CHAPTER 5

CLOUD PROGRAMMING MODELS

CHAPTER OUTLINE

After studying this chapter, students will be able to understand the:

- Thread Programming, Thread in Java and. NET., Creating Threads , Destroying Threads
- Parallel Computation with Threads
- Multithreading With Aneka5, The Aneka Environment, Developing Parallel Applications with Aneka Threads
- Class Design And Serialization
- Task Programming, Task Computing, Workflow Technologies
- Data-Intensive Computing, Technologies for Data-Intensive Computing
- Map-Reduce Programming,
- Google Map-reduce Infrastructure

THREAD PROGRAMMING

Concurrent computing is a type of computing in which many calculations are done concurrently or overlapping periods rather than sequentially in which one must finish before the next can begin. This is a feature of the software, computer, or network system in which each process has its execution point or "thread". With concurrent computing, program throughput will be increased. The parallel execution of a concurrent program increases the number of tasks done in a given time in proportion to the number of processors at the same time it will also increase the responsiveness for input/output of the program.

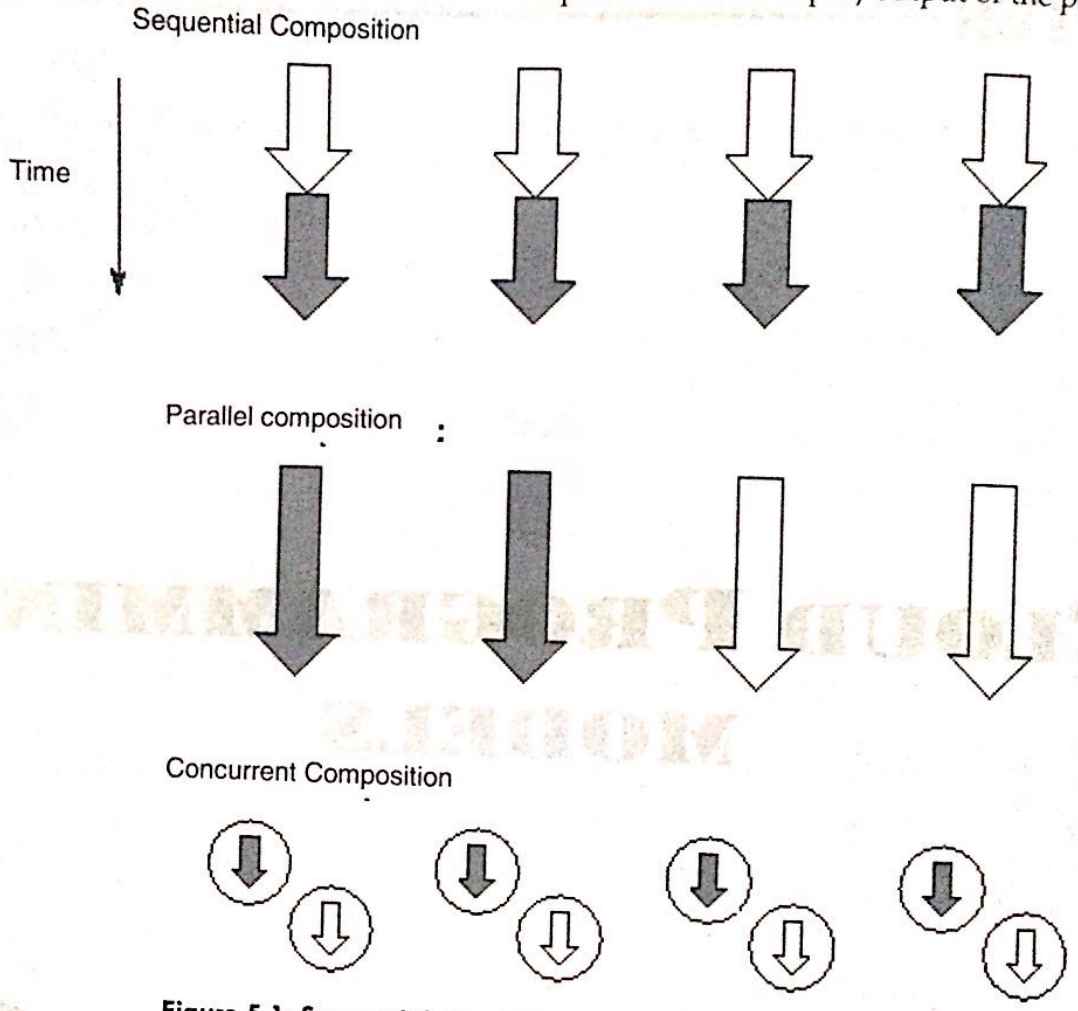


Figure 5.1: Sequential, Parallel and Concurrent Computing

The performance of tasks by a computing service or device over a certain period is referred to as throughput. It compares the quantity of work accomplished to the amount of time spent and may be used to evaluate the performance of a processor, memory, and/or network connections. Related metrics of system productivity include the rate at which a given job may be performed and response time, which is the amount of time that elapses between a single interactive user request and the delivery of a response.

