action is limited by the process boundaries. To create local threads it is only necessary to provide a pointer to a method to execute in the form of the ThreadStart or ParameterizedThreadStart delegates. Aneka threads live in the context of a distributed application, and multiple distributed applications can be managed within a single process; for this reason, thread creation also requires the specification of the reference to the application to which the thread belongs.

Interface compatibility between Aneka threading APIs and the base class library allow quick porting of most of the local multithreaded applications to Aneka by simply replacing the class names and modifying the thread constructors.

Distinction based on Thread life cycle

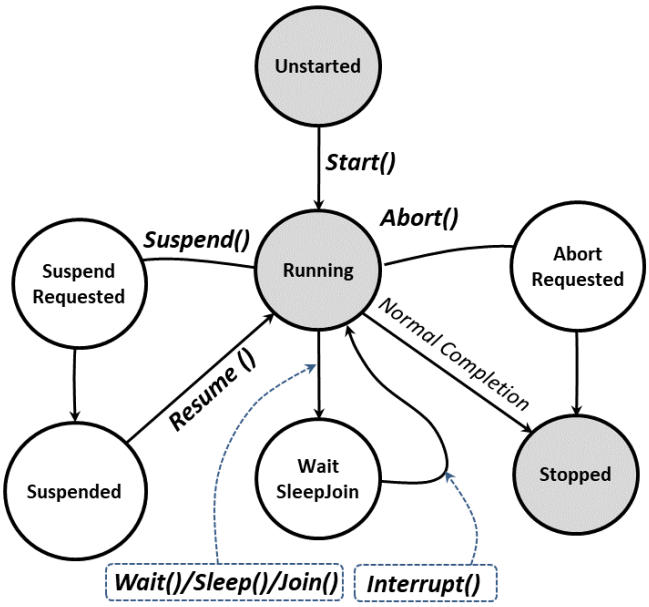
Since Aneka threads live and execute in a distributed environment, their life cycle is necessarily different from the life cycle of local threads. For this reason, it is not possible to directly map the state values of a local thread to those exposed by Aneka threads. Figure below provides a comparative view of the two life cycles.

The white balloons in the figure indicate states that do not have a corresponding mapping on the other life cycle; the shaded balloons indicate the common states. Moreover, in local threads most of the state transitions are controlled by the developer, who actually triggers the state transition by invoking methods on the thread instance, whereas in Aneka threads, many of the state transitions are controlled by the middleware. As depicted in Figure, Aneka threads exhibit more states than local threads because Aneka threads support file staging and they are scheduled by the middleware, which can queue them for a considerable amount of time. As Aneka supports the reservation of nodes for execution of thread related to a specific application, an explicit state indicating execution failure due to missing reservation credential has been introduced. This occurs when a thread is sent to an execution node in a time window where only nodes with specific reservation credentials can be executed.

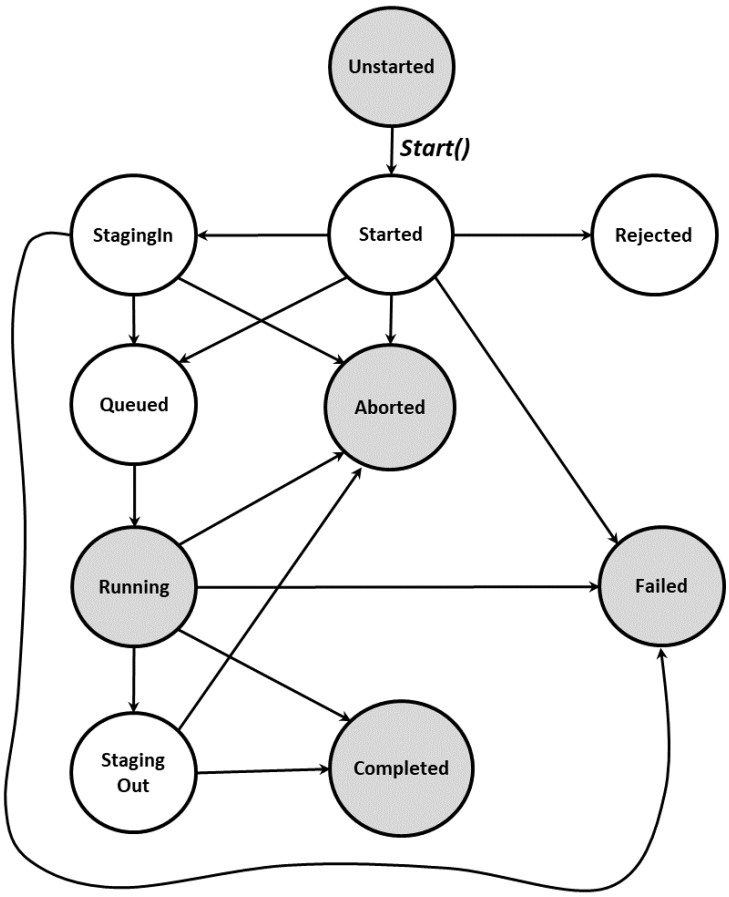
An Aneka thread is initially found in the Unstarted state. Once the Start() method is called, the thread transits to the Started state, from which it is possible to move to the StagingIn state if there are files to upload for its execution or directly to the Queued state. If there is any error while uploading files, the thread fails and it ends its execution with the Failed state, which can also be reached for any exception that occurred while invoking Start().

