CELEBRATING 25 YEARS CMRIT CMR INSTITUTE OF TECHNOLOGY, BENGALURU. ACCREDITED WITH A+ GRADE BY NAAC

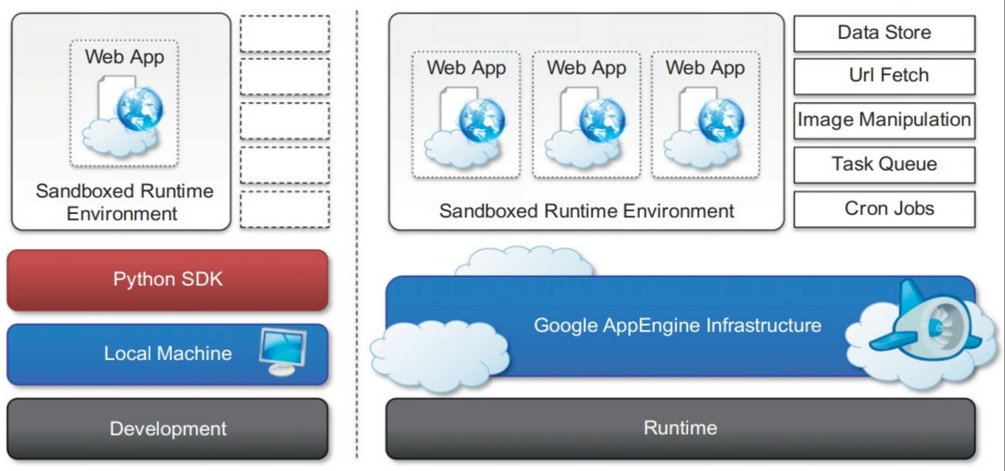
Internal Assessment Test 1 – March 2019										
Sub:	Cloud Computing and its Applications					Sub Code:	15CS742	Branch:	CSE	
Date:	12/10/19	Duration:	90 mins	Max Marks:	50	Sem / Sec:	VII A/B/C		OBE	
	Answer any FIVE FULL Questions							MARKS	CO	RBT

		MARKS	CO	RBT
1 (a)	Discuss six major categories of currently available configuration for EC2 instances **Answer:** 1. Standard instances. This class offers a set of configurations that are suitable for most applications. EC2 provides three different categories of increasing computing power, storage, and memory. 2. Micro instances. This class is suitable for those applications that consume a limited amount of computing power and memory and occasionally need bursts in CPU cycles to process surges in the workload. Micro instances can be used for small Web applications with limited traffic. 3. High-memory instances. This class targets applications that need to process huge workloads and require large amounts of memory. Three-tier Web applications characterized by high traffic are the target profile. Three categories of increasing memory and CPU are available, with memory proportionally larger than computing power. 4. High-CPU instances. This class targets compute-intensive applications. Two configurations are available where computing power proportionally increases more than memory. 5. Cluster Compute instances. This class is used to provide virtual cluster services. Instances in this category are characterized by high CPU compute power and large memory and an extremely high I/O and network performance, which makes it suitable for HPC applications. 6. Cluster GPU instances. This class provides instances featuring graphic processing units (GPUs) and high compute power, large memory, and extremely high I/O and network performance. This class is particularly suited for cluster applications that perform heavy graphic computations, such as rendering clusters. Since GPU can be used for general-purpose computing, users of such instances can benefit from additional computing power, which makes this class suitable for HPC applications.	[06]	CO3	L2
1(b)	Describe Amazon simple DB **Answer:** Amazon SimpleDB is a lightweight, highly scalable, and flexible data storage solution for applications that do not require a fully relational model for their data. SimpleDB provides support for semistructured data, the model for which is based on the concept of domains, items, and attributes. With respect to the relational model, this model provides fewer constraints on the structure of data entries, thus obtaining improved performance in querying large quantities of data. As happens for Amazon RDS, this service frees AWS users from performing configuration, management, and high-availability design for their data stores. SimpleDB uses domains as top-level elements to organize a data store. These domains are roughly comparable to tables in the relational model. Unlike tables, they allow items not to have all the same column structure; each item is therefore represented as a collection of attributes expressed in the form of a key-value pair. Each domain can grow up to 10 GB of data, and by default a single user can allocate a maximum of 250 domains. Clients can create, delete, modify, and make snapshots of domains. They can insert, modify, delete, and query items and attributes. Batch insertion and deletion are also supported. The capability of querying data is one of the most relevant functions of the model, and the select clause supports the following test operators: =, != ,< ,> ,<= ,>= , *like, not like, between, is null, is not null, and every()*. Here is a simple example on how to query data: *select * from domain_name where every(attribute_name) = ‘value’* Moreover, the select operator can extend its query beyond the boundaries of a single domain, thus allowing users to query effectively a large amount of data To efficiently provide AWS users with a scalable and fault-tolerant service, SimpleDB implements a relaxed constraint model, which leads to eventually consistent data. The adverb eventually denotes the fact that multiple accesses on the same data might not read the same value in the very short term, but they will eventually converge over time. This is because SimpleDB does not lock all the copies of the data during an update, which is propagated in the	[04]	CO3	L2