Throughput assesses the efficiency of computer processors which is usually expressed in terms of batch jobs or tasks per second and millions of instructions per second. The system's total throughput can be measured concerning the volume and complexity of work, the number of concurrent users, and the responsiveness of the application. So, throughput computing is concerned with delivering large amounts of computing in the form of transactions. Throughput computing was originally associated with the field of transaction processing, although it has subsequently been extended outside that sector. Advances in hardware technology enabled the development of multicore computers, which enabled the delivery of high-throughput calculations even inside a single computer system. Multiprocessing and multithreading are used to achieve throughput computing. The execution of numerous programs on a single computer is

referred to as multiprocessing, whereas multithreading refers to the possibility of several instruction streams inside the same program.

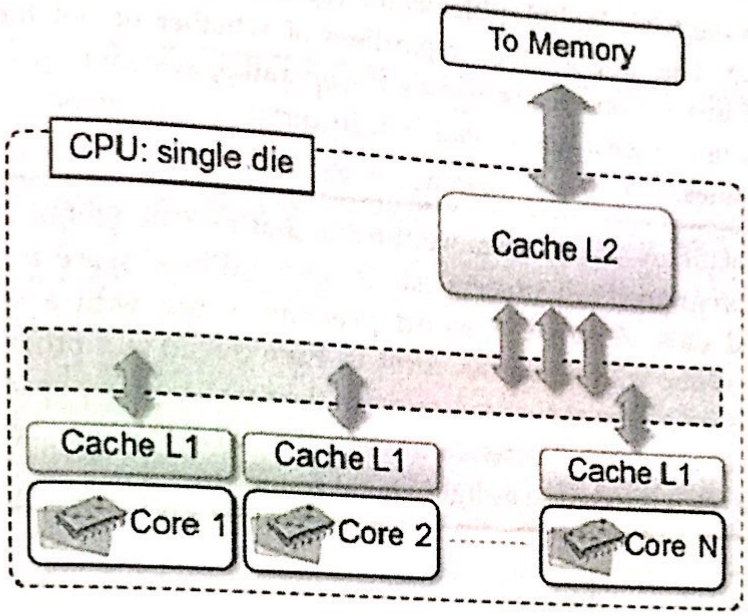


Figure 5.2: Multicore processor

Multiple operations can be performed by modern apps at the same time. To express intra-process concurrency, developers structure programs in terms of threads. Threads can be used in both implicit and explicit ways. When the underlying APIs employ internal threads to accomplish specific tasks supporting application execution, such as graphical user interface (GUI) rendering or garbage collection in the case of virtual machine-based languages, this is referred to as implicit threading. The usage of threads within a program by application developers, who employ this abstraction to add parallelism, is characterized as explicit threading. I/O from devices and network connections, extensive calculations, or the execution of background processes with no particular time boundaries are common scenarios in which threads are intentionally employed.