Another outcome might be the Rejected state that occurs if the thread is started with an invalid reservation token. This is a final state and implies execution failure due to lack of rights. Once the thread is in the queue, if there is a free node where to execute it, the middleware moves all the object data and depending files to the remote node and starts its execution, thus changing the state into Running. If the thread generates an exception or does not produce the expected output files, the execution is considered failed and the final state of the thread is set to Failed. If the execution is successful, the final state is set to Completed. If there are output files to retrieve, the thread state is set to StagingOut while files are collected and sent to their final destination, and then it transits to Completed. At any point, if the developer stops the execution of the application or directly calls the Abort() method, the thread is aborted and its final state is set to Aborted.

In most cases, the normal state transition will resemble the one occurring for local threads: Unstarted-[Started]-[Queued]-Running-Completed/Aborted/Failed



a. System.Threading.Thread life cycle.



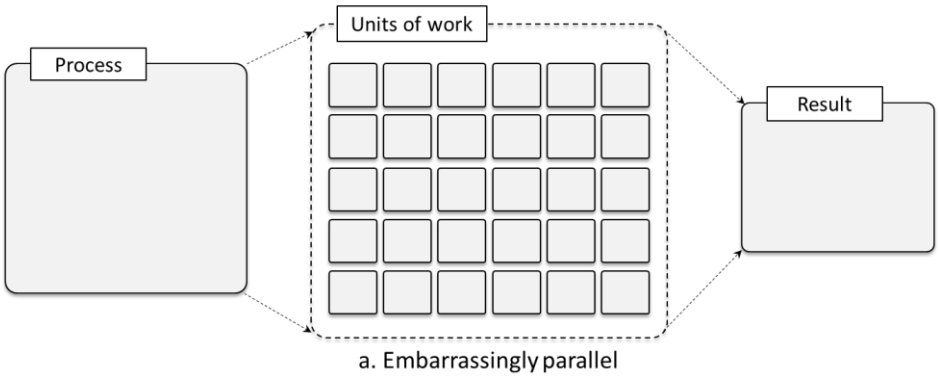
b. Aneka.Threading.AnekaThread life cycle.

Distinction based on Thread synchronization

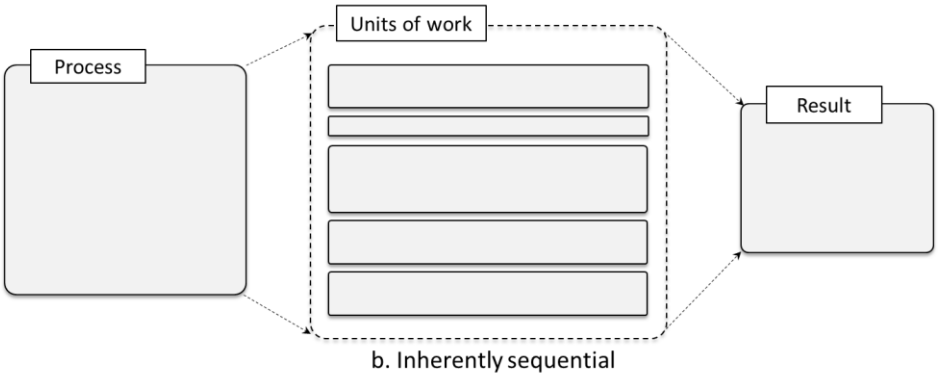
The .NET base class libraries provide advanced facilities to support thread synchronization by the means of monitors, semaphores, reader-writer locks, and basic synchronization constructs at the language level. Aneka provides minimal support for thread synchronization that is limited to the implementation of the join operation for thread abstraction. Most of the constructs and classes that are provided by the .NET framework are used to provide controlled access to shared data from different threads in order to preserve their integrity. This requirement is less stringent in a distributed environment, where there is no shared memory among the thread instances and therefore it is not necessary. Moreover, the reason for porting a local multithread application to Aneka threads implicitly involves the need for a distributed facility in which to execute a large number of threads, which might not be executing all at the same time. Providing coordination facilities that introduce a locking strategy in such an environment might lead to distributed deadlocks that are hard to detect. Therefore, by design Aneka threads do not feature any synchronization facility that goes beyond the simple join operation between executing threads