	background. Therefore, there is a transient period of time in which different clients can access different copies of the same data that have different values. This approach is very scalable with minor drawbacks, and it is also reasonable, since the application scenario for SimpleDB is mostly characterized by querying and indexing operations on data. Alternatively, it is possible to change the default behavior and ensure that all the readers are blocked during an update.			
2(a)	Summarize Access Control Policies (ACP) and permissions with respect to Amazon S3 **Answer:** Amazon S3 allows controlling the access to buckets and objects by means of Access Control Policies (ACPs). An ACP is a set of grant permissions that are attached to a resource expressed by means of an XML configuration file. A policy allows defining up to 100 access rules, each of them granting one of the available permissions to a grantee. Currently, five different permissions can be used: • READ allows the grantee to retrieve an object and its metadata and to list the content of a bucket as well as getting its metadata. • WRITE allows the grantee to add an object to a bucket as well as modify and remove it. • READ_ACP allows the grantee to read the ACP of a resource. • WRITE_ACP allows the grantee to modify the ACP of a resource • FULL_CONTROL grants all of the preceding permissions Grantees can be either single users or groups. Users can be identified by their canonical IDs or the email addresses they provided when they signed up for S3. For groups, only three options are available: all users, authenticated users, and log delivery users. ACPs provide a set of powerful rules to control S3 users' access to resources, but they do not exhibit fine grain in the case of non-authenticated users, who cannot be differentiated and are considered as a group. To provide a finer grain in this scenario, S3 allows defining signed URIs, which grant access to a resource for a limited amount of time to all the requests that can provide a temporary access token. **Advanced features** Besides the management of buckets, objects, and ACPs, S3 offers other additional features that can be helpful. These features are server access logging and integration with the BitTorrent file-sharing network. Server access logging allows bucket owners to obtain detailed information about the request made for the bucket and all the objects it contains. By default, this feature is turned off; it can be activated by issuing a PUT request to the bucket URI followed by *?logging*. The request should include an XML file specifying the target bucket in which to save the logging files and the file name prefix. A GET request to the same URI allows the user to retrieve the existing logging configuration for the bucket. The second feature of interest is represented by the capability of exposing S3 objects to the BitTorrent network, thus allowing files stored in S3 to be downloaded using the BitTorrent protocol. This is done by appending *?torrent* to the URI of the S3 object. To actually download the object, its ACP must grant read permission to everyone	[06]	CO2	L2
2(b)	Discuss the variations and extensions of MapReduce **Answer:**	[04]	CO3	L2
3(a)	Explain various types of NoSQL systems with examples. How NoSQL helps in data intensive computing in distributed systems? **Answer:**			
3(b)	Demonstrate the role of cloud technologies in social networking **Answer:** Social networking applications have grown considerably in the last few years to become the most active sites on the Web. To sustain their traffic and serve millions of users seamlessly, services such as Twitter and Facebook have leveraged cloud computing technologies. The possibility of continuously adding capacity while systems are running is the most attractive feature for social networks, which constantly increase their user base. **Facebook** Facebook is probably the most evident and interesting environment in social networking. With more than 800 million users, it has become one of the largest Websites in the world. To sustain this incredible growth, it has been fundamental that Facebook be capable of continuously adding capacity and developing new scalable	[04]	CO4	L3

	technologies and software systems while maintaining high performance to ensure a smooth user experience. Currently, the social network is backed by two data centers that have been built and optimized to reduce costs and impact on the environment. On top of this highly efficient infrastructure, built and designed out of inexpensive hardware, a completely customized stack of opportunely modified and refined open-source technologies constitutes the back-end of the largest social network. Taken all together, these technologies constitute a powerful platform for developing cloud applications. This platform primarily supports Facebook itself and offers APIs to integrate third-party applications with Facebook's core infrastructure to deliver additional services such as social games and quizzes created by others. The reference stack serving Facebook is based on LAMP (Linux, Apache , MySQL, and PHP). This collection of technologies is accompanied by a collection of other services developed in-house. These services are developed in a variety of languages and implement specific functionalities such as search, news feeds, notifications, and others. While serving page requests, the social graph of the user is composed. The social graph identifies a collection of interlinked information that is of relevance for a given user. Most of the user data are served by querying a distributed cluster of MySQL instances, which mostly contain key-value pairs. These data are then cached for faster retrieval. The rest of the relevant information is then composed together using the services mentioned before. These services are located closer to the data and developed in languages that provide better performance than PHP. The development of services is facilitated by a set of internally developed tools. One of the core elements is Thrift. This is a collection of abstractions (and language bindings) that allow cross-language development. Thrift allows services developed in different languages to communicate and exchange data. Bindings for Thrift in different languages take care of data serialization and deserialization, communication, and client and server boilerplate code. This simplifies the work of the developers, who can quickly prototype services and leverage existing ones. Other relevant services and tools are Scribe, which aggregates streaming log feeds, and applications for alerting and monitoring.			
4(a)	Describe the usage of cloud computing in remote ECG monitoring **Answer:** Healthcare is a domain in which computer technology has found several and diverse applications: from supporting the business functions to assisting scientists in developing solutions to cure diseases. An important application is the use of cloud technologies to support doctors in providing more effective diagnostic processes. In particular, here we discuss electrocardiogram (ECG) data analysis on the cloud. ECG is the electrical manifestation of the contractile activity of the heart's myocardium. This activity produces a specific waveform that is repeated over time and that represents the heartbeat. The analysis of the shape of the ECG waveform is used to identify arrhythmias and is the most common way to detect heart disease. Cloud computing technologies allow the remote monitoring of a patient's heartbeat data, data analysis in minimal time, and the notification of first-aid personnel and doctors should these data reveal potentially dangerous conditions. This way a patient at risk can be constantly monitored without going to a hospital for ECG analysis. At the same time, doctors and first-aid personnel can instantly be notified of cases that require their attention. Wearable computing devices equipped with ECG sensors constantly monitor the patient's heartbeat. Such information is transmitted to the patient's mobile device, which will eventually forward it to the cloud-hosted Web service for analysis. The Web service forms the front-end of a platform that is entirely hosted in the cloud and that leverages the three layers of the cloud computing stack: SaaS, PaaS, and IaaS. The Web service constitute the SaaS application that will store ECG data in the Amazon S3 service and issue a processing request to the scalable cloud platform. The runtime platform is composed of a dynamically sizable number of instances running the workflow engine and Aneka. The number of workflow engine instances is controlled according to the number of requests in the queue of each instance, while Aneka controls the number of EC2 instances used to execute the single tasks defined by the workflow engine for a single ECG processing job. Each of these jobs	[05]	CO4	L2