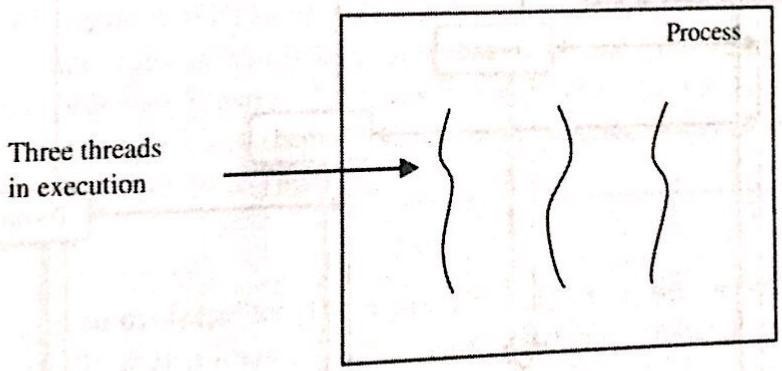


Figure 5.3: A process containing multiple threads

The usage of threads was first aimed at facilitating asynchronous operations—specifically, giving tools for asynchronous I/O or extensive calculations so that application user interfaces did not stop or become unresponsive. With the introduction of parallel architectures, multithreading has become a helpful tool for increasing system throughput and a feasible choice for throughput computing. To that end, the usage of threads has a significant influence on the design of algorithms that must be refactored to exploit threads.

What is a thread?

Within a process, a thread specifies a single control flow, which is a logical series of instructions. By the logical sequence of instructions, we imply a set of instructions that are meant to be performed sequentially. More frequently, a thread specifies a type of yarn used for sewing, and the sense of continuity given by the

interlocked fibers of that yarn is used to remember the idea that thread instructions represent a logically continuous sequence of actions.

Threads are identified as the basic building blocks for representing executing code in operating systems that allow multithreading. This implies that, regardless of whether or not they are explicitly used by developers, each series of instructions performed by the operating system is inside the context of a thread. As a result, each process has at least one thread but, in certain circumstances, is made up of numerous threads with varying lifetimes.

A running program is made up of one or more threads that execute within a process. A process is a running instance of a program. Each process has its own address space in memory, as well as its executable program and data. A single-threaded program is one with a single thread, whereas a multithreaded program is one with numerous threads. Each thread in a program has its stack to keep track of its state. Threads inside the same process share the same memory space and execution context. A process is a collection of resources, whereas a thread is an object that may be scheduled for execution on the CPU. Threads are often referred to as lightweight processes.