	Distinction based on 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. For interface compatibility purposes the Aneka.Threading.Thread class exhibits a Priority property whose type is ThreadPriority, but its value is always set to Normal, and changes to it do not produce any effect on thread scheduling by the Aneka middleware **Distinction based on Type serialization** Aneka threads execute in a distributed environment in which the object code in the form of libraries and live instances information are moved over the network. This condition imposes some limitations that are mostly concerned with the serialization of types in the .NET framework. Local threads execute all within the same address space and share memory; therefore, they do not need objects to be copied or transferred into a different address space. Aneka threads are distributed and execute on remote computing nodes, and this implies that the object code related to the method to be executed within a thread needs to be transferred over the network. Since delegates can point to instance methods, the state of the enclosing instance needs to be transferred and reconstructed on the remote execution environment. This is a particular feature at the class level and goes by the term type serialization. A .NET type is considered serializable if it is possible to convert an instance of the type into a binary array containing all the information required to revert it to its original form or into a possibly different execution context. This property is generally given for several types defined in the .NET framework by simply tagging the class definition with the Serializable attribute. If the class exposes a specific set of characteristics, the framework will automatically provide facilities to serialize and deserialize instances of that type. Alternatively, custom serialization can be implemented for any user-defined type. Aneka threads execute methods defined in serializable types, since it is necessary to move the enclosing instance to remote execution method. In most cases, providing serialization is as easy as tagging the class definition with the Serializable attribute; in other cases, it might be necessary to implement the ISerializable interface and provide appropriate constructors for the type. This is not a strong limitation, since there are very few cases in which types cannot be defined as serializable. For example, local threads, network connections, and streams are not serializable since they directly access local resources that cannot be implicitly moved onto a different node.	05	CO1	L1

4(a)	Distinguish between domain and functional decomposition techniques with illustrative examples **Answer:** **Domain Decomposition** Domain decomposition is the process of identifying patterns of functionally repetitive, but independent, computation on data. This is the most common type of decomposition in the case of throughput computing, and it relates to the identification of repetitive calculations required for solving a problem. When these calculations are identical, only differ from the data they operate on, and can be executed in any order, the problem is said to be embarrassingly parallel. Embarrassingly parallel problems constitute the easiest case for parallelization because there is no need to synchronize different threads that do not share any data. Moreover, coordination and communication between threads are minimal; this strongly simplifies the code logic and allows a high computing throughput. In many cases it is possible to devise a general structure for solving such problems and, in general, problems that can be parallelized through domain decomposition. 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. Although the complexity of the repetitive computation strictly depends on the nature of the problem, the coordination and decomposition logic is often quite simple and involves identifying the appropriate number of units of work to create. In general, a while or a for loop is used to express the decomposition logic, and each iteration generates a new unit of work to be assigned to a slave thread. An optimization, of this process involves the use of thread pooling to limit the number of threads used to execute repetitive computations. Several practical problems fall into this category; in the case of embarrassingly parallel problems, we can mention: • Geometrical transformation of two (or higher) dimensional data sets • Independent and repetitive computations over a domain such as Mandelbrot set and Monte Carlo computations Even though embarrassingly parallel problems are quite common, they are based on the strong assumption that at each of the iterations of the decomposition method, it is possible to isolate an independent unit of work. This is what makes it possible to obtain a high computing throughput. Such a condition is not met if the values of all the iterations are dependent on some of the values obtained in the previous iterations. In this case, the problem is said to be inherently sequential, and it is not possible to directly apply the methodology described previously. Despite this, it can still be possible to break down the whole computation into a set of independent units of work, which might have a different granularity—for example, by grouping into single computation-dependent iterations. Figure below provides a schematic representation of the decomposition of embarrassingly parallel and inherently sequential problems	10	CO3	L2