	consists of a set of operations involving the extraction of the waveform from the heartbeat data and the comparison of the waveform with a reference waveform to detect anomalies. If anomalies are found, doctors and first-aid personnel can be notified to act on a specific patient. Even though remote ECG monitoring does not necessarily require cloud technologies, cloud computing introduces opportunities that would be otherwise hardly achievable. The first advantage is the elasticity of the cloud infrastructure that can grow and shrink according to the requests served. As a result, doctors and hospitals do not have to invest in large computing infrastructures designed after capacity planning, thus making more effective use of budgets. The second advantage is ubiquity. Cloud computing technologies have now become easily accessible and promise to deliver systems with minimum or no downtime. Computing systems hosted in the cloud are accessible from any Internet device through simple interfaces (such as SOAP and REST-based Web ser-vices). This makes these systems not only ubiquitous, but they can also be easily integrated with other systems maintained on the hospital's premises. Finally, cost savings constitute another reason for the use of cloud technology in healthcare. Cloud services are priced on a pay-per-use basis and with volume prices for large numbers of service requests. These two models provide a set of flexible options that can be used to price the service, thus actually charging costs based on effective use rather than capital costs.			
4(b)	Demonstrate the relevance of cloud computing in Biology with practical examples **Answer:** Applications in biology often require high computing capabilities and often operate on large datasets that cause extensive I/O operations. Because of these requirements, biology applications have often made extensive use of supercomputing and cluster computing infrastructures. Similar capabilities can be leveraged on demand using cloud computing technologies in a more dynamic fashion, thus opening new opportunities for bioinformatics applications. **Protein Structure Prediction** Protein structure prediction is a computationally intensive task that is fundamental to different types of research in the life sciences. Among these is the design of new drugs for the treatment of diseases. The geometric structure of a protein cannot be directly inferred from the sequence of genes that compose its structure, but it is the result of complex computations aimed at identifying the structure that minimizes the required energy. This task requires the investigation of a space with a massive number of states, consequently creating a large number of computations for each of these states. The computational power required for protein structure prediction can now be acquired on demand, without owning a cluster or navigating the bureaucracy to get access to parallel and distributed computing facilities. Cloud computing grants access to such capacity on a pay-per-use basis. One project that investigates the use of cloud technologies for protein structure prediction is Jeeva-an integrated Web portal that enables scientists to offload the prediction task to a computing cloud based on Aneka. The prediction task uses machine learning techniques (support vector machines) for determining the secondary structure of proteins. These techniques translate the problem into one of pattern recognition, where a sequence has to be classified into one of three possible classes (E, H, and C). A popular implementation based on support vector machines divides the pattern recognition problem into three phases: initialization, classification, and a final phase . Even though these three phases have to be executed in sequence, it is possible to take advantage of parallel execution in the classification phase, where multiple classifiers are executed concurrently. This creates the opportunity to sensibly reduce the computational time of the prediction. The prediction algorithm is then translated into a task graph that is submitted to Aneka. Once the task is completed, the middleware makes the results available for visualization through the portal. The advantage of using cloud technologies (i.e., Aneka as scalable cloud middleware) versus conventional grid infrastructures is the capability to leverage a scalable computing infrastructure that can be grown and shrunk on demand. This concept is distinctive of cloud technologies and constitutes a strategic advantage when applications are offered and delivered as a service. **Gene expression data analysis for cancer diagnosis**	[05]	CO4	L3

	Gene expression profiling is the measurement of the expression levels of thousands of genes at once. It is used to understand the biological processes that are triggered by medical treatment at a cellular level. Together with protein structure prediction, this activity is a fundamental component of drug design, since it allows scientists to identify the effects of a specific treatment. Another important application of gene expression profiling is cancer diagnosis and treatment. Cancer is a disease characterized by uncontrolled cell growth and proliferation. This behavior occurs because genes regulating the cell growth mutate. This means that all the cancerous cells contain mutated genes. In this context, gene expression profiling is utilized to provide a more accurate classification of tumors. The classification of gene expression data samples into distinct classes is a challenging task. The dimensionality of typical gene expression datasets ranges from several thousands to over tens of thousands of genes. However, only small sample sizes are typically available for analysis. This problem is often approached with learning classifiers, which generate a population of condition-action rules that guide the classification process. Among these, the eXtended Classifier System (XCS) has been successfully utilized for classifying large datasets in the bioinformatics and computer science domains. However, the effectiveness of XCS, when confronted with high dimensional datasets (such as microarray gene expression data sets), has not been explored in detail. A variation of this algorithm, CoXCS, has proven to be effective in these conditions. CoXCS divides the entire search space into subdomains and employs the standard XCS algorithm in each of these subdomains. Such a process is computationally intensive but can be easily parallelized because the classifications problems on the subdomains can be solved concurrently. Cloud-CoXCS is a cloud-based implementation of CoXCS that leverages Aneka to solve the classification problems in parallel and compose their outcomes. The algorithm is controlled by strategies, which define the way the outcomes are composed together and whether the process needs to be iterated. Because of the dynamic nature of XCS, the number of required compute resources to execute it can vary over time. Therefore, the use of scalable middleware such as Aneka offers a distinctive advantage.			
5(a)	With a neat diagram, explain Google App Engine platform architecture  Web App Sandboxed Runtime Environment Python SDK Local Machine Development Web App Web App Web App Sandboxed Runtime Environment Data Store Url Fetch Image Manipulation Task Queue Cron Jobs Google AppEngine Infrastructure Runtime AppEngine is a platform for developing scalable applications accessible through the Web (see Figure). The platform is logically divided into four major components: infrastructure, the runtime environment, the underlying storage, and the set of scalable services that can be used to develop applications. **Infrastructure** AppEngine hosts Web applications, and its primary function is to serve users requests efficiently. To do so, AppEngine's infrastructure takes advantage of many servers available within Google datacenters. For each HTTP request, AppEngine locates the servers hosting the application that processes the request, evaluates their load, and, if necessary, allocates additional resources (i.e., servers) or redirects the request to an existing server. The particular design of applications, which does not expect any state information to be implicitly maintained between requests to the same application,	10	CO2	L2

simplifies the work of the infrastructure, which can redirect each of the requests to any of the servers hosting the target application or even allocate a new one.

Runtime environment

The runtime environment represents the execution context of applications hosted on AppEngine. With reference to the AppEngine infrastructure code, which is always active and running, the runtime comes into existence when the request handler starts executing and terminates once the handler has completed.