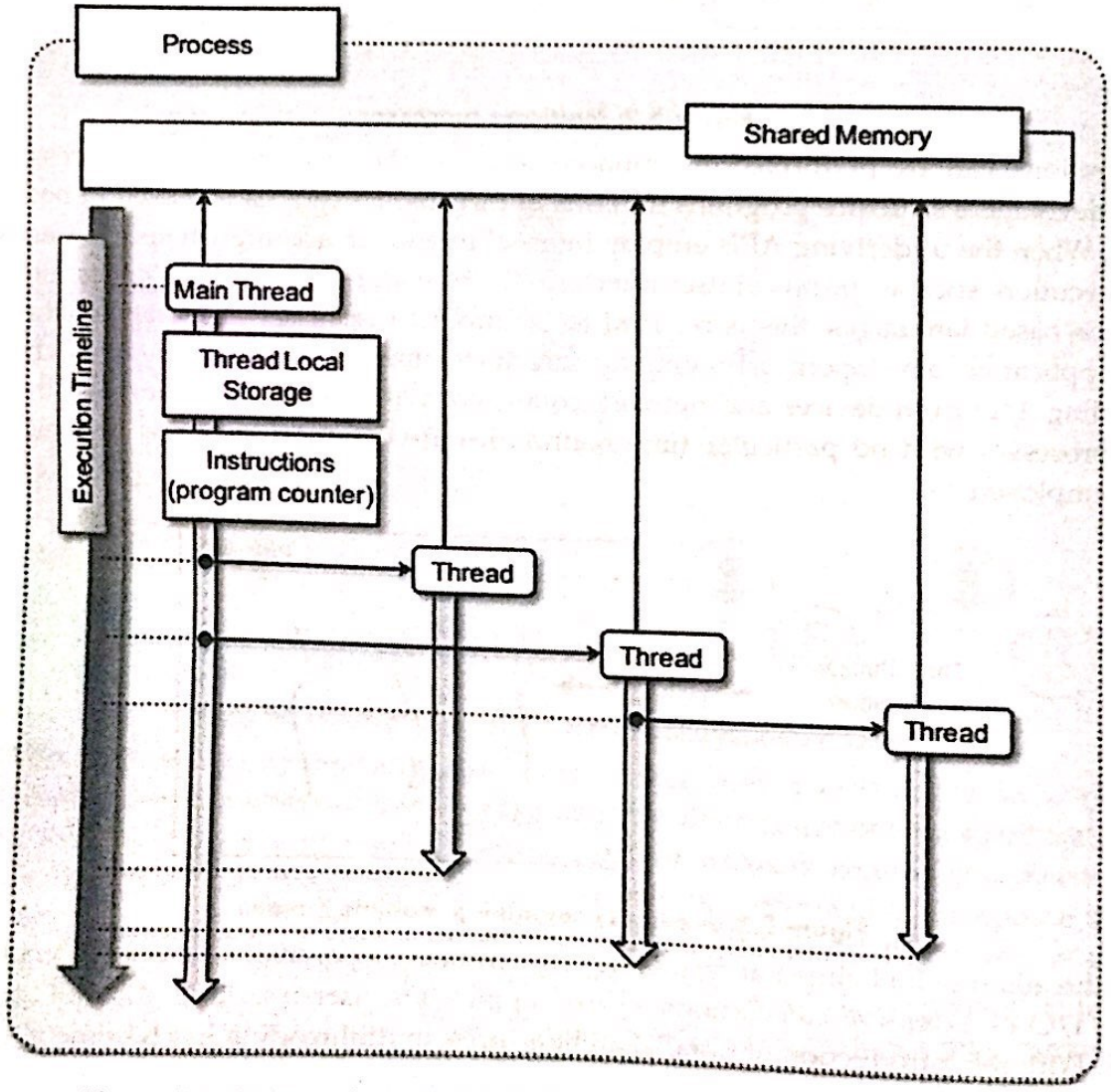


Figure 5.4: An overview of the relation between threads and processes[5]

The operating system in a multitasking environment assigns various time slices to each process and interleaves their execution. A context switch is a process of momentarily suspending the execution of one process, storing all the information in the registers, and replacing it with information relevant to another process.This procedure is often seen as difficult, and the use of multithreading reduces the delay imposed by context shifts, allowing for the execution of several tasks in a lighter manner. The state representing the

execution of a thread is minimal compared to the one describing a process.As a result, moving between threads is preferable to moving between processes. The usage of many threads instead of several processes is justifiable if and only if the tasks executed are logically connected and require sharing memory or other resources.

BASICS OF THREAD

- Thread operations include thread creation, termination, synchronization (joins, blocking), scheduling, data management, and process interaction.
- A thread does not maintain a list of created threads, nor does it know the thread that created it.
- All threads within a process share the same address space.
- Threads in the same process share:
  - Process instructions and data
  - open files (descriptors)
  - signals and signal handlers
  - current working directory
  - User and group id
- Each thread has a unique:
  - Thread ID
  - set of registers, stack pointer
  - stack for local variables, return addresses
  - signal mask
  - priority
  - Return value: errno
- pthread functions return "0" if OK.

Figure 5.4 depicts the relationship between threads and processes, as well as a simplified depiction of the runtime execution of a multithreaded program. A process, which has at least one thread, usually known as the main thread, identifies a running application. The compiler or the runtime environment that executes the program creates such a thread implicitly. This thread is likely to last for the duration of the process and to be the genesis of other threads, which have a shorter length in general. These threads, as main threads, can spawn additional threads.There is no distinction between the main thread and the additional threads that are produced over the process's lifespan. Each has its local storage and a set of instructions to execute, and they all share the memory space set aside for the overall process. When all of the threads have been completed, the procedure is deemed to be finished.

Posix Threads

The Portable Operating System Interface for Unix (POSIX) is a set of standards for application programming interfaces (API) that allows the creation of portable applications on the Unix operating system. POSIX 1.c defines thread implementation and the functions to which application programmers should have access to build portable multithreaded applications. Although the specifications are intended for Unix-based operating systems, a Windows-based implementation of the same specification is also available. The POSIX method has been used as a paradigm for multiple implementations, which may provide developers with a different interface but the same functionality.Crucial issuesto note from a programming point of view as mentioned byBuyya et al.is listed below[5]:

- 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.

POSIX 1.c specification is provided for the C programming language. The *thread.h* header file, which is included with typical C implementations, exposes all accessible functions and data structures.