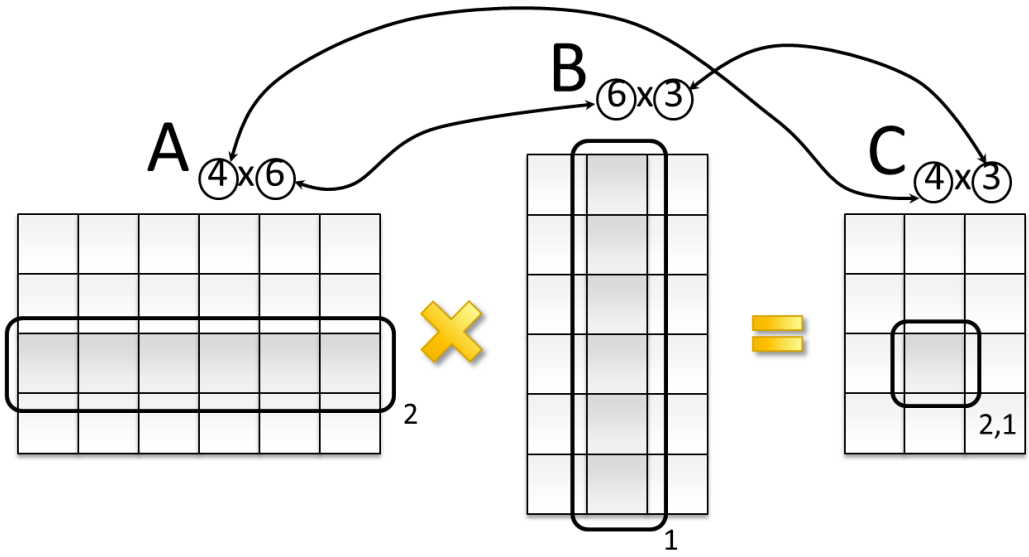


a. Embarrassingly parallel



b. Inherently sequential

To show how domain decomposition can be applied, it is possible to create a simple program that performs matrix multiplication using multiple threads. Matrix multiplication is a binary operation that takes two matrices and produces another matrix as a result. This is obtained as a result of the composition of the linear transformation of the original matrices. There are several techniques for performing matrix multiplication; among them, the matrix product is the most popular. Figure below provides an overview of how a matrix product can be performed.



The matrix product computes each element of the resulting matrix as a linear combination of the corresponding row and column of the first and second input matrices, respectively. The formula that applies for each of the resulting matrix elements is the following:

$$C_{ij} = \sum_{k=0}^{n-1} A_{ik} B_{kj}$$

Therefore, two conditions hold in order to perform a matrix product:

- Input matrices must contain values of a comparable nature for which the scalar product is defined.
- The number of columns in the first matrix must match the number of rows of the second matrix.

Given these conditions, the resulting matrix will have the number of rows of the first matrix and the number of columns of the second matrix, and each element will be computed as described by the preceding equation.

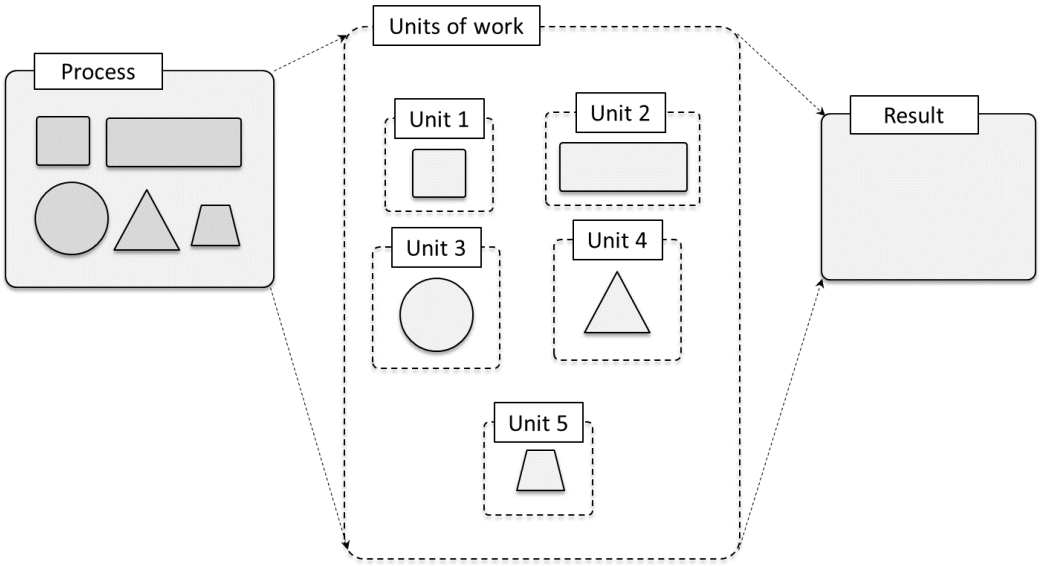
It is evident that the repetitive operation is the computation of each of the elements of the resulting matrix. These are subject to the same formula, and the computation does not depend on values that have been obtained by the computation of other elements of the resulting matrix. Hence, the problem is embarrassingly parallel, and we can logically organize the multithreaded program in the following steps:

- Define a function that performs the computation of the single element of the resulting matrix by implementing the previous equation.
- Create a double for loop (the first index iterates over the rows of the first matrix and the second over the columns of the second matrix) that spawns a thread to compute the elements of the resulting matrix.
- Join all the threads for completion and compose the resulting matrix.