Sandboxing

One of the major responsibilities of the runtime environment is to provide the application environment with an isolated and protected context in which it can execute without causing a threat to the server and without being influenced by other applications. In other words, it provides applications with a sandbox.

Currently, AppEngine supports applications that are developed only with managed or interpreted languages, which by design require a runtime for translating their code into executable instructions. Therefore, sandboxing is achieved by means of modified runtimes for applications that disable some of the common features normally available with their default implementations. If an application tries to perform any operation that is considered potentially harmful, an exception is thrown and the execution is interrupted. Some of the operations that are not allowed in the sandbox include writing to the server's file system; accessing computer through network besides using Mail, UrlFetch, and XMPP ; executing code outside the scope of a request, a queued task, and a cron job; and processing a request for more than 30 seconds.

Supported runtimes

Currently, it is possible to develop AppEngine applications using three different languages and related technologies: Java, Python, and Go.

AppEngine currently supports Java 6, and developers can use the common tools for Web application development in Java, such as the Java Server Pages (JSP), and the applications interact with the environment by using the Java Servlet standard. Furthermore, access to AppEngine services is provided by means of Java libraries that expose specific interfaces of provider-specific implementations of a given abstraction layer. Developers can create applications with the AppEngine Java SDK, which allows developing applications with either Java 5 or Java 6 and by using any Java library that does not exceed the restrictions imposed by the sandbox.

Support for Python is provided by an optimized Python 2.5.2 interpreter. As with Java, the runtime environment supports the Python standard library, but some of the modules that implement potentially harmful operations have been removed, and attempts to import such modules or to call specific methods generate exceptions. To support application development, AppEngine offers a rich set of libraries connecting applications to AppEngine services. In addition, developers can use a specific Python Web application framework, called webapp, simplifying the development of Web applications.

The Go runtime environment allows applications developed with the Go programming language to be hosted and executed in AppEngine. Currently the release of Go that is supported by AppEngine is r58.1. The SDK includes the compiler and the standard libraries for developing applications in Go and interfacing it with AppEngine services. As with the Python environment, some of the functionalities have been removed or generate a runtime exception. In addition, developers can include third-party libraries in their applications as long as they are implemented in pure Go.

9.2.1.3 Storage

AppEngine provides various types of storage, which operate differently depending on the volatility
of the data. There are three different levels of storage: in memory-cache, storage for semistructured
data, and long-term storage for static data. In this section, we describe DataStore and the use of
static file servers. We cover MemCache in the application services section.

Static file servers

Web applications are composed of dynamic and static data. Dynamic data are a result of the logic

of the application and the interaction with the user. Static data often are mostly constituted of the
components that define the graphical layout of the application (CSS files, plain HTML files,
JavaScript files, images, icons, and sound files) or data files. These files can be hosted on static file
servers, since they are not frequently modified. Such servers are optimized for serving static con-
tent, and users can specify how dynamic content should be served when uploading their applica-
tions to AppEngine.

DataStore

DataStore is a service that allows developers to store semistructured data. The service is designed to scale and optimized to quickly access data. DataStore can be considered as a large object database in which to store objects that can be retrieved by a specified key. Both the type of the key and the structure of the object can vary.

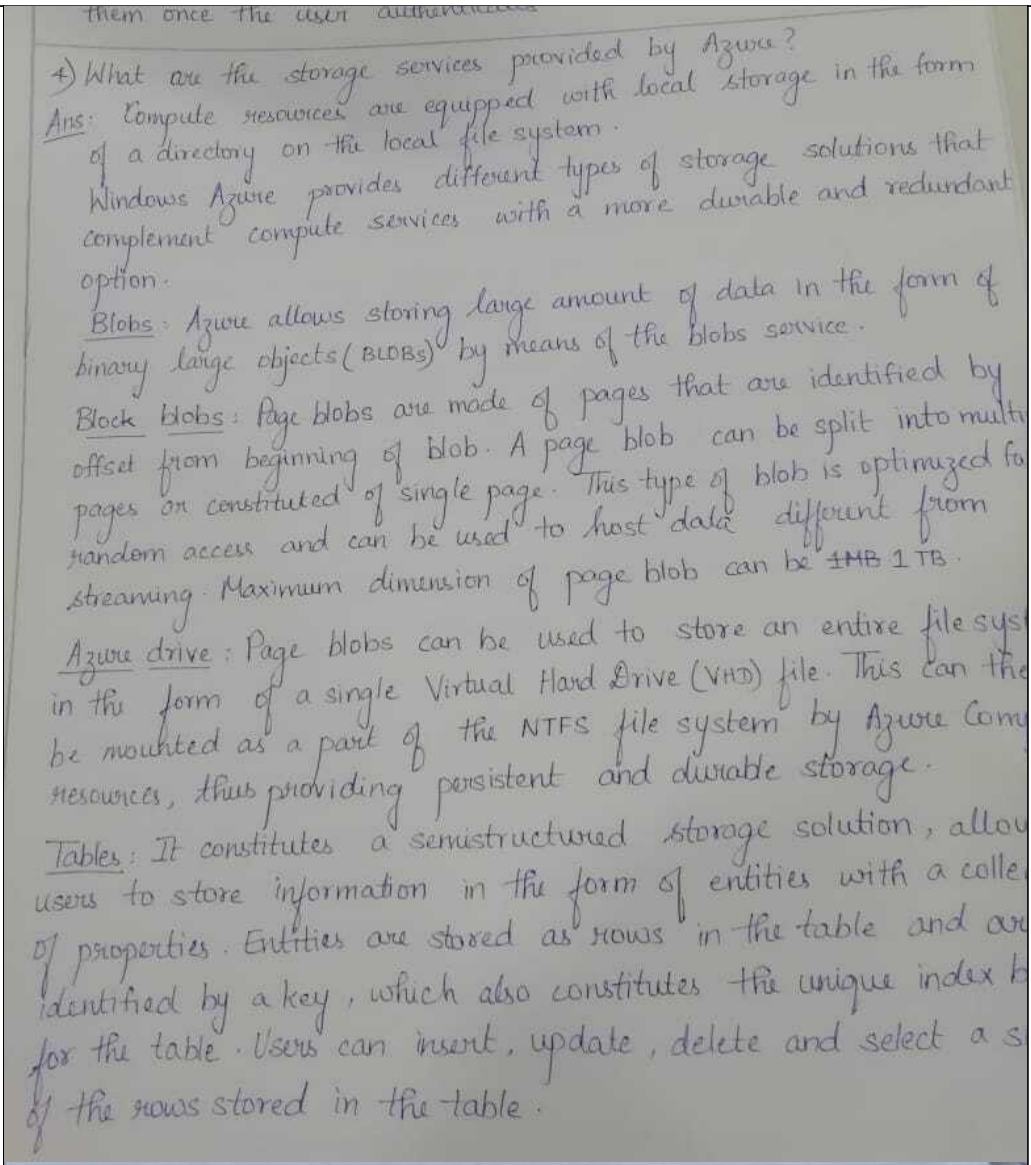
With respect to the traditional Web applications backed by a relational database, DataStore imposes less constraint on the regularity of the data but, at the same time, does not implement some of the features of the relational model (such as reference constraints and join operations). These design decisions originated from a careful analysis of data usage patterns for Web applica-
tions and were taken in order to obtain a more scalable and efficient data store. The underlying infrastructure of DataStore is based on Bigtable, a redundant, distributed, and semistructured data store that organizes data in the form of tables.