The POSIX thread libraries are a standard thread API for C/C++. It allows the creation of a new concurrent process flow. It works well on multi-processor or multi-core systems, where the process flow may be scheduled to execute on another processor, increasing speed through parallel or distributed processing. Threads require less overhead than "*forking*" or creating a new process because the system does not initialize a new system virtual memory space and environment for the process. While multiprocessor systems are the most successful, improvements are also discovered on uniprocessor systems that exploit delays in I/O and other system operations that may stall process execution. One thread may be running while another waits for I/O or other system delays. Parallel programming technologies such as Message Passing Interface (*MPI*) and Parallel Virtual Machine (PVM) are used in a distributed computing environment while threads are limited to a single computer system. All threads within a process share the same address space. A thread is formed by specifying a function and the parameters that will be handled by the thread. The goal of utilizing the POSIX thread library in your software is to make it run quicker.

Thread Creation and Termination with C

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void *print_message_function( void *ptr );
main()
{
pthread_t thread1, thread2;
   char *message1 = "Thread 1";
   char *message2 = "Thread 2";
   int  iret1, iret2;
  /* Create independent threads each of which will execute the function.        */
   iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message1);
   iret2 = pthread_create( &thread2, NULL, print_message_function, (void*) message2);

 /*
 Wait till threads are complete before the main continues. Unless we wait, we run the risk of executing an
 exit which will terminate the process and all threads before the threads have completed.
   */
 pthread_join( thread1, NULL);
 pthread_join( thread2, NULL);
 printf("Thread 1 returns: %d\n",iret1);
 printf("Thread 2 returns: %d\n",iret2);
   exit(0);
}
void *print_message_function( void *ptr )
{
   char *message;
   message = (char *) ptr;
printf("%s \n", message);
}
```

Results:

Thread 1
Thread 2
Thread 1 returns: 0
Thread 2 returns: 0

THREAD IN JAVA AND.NET

Using an object-oriented approach, languages such as Java and C# programming platforms give a broad range of abilities for multithreaded application development. As Java and.NET both run code on top of virtual machines, the APIs offered by the libraries relate to managed or logical threads. These are translated to actual threads, which are made available as abstractions by the underlying OS via the runtime environment in which these languages' applications operate.[5]

The 'Thread' class in Java and .NET provides the thread abstraction by offering the frequent operations done on threads such as start, stop, suspend, resume, abort, sleep, join, and interrupt. Start and stop are used to govern the thread instance's lifetime, whereas suspend and resume are used to programmatically pause and then continue the thread operation. These two operations are deprecated from implementations with Java and .Net. So,similar feature can be gained with the use of sleep operationwhich allows pausing the execution of a thread for a set amount of time. This operation is distinct from the join operation, which causes one thread to wait while another thread completes. These waiting states can be interrupted by employing the interrupt operation, which resumes the thread's execution and raises an exception within the thread's code to indicate the unusual resumption.[5]

The Java and .Net frameworks offer different support for establishing thread synchronization. The base class libraries or supplementary libraries entirely cover the essential features for implementing mutexes, critical regions, and reader-writer locks. Both languages support more complex constructions than the thread concept. In the case of Java, the majority of them are contained in the *java.util.concurrent* package, whereas the *.NET Parallel Extension Framework* extends the rich set of APIs for concurrent programming using .Net.[5]

THREAD PROGRAMMING IN JAVA

Java gives two ways to create a thread. The first one is by extending the *Thread* class and the next is by implementing the *Runnable* interface.The Thread class provides constructors and methods for creating and operating on threads. The thread extends Object and implements the Runnable interface.

Constructors of Thread class

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r, String name)

Methods of Thread class

Methods	Description
public void run()	Any class whose instances are meant to be run by a thread should implement the Runnable interface. The Runnable interface has a single function called run 0.
public void start()	This method starts the execution of the newly created thread. JVM calls the run() method on the thread.
public void sleep(long milliseconds)	Causes the currently executing thread to sleep temporarily for

	the specified number of milliseconds.
public void join()	Waits for a thread to die.
public void join(long milliseconds)	Waits for a thread to die for the specified milliseconds.
public int getPriority()	Returns the priority of the thread.
public int setPriority(int priority)	Changes the priority of the thread.
public String getName()	Returns the name of the thread.
public void setName(String name)	Changes the name of the thread.
public Thread currentThread()	Returns the reference of the currently executing thread.
public int getId()	Returns the id of the thread.
public Thread.StategetState()	Returns the state of the thread.
public booleanisAlive()	Tests if the thread is alive.
public void yield()	Causes the currently executing thread object to temporarily pause and allow other threads to execute.
public booleanisDaemon()	Tests if the thread is a daemon thread.
public void setDaemon(boolean b)	Marks the thread as daemon or user thread.
public void interrupt()	Interrupts the thread
public booleanisInterrupted()	Tests if the thread has been interrupted.
public static booleaninterrupted()	Tests if the current thread has been interrupted.

Creating Thread in Java by extending Thread class