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 many threads, since the different computations that are performed by a single program are limited.

Functional decomposition leads to a natural decomposition of the problem in separate units of work because it does not involve partitioning the dataset, but the separation among them is clearly defined by distinct logic operations. Figure below provides a pictorial view of how decomposition operates and allows parallelization.

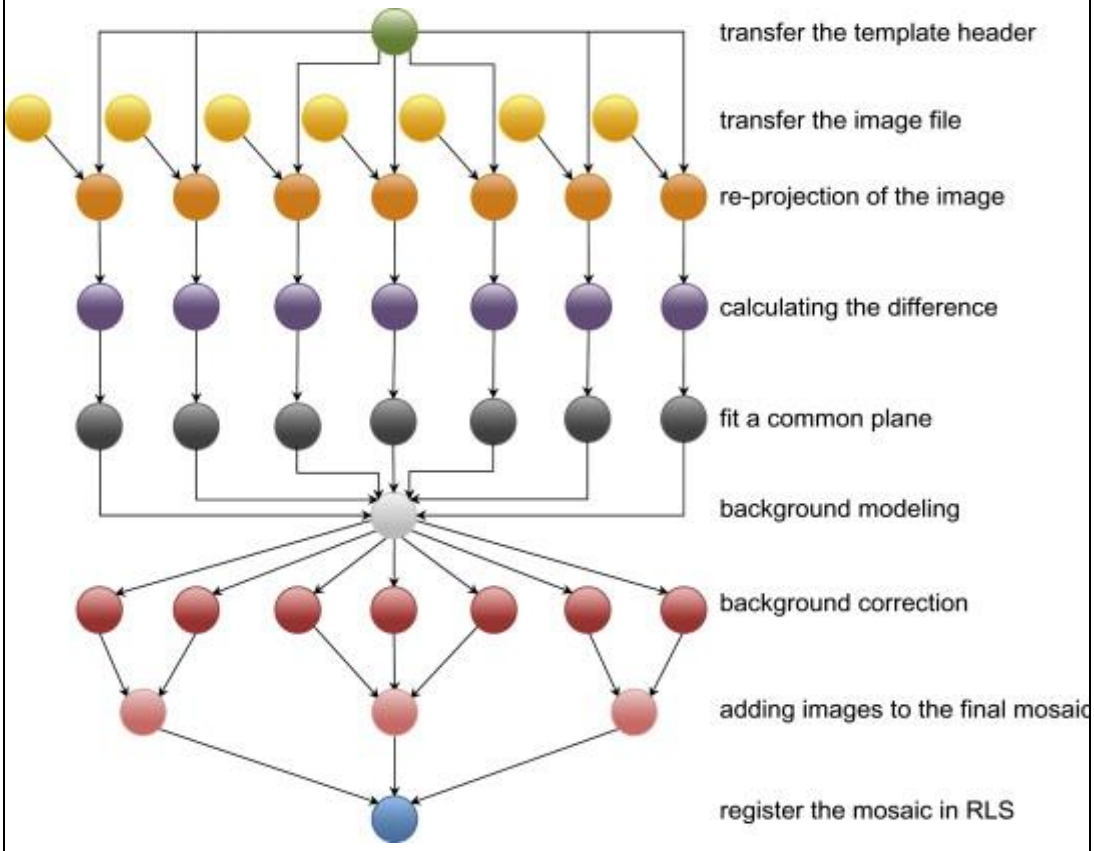


As described by the schematic in the Figure, problems that are subject to functional decomposition can also require a composition phase in which the outcomes of each of the independent units of work are composed together. In the case of domain decomposition, this phase often results in an aggregation process. The way in which results are composed in this case strongly depends on the type of operations that define the problem.