DataStore provides high-level abstractions that simplify interaction with Bigtable. Developers define their data in terms of entity and properties , and these are persisted and maintained by the service into tables in Bigtable. An entity constitutes the level of granularity for the storage, and it identifies a collection of properties that define the data it stores. Properties are defined according to one of the several primitive types supported by the service. Each entity is associated with a key, which is either provided by the user or created automatically by AppEngine. An entity is associated with a named kind that AppEngine uses to optimize its retrieval from Bigtable. Although entities and properties seem to be similar to rows and tables in SQL, there are a few differences that have to be taken into account. Entities of the same kind might not have the same properties, and proper-
ties of the same name might contain values of different types. Moreover, properties can store different versions of the same values. Finally, keys are immutable elements and, once created, they cannot be changed.

DataStore also provides facilities for creating indexes on data and to update data within the context of a transaction. Indexes are used to support and speed up queries. A query can return zero or more objects of the same kind or simply the corresponding keys. It is possible to query the data store by specifying either the key or conditions on the values of the properties. Returned result sets can be sorted by key value or properties value. Even though the queries are quite similar to SQLqueries, their implementation is substantially different. DataStore has been designed to be extremely fast in returning result sets; to do so it needs to know in advance all the possible queries that can be done for a given kind, because it stores for each of them a separate index. The indexes are provided by the user while uploading the application to AppEngine and can be automatically defined by the development server. When the developer tests the application, the server monitors all the different types of queries made against the simulated data store and creates an index for them. The structure of the indexes is saved in a configuration file and can be further changed by the developer before uploading the application. The use of precomputed indexes makes the query execution time-independent from the size of the stored data but only influenced by the size of the result set.

The implementation of transaction is limited in order to keep the store scalable and fast. AppEngine ensures that the update of a single entity is performed atomically. Multiple operations on the same entity can be performed within the context of a transaction. It is also possible to update multiple entities atomically. This is only possible if these entities belong to the same entity group. The entity group to which an entity belongs is specified at the time of entity creation and cannot be

	changed later. With regard to concurrency, AppEngine uses an optimistic concurrency control: If one user tries to update an entity that is already being updated, the control returns and the operation fails. Retrieving an entity never incurs into exceptions. *Application services* Applications hosted on AppEngine take the most from the services made available through the runtime environment. These services simplify most of the common operations that are performed in Web applications: access to data, account management, integration of external resources, messaging and communication, image manipulation, and asynchronous computation. *UrlFetch* Web 2.0 has introduced the concept of composite Web applications. Different resources are put together and organized as meshes within a single Web page. Meshes are fragments of HTML generated in different ways. They can be directly obtained from a remote server or rendered from an XML document retrieved from a Web service, or they can be rendered by the browser as the result of an embedded and remote component. A common characteristic of all these examples is the fact that the resource is not local to the server and often not even in the same administrative domain. Therefore, it is fundamental for Web applications to be able to retrieve remote resources. The sandbox environment does not allow applications to open arbitrary connections through sockets, but it does provide developers with the capability of retrieving a remote resource through HTTP/HTTPS by means of the UrlFetch service. Applications can make synchronous and asynchronous Web requests and integrate the resources obtained in this way into the normal request handling cycle of the application. One of the interesting features of UrlFetch is the ability to set deadlines for requests so that they can be completed (or aborted) within a given time. Moreover, the ability to perform such requests asynchronously allows the applications to continue with their logic while the resource is retrieved in the background. UrlFetch is not only used to integrate meshes into a Web page but also to leverage remote Web services in accordance with the SOA reference model for distributed application			
6(a)	List storage services provided by Microsoft Azure? **Answer:**	[06]	CO3	L1


them once the user authenti

4) What are the storage services provided by Azure?

Ans: Compute resources are equipped with local storage in the form of a directory on the local file system.

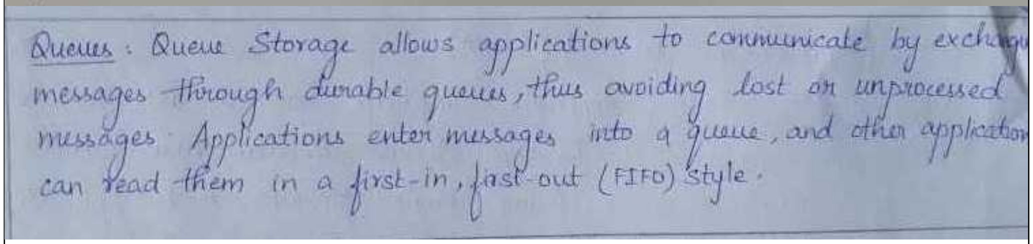
Windows Azure provides different types of storage solutions that complement compute services with a more durable and redundant option.

Blobs: Azure allows storing large amount of data in the form of binary large objects (BLOBs) by means of the blobs service.