```
class ThreadClass extends Thread{
public void run(){
    System.out.println("Thread created with Thread class is running.");
}
public static void main(String args[]){
    ThreadClass  t1=new ThreadClass  ();
    t1.start();
}
}
```

Creating Thread in Java by implementing Runnable interface

```
class ThreadClassR implements Runnable{
public void run(){
    System.out.println("Thread created with Runnable interface is running.");
}
public static void main(String args[]){
    ThreadClassR m1=new ThreadClassR ();
    Thread t1 =new Thread(m1);
    t1.start();
}
```

THREAD PROGRAMMING IN .NET

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

C# Thread Properties

A list of some of the important properties of the Thread class is given below:

Property	Description
CurrentContext	Gets the current context in which the thread is executing.
CurrentThread	Returns the instance of the currently running thread.
IsAlive	Checks whether the current thread is alive or not. It is used to find the execution status of the thread.
IsBackground	Used to get or set value whether the current thread is in the background or not.
IsThreadPoolThread	Gets a value indicating whether or not a thread belongs to the managed thread pool.
ManagedThreadId	Used to get a unique id for the currently managed thread.
Name	Used to get or set the name of the current thread.
Priority	Used to get or set the priority of the current thread.
ThreadState	Used to return a value representing the thread state.

C# Thread Methods

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

Method	Description
public void Abort()	Used to terminate the thread. It raises ThreadAbortException.
public void Interrupt()	Used to interrupt a thread that is in *WaitSleepJoin* state.
public void Join()	Used to block all the calling threads until this thread terminates.
public static void ResetAbort()	Used to cancel the Abort request for the current thread.
public static void Sleep(int millisecondsTimeout)	Used to suspend the current thread for the specified milliseconds.
public void Start()	Changes the current state of the thread to Runnable.
public static bool Yield()	Used to yield the execution of the current thread to another thread.

C# Main Thread

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

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

```
        Console.WriteLine(t.Name);
    }
}
```

CREATING THREADS

Threads are created by extending the Thread class. The extended Thread class then calls the Start() method to begin the child thread execution.

```
using System;
using System.Threading;
namespace MultithreadingApplication {
  class ThreadCreationProgram {
    public static void CallToChildThread() {
Console.WriteLine("Child thread starts.");
    }
    static void Main(string[] args) {
ThreadStartchildref = new ThreadStart(CallToChildThread);
Console.WriteLine("Main: Creating the Child thread.");
      Thread childThread = new Thread(childref);
childThread.Start();
Console.ReadKey();
    }
  }
}
```

Output

```
Main: Creating the Child thread.
Child thread starts.
```

Managing Threads

The Thread class provides various methods for managing threads. The example below demonstrates the use of the *sleep()* method for making a thread pause for a specific period.

```
using System;
using System.Threading;
namespace MultithreadingApplication {
  class ThreadCreationProgram {
    public static void CallToChildThread() {
Console.WriteLine("Child thread starts.");
        // Thread is paused for 4000 milliseconds
      int sleepfor = 4000;
Console.WriteLine("Child Thread Paused for {0} seconds.", sleepfor / 1000);
Thread.Sleep(sleepfor);
Console.WriteLine("Child thread resumes.");
    }
  static void Main(string[] args) {
ThreadStartchildref = new ThreadStart(CallToChildThread);
Console.WriteLine("Main: Creating the Child thread.");

    Thread childThread = new Thread(childref);
childThread.Start();
```

```
Console.ReadKey();
    }
  }
}
```

Output

```
Main: Creating the Child thread.
Child thread starts.
Child Thread Paused for 4 seconds.
Child thread resumes.
```

DESTROYING THREADS

The Abort() method is used for destroying threads. The runtime aborts the thread by throwing a ThreadAbortException. This exception cannot be caught, the control is sent to the *finally* block.