In the following, we show a very simple example of how a mathematical problem can be parallelized using functional decomposition. Suppose, for example, that we need to calculate the value of the following function for a given value of x:

$$\boldsymbol{f(x) = six(x) + cos(x) + tan(x)}$$

	It is apparent that, once the value of x has been set, the three different operations can be performed independently of each other. This is an example of functional decomposition because the entire problem can be separated into three distinct operations. The program computes the sine, cosine, and tangent functions in three separate threads and then aggregates the results.			
5(a)	Explain work flow with practical example **Answer:** Workflow applications are characterized by a collection of tasks that exhibit dependencies among them. Such dependencies, which are mostly data dependencies (i.e., the output of one task is a prerequisite of another task), determine the way in which the applications are scheduled as well as where they are scheduled. Concerns in this case are related to providing a feasible sequencing of tasks and to optimizing the placement of tasks so that the movement of data is minimized. The term workflow has a long tradition in the business community, where the term is used to describe a composition of services that all together accomplish a business process. As defined by the Workflow Management Coalition, a workflow is the automation of a business process, in whole or part, during which documents, information, or tasks are passed from one participant (a resource; human or machine) to another for action, according to a set of procedural rules. The concept of workflow as a structured execution of tasks that have dependencies on each other has demonstrated itself to be useful for expressing many scientific experiments and gave birth to the idea of scientific workflow. Many scientific experiments are a combination of problem-solving components, which, connected in a order, define the specific nature of the experiment. When such experiments exhibit a natural parallelism and need to execute a large number of operations or deal with huge quantities of data, it makes sense to execute them on a distributed infrastructure. In the case of scientific workflows, the process is identified by an application to run, the elements that are passed among participants are mostly tasks and data, and the participants are mostly computing or storage nodes. The set of procedural rules is defined by a workflow definition scheme that guides the scheduling of the application. A scientific workflow generally involves data management, analysis, simulation, and middleware supporting the execution of the workflow. A scientific workflow is generally expressed by a directed acyclic graph (DAG), which defines the dependencies among tasks or operations. The nodes on the DAG represent the tasks to be executed in a workflow application; the arcs connecting the nodes identify the dependencies among tasks and the data paths that connect the tasks. The most common dependency that is realized through a DAG is data dependency, which means that the output files of a task (or some of them) constitute the input files of another task. This dependency is represented as an arc originating from the node that identifies the first task and terminating in the node that identifies the second task. **Example** The DAG in Figure below describes a sample Montage workflow. Montage is a toolkit for assembling images into mosaics; it has been specially designed to support astronomers in composing the images taken from different telescopes or points of view into a coherent image. The toolkit provides several applications for manipulating images and composing them together; some of the applications perform background reprojection, perspective transformation, and brightness and color correction. The workflow depicted here describes the general process for composing a mosaic; the labels on the right describe the different tasks that have to be performed to compose a mosaic. In the case presented in the diagram, a mosaic is composed of seven images. The entire process can take advantage of a distributed infrastructure for its execution, since there are several operations that can be performed in parallel. For each of the image files, the following process has to be performed: image file transfer, reprojection, calculation of the difference, and common plane placement. Therefore, each of the images can be processed in parallel for these tasks. Here is where a distributed infrastructure helps in executing workflows. There might be another reason for executing workflows on a distributed infrastructure: It might be convenient to move the computation on a specific node because of data locality issues. For example, if an operation needs to access specific resources that are only	[06]	CO2	L2

	available on a specific node, that operation cannot be performed elsewhere, whereas the rest of the operations might not have the same requirements. A scientific experiment might involve the use of several problem solving components that might require the use of specific instrumentation; in this case all the tasks that have these constraints need to be executed where the instrumentation is available, thus creating a distributed execution of a process that is not parallel in principle.  			
5(b)	**Describe two work flow technologies** **Answer: Any two from** **Kepler** is an open-source scientific workflow engine built from the collaboration of several research projects. The system is based on the Ptolemy II system, which provides a solid platform for developing dataflow-oriented workflows. Kepler provides a design environment based on the concept of actors, which are reusable and independent blocks of computation such as Web services, database calls, and the like. The connection between actors is made with ports. An actor consumes data from the input ports and writes data/results to the output ports. The novelty of Kepler is in its ability to separate the flow of data among components from the coordination logic that is used to execute workflow. Thus, for the same workflow, Kepler supports different models, such as synchronous and asynchronous models. The workflow specification is expressed using a proprietary XML language. **DAGMan (Directed Acyclic Graph Manager)**, part of the Condor project, constitutes an extension to the Condor scheduler to handle job interdependencies. Condor finds machines for the execution of programs but does not support the scheduling of jobs in a specific sequence. Therefore, DAGMan acts as a metascheduler for Condor by submitting the jobs to the scheduler in the appropriate order. The input of DAGMan is a simple text file that contains the information about the jobs, pointers to their job submission files, and the dependencies among jobs. **Cloudbus Workflow Management System (WfMS)** is a middleware platform built for managing large application workflows on distributed computing platforms such as grids and clouds. It comprises software tools that help end users compose, schedule, execute, and monitor workflow applications through a Web-based portal.	04	CO1	L1