Block blobs: Page blobs are made of pages that are identified by offset from beginning of blob. A page blob can be split into multi pages or constituted of single page. This type of blob is optimized fo random access and can be used to host data different from streaming. Maximum dimension of page blob can be ~~1MB~~ 1 TB.

Azure drive: Page blobs can be used to store an entire file sys in the form of a single Virtual Hard Drive (VHD) file. This can the be mounted as a part of the NTFS file system by Azure Com resources, thus providing persistent and durable storage.

Tables: It constitutes a semistructured storage solution, allo users to store information in the form of entities with a colle of properties. Entities are stored as rows in the table and ar identified by a key, which also constitutes the unique index b for the table. Users can insert, update, delete and select a s of the rows stored in the table.

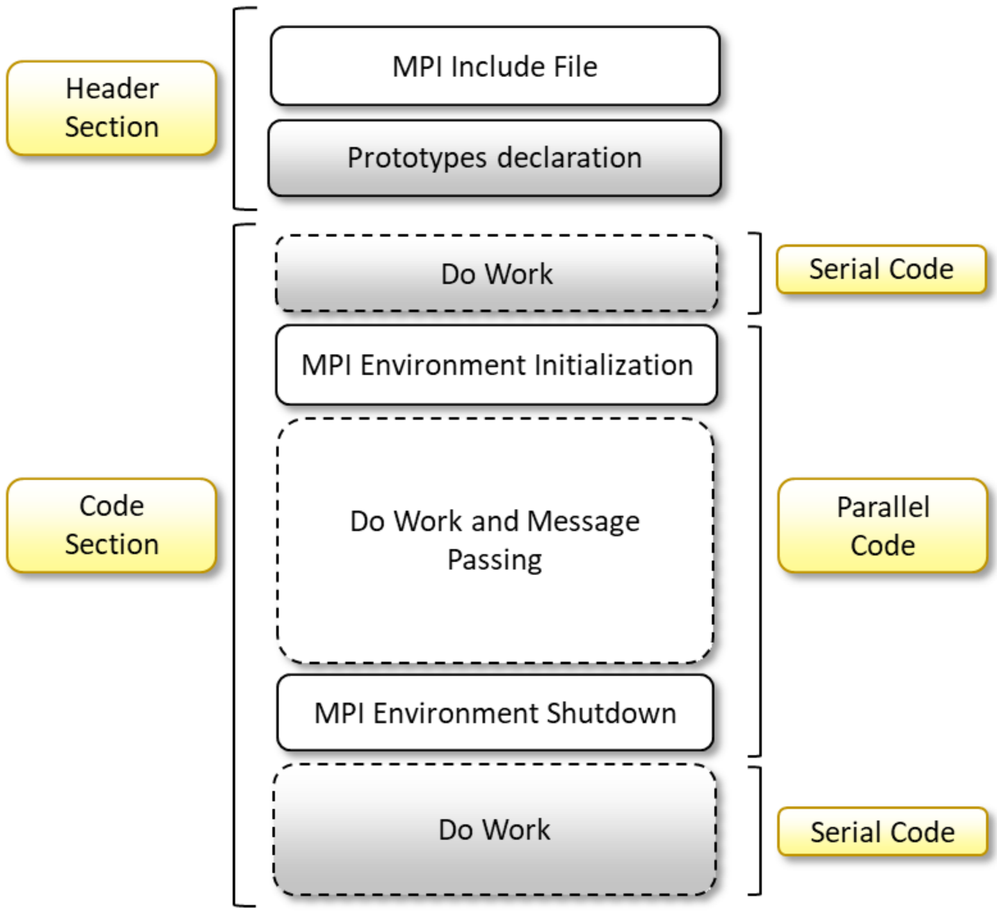

Queues: Queue Storage allows applications to communicate by exchan messages through durable queues, thus avoiding lost or unprocessed messages. Applications enter messages into a queue, and other application can read them in a first-in, first-out (FIFO) style.

2 (a) Describe MPI program structure with a neat diagram [05] CO2 L1

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.



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.

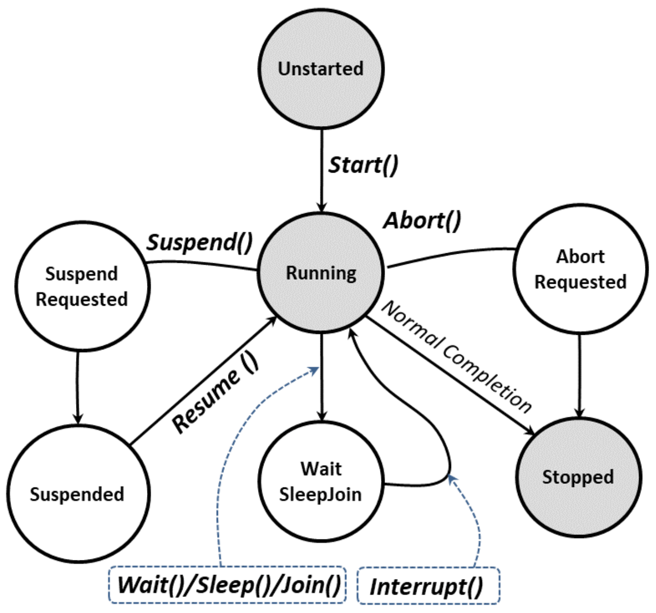
The white balloons in the figure indicate states that do not have a corresponding mapping on the other life cycle; the shaded balloons indicate the common states. Moreover, in local threads most of the state transitions are controlled by the developer, who actually triggers the state transition by invoking 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().