```
using System;
using System.Threading;
namespace MultithreadingApplication {
  class ThreadCreationProgram {
    public static void CallToChildThread() {
      try {
Console.WriteLine("Child thread starts.");
        // Counting to 10
        for (int counter = 0; counter <= 10; counter++) {
Thread.Sleep(500);
Console.WriteLine(counter);
        }
Console.WriteLine("Child Thread Completed.");
      } catch (ThreadAbortException e) {
Console.WriteLine("Thread Abort Exception.");
      } finally {
Console.WriteLine("Could not catch the Thread Exception.");
      }
    }
    static void Main(string[] args) {
ThreadStartchildref = new ThreadStart(CallToChildThread);
Console.WriteLine("Main: Creating the Child thread.");
      Thread childThread = new Thread(childref);
childThread.Start();
      // Stop the main thread for some time
Thread.Sleep(2000);
      //Abort the child
Console.WriteLine("Main: Aborting the Child thread.");
childThread.Abort();
Console.ReadKey();
    }
  }
}
```

Output

```
Main: Creating the Child thread.
Child thread starts.
0
1
2
Main: Aborting the Child thread.
Thread Abort Exception.
Could not catch the Thread Exception.
```

PARALLEL COMPUTATION WITH THREADS

Creating parallel applications needs a comprehensive understanding of the problem and its logical structure. Understanding the interdependence and correlations of tasks inside an application is critical for developing the optimal software structure and adding parallelism when necessary. A decomposition is a useful approach for determining if a problem can be broken down into components that can be done concurrently. If such decomposition is achievable, it also provides a starting point for parallel implementation since it allows for the division of work into separate units that may be processed concurrently with the help of threads. Domain and functional decompositions are the two basic decomposition approaches.[5]

MULTITHREADING WITH ANEKA[5]

Traditional thread programming has traditionally been done in the context of a single process. That is, a process is made up of many execution threads that share memory and other resources but operate inside the bounds of the process's memory area. The operating system allocates a portion of the time slice allotted to the entire process to each of the threads included inside. This technique, known as context switching, gives the impression that numerous threads are functioning at the same time. True parallelism is achieved on current computers, when multi-core processors are common, by allocating each thread to a single core.

A multi-threaded program has various benefits, including greater responsiveness in interactive programs, better throughput in I/O heavy applications, enhanced server responsiveness when dealing with several clients, and a streamlined program structure. Threads within a single process, on the other hand, are unable to connect with threads in other processes through shared memory and must instead rely on other means of inter-process communication, such as named pipes and sockets.

The increasing complexity of the applications, there arises a larger demand for processing capacity. Often, the computational capability of a single computer is insufficient to meet this demand. Then, distributed infrastructures such as clouds must be used. Decomposition methods may be used to divide a particular program into multiple pieces of work that, instead of being run as threads on a single node, can be submitted for execution through cloud computing which will improve the execution time and increase the performance.

Although a distributed infrastructure may significantly improve the parallelism of programs, its use comes at a cost in terms of application design and performance. Because the multiple units of work are not executing in the same process space but on distinct nodes, both the code and the data must be relocated to a new execution context; the same is true for results that must be gathered remotely and returned to the master process. Furthermore, if there is any communication among the different employees, the communication model must finally be redesigned by using the APIs given by the middleware.In other words, the shift from a single-process multithreaded execution to a distributed execution is not always smooth, and application redesign and reimplementation are frequently necessary.