	The portal provides the capability of uploading workflows or defining new ones with a graphical editor. To execute workflows, WfMS relies on the Gridbus Broker, a grid/cloud resource broker that supports the execution of applications with quality-ofservice (QoS) attributes over a heterogeneous distributed computing infrastructure, including Linux-based clusters, Globus, and Amazon EC2. WfMS uses a proprietary XML language for the specification of workflows. **Offspring** has a different perspective, which offers a programming-based approach to developing workflows. Users can develop strategies and plug them into the environment, which will execute them by leveraging a specific distribution engine. The advantage provided by Offspring over other solutions is the ability to define dynamic workflows. This strategy represents a semi-structured workflow that can change its behavior at runtime according to the execution of specific tasks. This allows developers to dynamically control the dependencies of tasks at runtime rather than statically defining them. Offspring supports integration with any distributed computing middleware that can manage a simple bag-of-tasks application. It provides a native integration with Aneka and supports a simulated distribution engine for testing strategies during development. Because Offspring allows the definition of workflows in the form of plug-ins, it does not use any XML specification.			
6(a)	Explain the importance of computation and communication with respect to the design of parallel and distributed applications. **Answer:** In designing parallel and in general distributed applications, it is very important to carefully evaluate the communication patterns among the components that have been identified during problem decomposition. The two decomposition methods presented in this section and the corresponding sample applications are based on the assumption that the computations are independent. This means that: • The input values required by one computation do not depend on the output values generated by another computation. • The different units of work generated as a result of the decomposition do not need to interact (i.e., exchange data) with each other. These two assumptions strongly simplify the implementation and allow achieving a high degree of parallelism and a high throughput. Having all the worker threads independent from each other gives the maximum freedom to the operating system (or the virtual runtime environment) scheduler in scheduling all the threads. The need to exchange data among different threads introduces dependencies among them and ultimately can result in introducing performance bottlenecks. For example, we did not introduce any queuing technique for threads; but queuing threads might potentially constitute a problem for the execution of the application if data need to be exchanged with some threads that are still in the queue. A more common disadvantage is the fact that while a thread exchanges data with another one, it uses synchronization strategy that might lead to blocking the execution of other threads. The more data that need to be exchanged, the more they block threads for synchronization, thus ultimately impacting the overall throughput. As a general rule of thumb, it is important to minimize the amount of data that needs to be exchanged while implementing parallel and distributed applications. The lack of communication among different threads constitutes the condition leading to the highest throughput.	06	CO3	L3
6(b)	**Discuss about POSIX threads** **Answer:** **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. Standard POSIX 1.c (IEEE Std 1003.1c-1995) addresses the implementation of threads and the functionalities that	04	CO3	L3

	should be available for application programmers to develop portable multithreaded applications. The standards address the Unix-based operating systems, but an implementation of the same specification has been provided for Windows-based systems. The POSIX standard defines the following operations: creation of threads with attributes, termination of a thread, and waiting for thread completion (join operation). In addition to the logical structure of a thread, other abstractions, such as semaphores, conditions, reader-writer locks, and others, are introduced in order to support proper synchronization among threads. 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. • Different synchronization abstractions are provided to solve different synchronization problems. • A default implementation of the POSIX 1.c specification has been provided for the C language. All the available functions and data structures are exposed in the pthread.h header file, which is part of the standard C implementations.			
7(a)	Describe the different task-based application models **Answer:** There are several models based on the concept of the task as the fundamental unit for composing distributed applications. What makes these models different from one another is the way in which tasks are generated, the relationships they have with each other, and the presence of dependencies or other conditions—for example, a specific set of services in the runtime environment—that must be met. In this section, we quickly review the most common and popular models based on the concept of the task. **Embarrassingly parallel applications** Embarrassingly parallel applications constitute the most simple and intuitive category of distributed applications. As we discussed in Chapter 6, embarrassingly parallel applications constitute 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 types, and they do not need to communicate among themselves. This category of applications is supported by most of the frameworks for distributed computing. Since tasks do not need to communicate, there is a lot of freedom regarding the way 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 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.	[06]	CO1	L1

There are several problems that can be modeled as embarrassingly parallel. These include image and video rendering, evolutionary optimization, and model forecasting. In image and video rendering the task is represented by the rendering of a pixel (more likely a portion of the image) or a frame, respectively. For evolutionary optimization metaheuristics, a task is identified by a single run of the algorithm with a given parameter set. The same applies to model forecasting applications. In general, scientific applications constitute a considerable source of embarrassingly parallel applications, even though they mostly fall into the more specific category of parameter sweep applications.