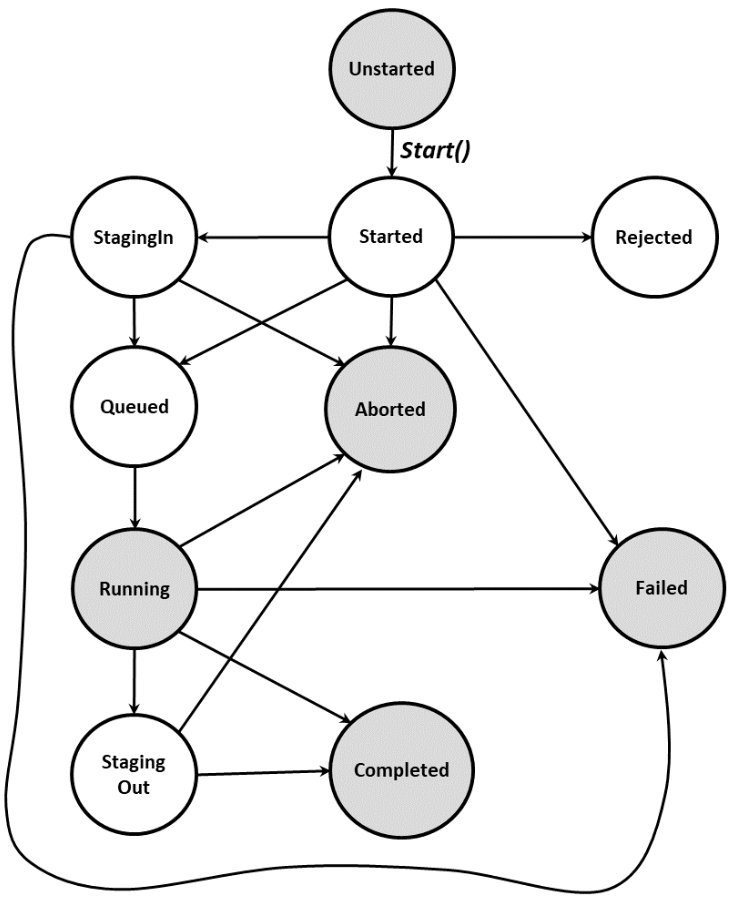
Another outcome might be the Rejected state that occurs if the thread is started with an invalid reservation token. This is a final state and implies execution failure due to lack of rights. Once the thread is in the queue, if there is a free node where to execute it, the middleware moves all the object 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

	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			



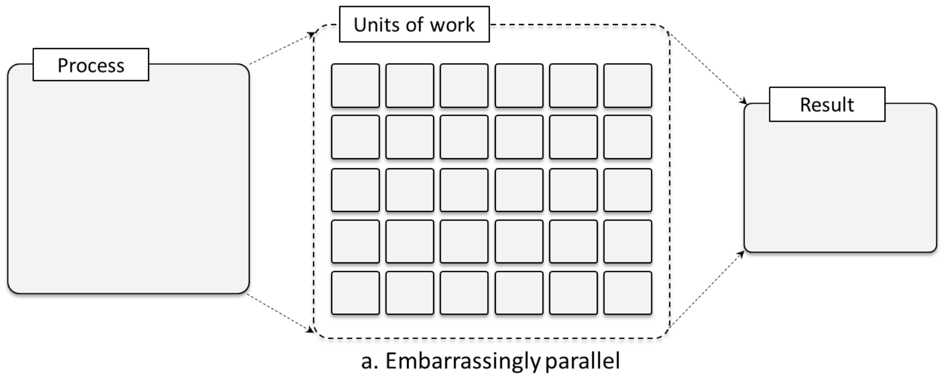
a. System.Threading.Thread life cycle.



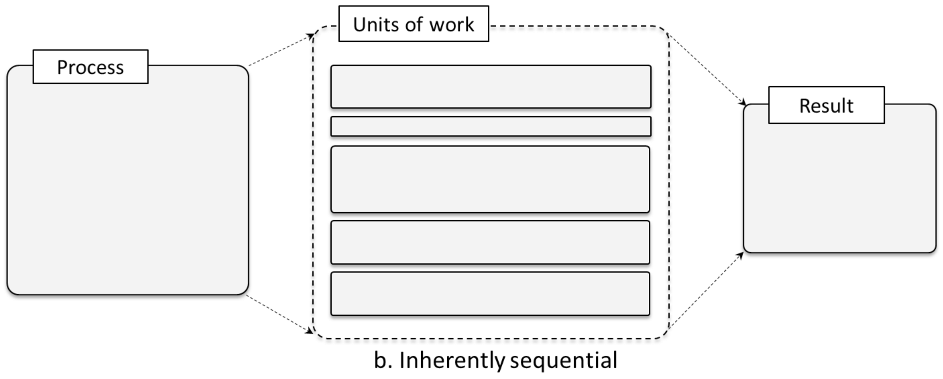
b. Aneka.Threading.AnekaThread life cycle.

	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

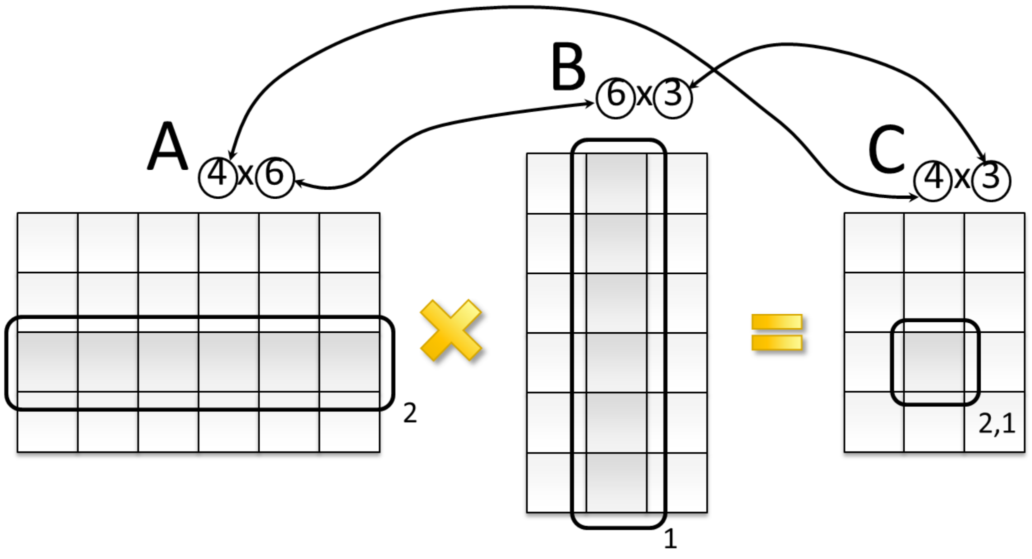


a. Embarrassingly parallel



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.