Parameter sweep applications are a specific class of embarrassingly parallel applications for which the tasks are identical in their nature and differ only by the specific parameters used to execute them. Parameter sweep applications are identified by a template task and a set of parameters. The template task defines the operations that will be performed on the remote node for the execution of tasks. The template task is parametric, and the parameter set identifies the combination of variables whose assignments specialize the template task into a specific instance. The combination of parameters, together with their range of admissible values, identifies the multidimensional domain of the application, and each point in this domain identifies a task instance.

Any distributed computing framework that provides support for embarrassingly parallel applications can also support the execution of parameter sweep applications, since the tasks composing the application can be executed independently of each other. The only difference is that the tasks that will be executed are generated by iterating over all the possible and admissible combinations of parameters. This operation can be performed by frameworks natively or tools that are part of the distributed computing middleware. For example, Nimrod/G is natively designed to support the execution of parameter sweep applications, and Aneka provides client-based tools for visually composing a template task, defining parameters, and iterating over all the possible combinations of such parameters.

A plethora of applications fall into this category. Mostly they come from the scientific computing domain: evolutionary optimization algorithms, weather-forecasting models, computational fluid dynamics applications, Monte Carlo methods, and many others. For example, in the case of evolutionary algorithms it is possible to identify the domain of the applications as a combination of the relevant parameters of the algorithm. For genetic algorithms these might be the number of individuals of the population used by the optimizer and the number of generations for which to run the optimizer.

MPI applications

Message Passing Interface (MPI) is a specification for developing parallel programs that communicate by exchanging messages. Compared to earlier models, MPI introduces the constraint of communication that involves MPI tasks that need to run at the same time. MPI has originated as an attempt to create common ground from the several distributed shared memory and message-passing infrastructures available for distributed computing. Nowadays, MPI has become a de facto standard for developing portable and efficient message-passing HPC applications. Interface specifications have been defined and implemented for C/C11 and Fortran.

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

7(b)	What is data- intensive computing? Describe the open challenges in data-intensive computing **Answer:** Data-intensive computing is concerned with production, manipulation, and analysis of large-scale data in the range of hundreds of megabytes (MB) to petabytes (PB) and beyond. The term dataset is commonly used to identify a collection of information elements that is relevant to one or more applications. Datasets are often maintained in repositories, which are infrastructures supporting the storage, retrieval, and indexing of large amounts of information. To facilitate the classification and search, relevant bits of information, called metadata, are attached to datasets. Data-intensive computations occur in many application domains. Computational science is one of the most popular ones. People conducting scientific simulations and experiments are often keen to produce, analyze, and process huge volumes of data. Hundreds of gigabytes of data are produced every second by telescopes mapping the sky; the collection of images of the sky easily reaches the scale of petabytes over a year. Bioinformatics applications mine databases that may end up containing terabytes of data. Earthquake simulators process a massive amount of data, which is produced as a result of recording the vibrations of the Earth across the entire globe. Data-intensive applications not only deal with huge volumes of data but, very often, also exhibit compute-intensive properties. Data-intensive applications handle datasets on the scale of multiple terabytes and petabytes. Datasets are commonly persisted in several formats and distributed across different locations. Such applications process data in multistep analytical pipelines, including transformation and fusion stages. The processing requirements scale almost linearly with the data size, and they can be easily processed in parallel. They also need efficient mechanisms for data management, filtering and fusion, and efficient querying and distribution. **Challenges:** 1. Scalable algorithms that can search and process massive datasets 2. New metadata management technologies that can scale to handle complex, heterogeneous, and distributed data sources 3. Advances in high-performance computing platforms aimed at providing a better support for accessing in-memory multiterabyte data structures 4. High-performance, highly reliable, petascale distributed file systems 5. Data signature-generation techniques for data reduction and rapid processing 6. New approaches to software mobility for delivering algorithms that are able to move the computation to where the data are located 7. Specialized hybrid interconnection architectures that provide better support for filtering multigigabyte datastreams coming from high-speed networks and scientific instruments 8. Flexible and high-performance software integration techniques that facilitate the combination of software modules running on different platforms to quickly form analytical pipelines	[04]	CO3	L2