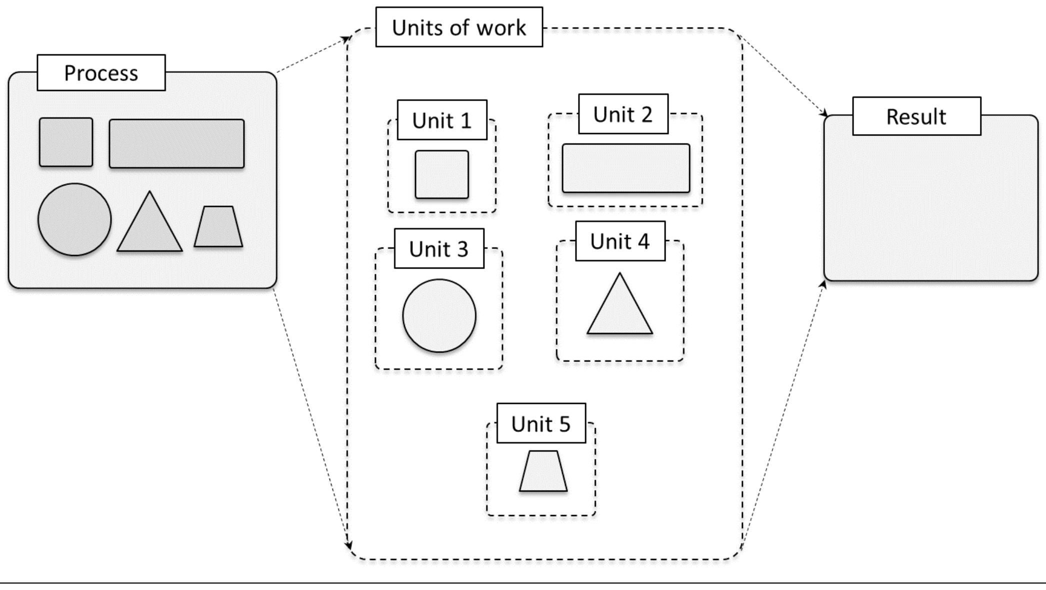
It is evident that the repetitive operation is the computation of each of the elements of the resulting matrix. These are subject to the same formula, and the computation does not depend on values that have been obtained by the computation of other elements of the resulting matrix. Hence, the problem 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.

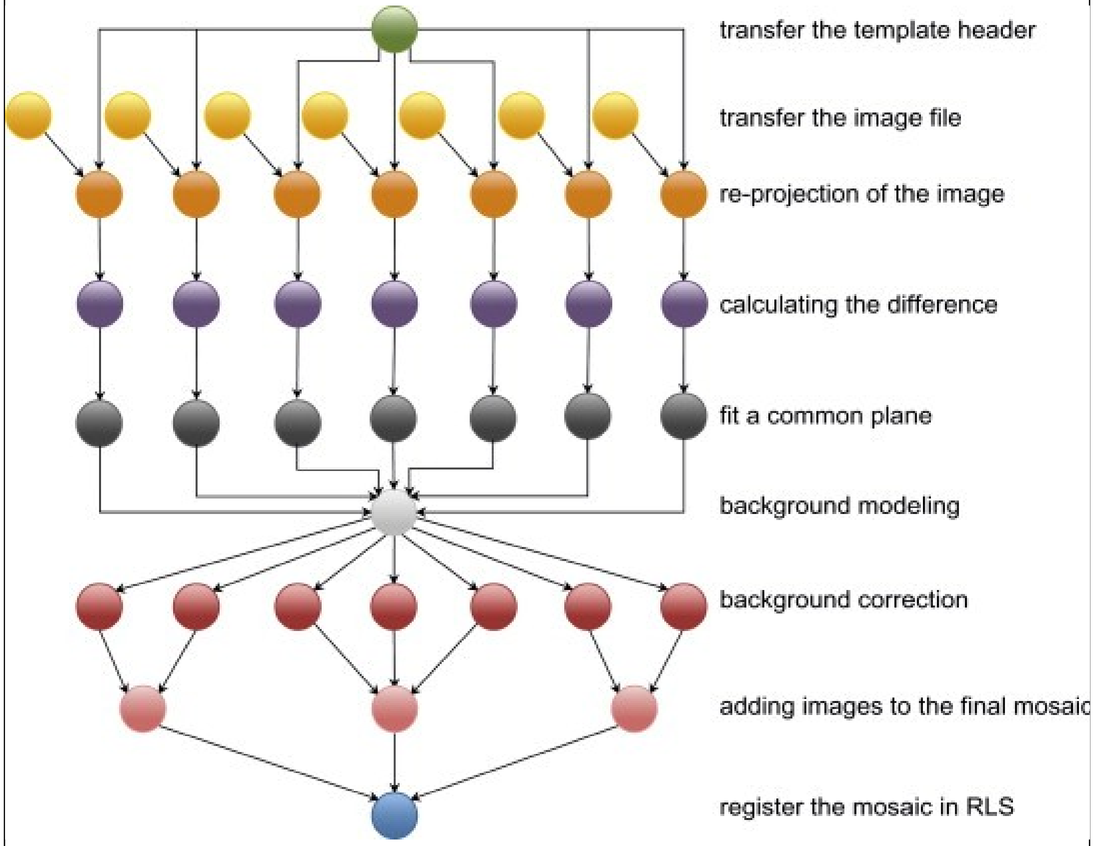
Functional Decomposition

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: $f(x)=six(x)+\cos(x)+\tan(x)$ 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	[06]	CO2	L2

	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 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.  transfer the template header transfer the image file re-projection of the image calculating the difference fit a common plane background modeling background correction adding images to the final mosaic register the mosaic in RLS			
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	04	CO1	L1

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. 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.	06	CO3	L3

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.			
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 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.	04	CO3	L3

7(a)	Describe the different task-based application models	[06]	CO1	L1

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.

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	[04]	CO3	L2

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