The amount of work necessary to convert an application is determined by the capabilities provided by the middleware that manages the distributed infrastructure. Aneka, as middleware for managing clusters, grids, and clouds, provides sophisticated features for developers to construct distributed applications.Aneka pushes the boundaries of typical thread programming model. It allows you to design standard multi-threaded programs, with the extra benefit that each of these threads can now be operated outside of the parent process and more specially on a different computer. These "threads" are separate processes that run on distinct nodes and do not share a memory or other resources. AnekaThreads, allows you to design programs that use the same thread mechanisms for concurrency and synchronization as traditional threads. This makes it simple to convert existing multi-threaded compute-intensive apps to parallel ones that can run faster by leveraging several computers at the same time.

THE ANEKA ENVIRONMENT

Aneka is a development as well as a runtime environment. As a development environment, Aneka provides a collection of libraries that allow you to construct parallel applications using one of the three supported programming paradigms. Those are Task, Thread, and MapReduce. As a runtime environment, Aneka parallelizes the units of work that comprise your program. It should be noted that executing apps on Aneka requires access to a pre-installed runtime environment on a cluster. You may, however, run Aneka solo on your personal computer for development reasons.

AnekaThreads

An *AnekaThread* is a piece of work that may be run on a remote computer. Unlike ordinary threads, each AnekaThread runs in its process on the distant system. An *AnekaThread* provides a comparable interface to a normal Thread and may be started and aborted in the same way. AnekaThreads, on the other hand, are simpler and do not support all of the behaviors provided by the .Net Thread class, such as thread priority management and actions like suspend and resume.

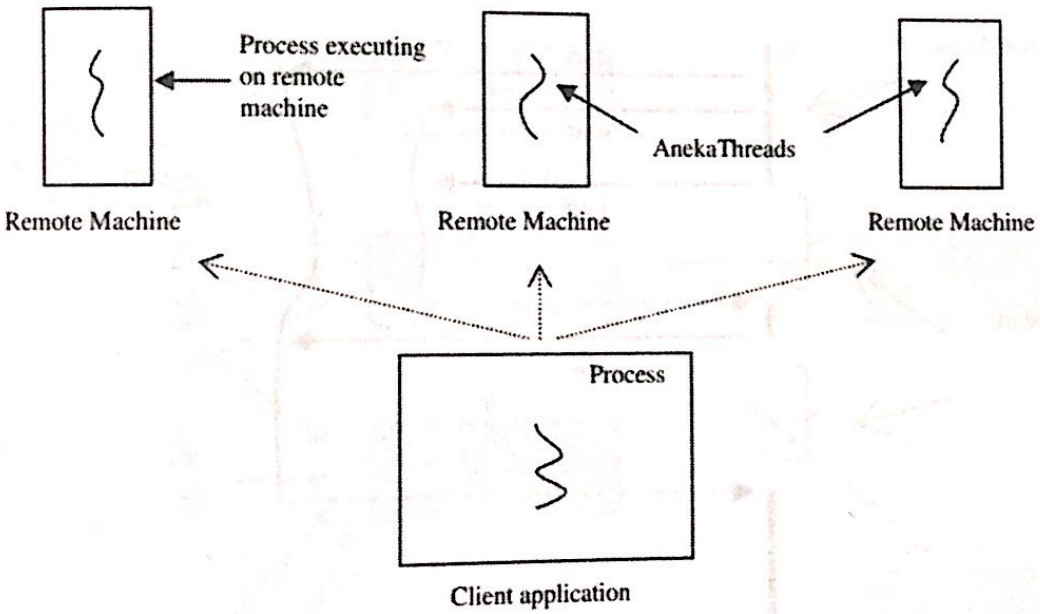


Figure 5.5: A process executing AnekaThreads on remote machines

Standard .Net Threads in comparison with AnekaThreads

.NET Threading API	Aneka Threading API
System.Threading	Aneka.Threading
Thread	AnekaThread
Thread.ManagedThreadId (int)	AnekaThread.Id
Thread.Name	AnekaThread.Name
Thread.ThreadState (ThreadState)	AnekaThread.State

Thread.IsAlive	AnekaThread.IsAlive
Thread.IsRunning	AnekaThread.IsRunning
Thread.IsBackground	[Not provided]
Thread.Priority	[Not provided]
Thread.IsThreadPoolThread	[Not provided]
Thread.Start	AnekaThread.Start
Thread.Abort	AnekaThread.Abort
Thread.Sleep	[Not provided]
Thread.Interrupt	[Not provided]
Thread.Suspend	[Not provided]
Thread.Resume	[Not provided]
Thread.Join	AnekaThread.Join

Thread Synchronization

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

Invoking AnekaThread's join method causes the main application thread to block until the AnekaThread quits, either successfully or unsuccessfully. This fundamental level of synchronization might be advantageous in applications where partial results of calculations are required to proceed further. Because one of these threads executes independently of the others and uses its own private data structures, no further types of synchronization such as locking or signaling are required.

As the Aneka runtime environment is shared by several users, and different programs use the execution nodes, it is not possible to conduct actions such as Suspend, Resume, Interrupt, and Sleep, which might result in retaining a resource forever and preventing it from being utilized.

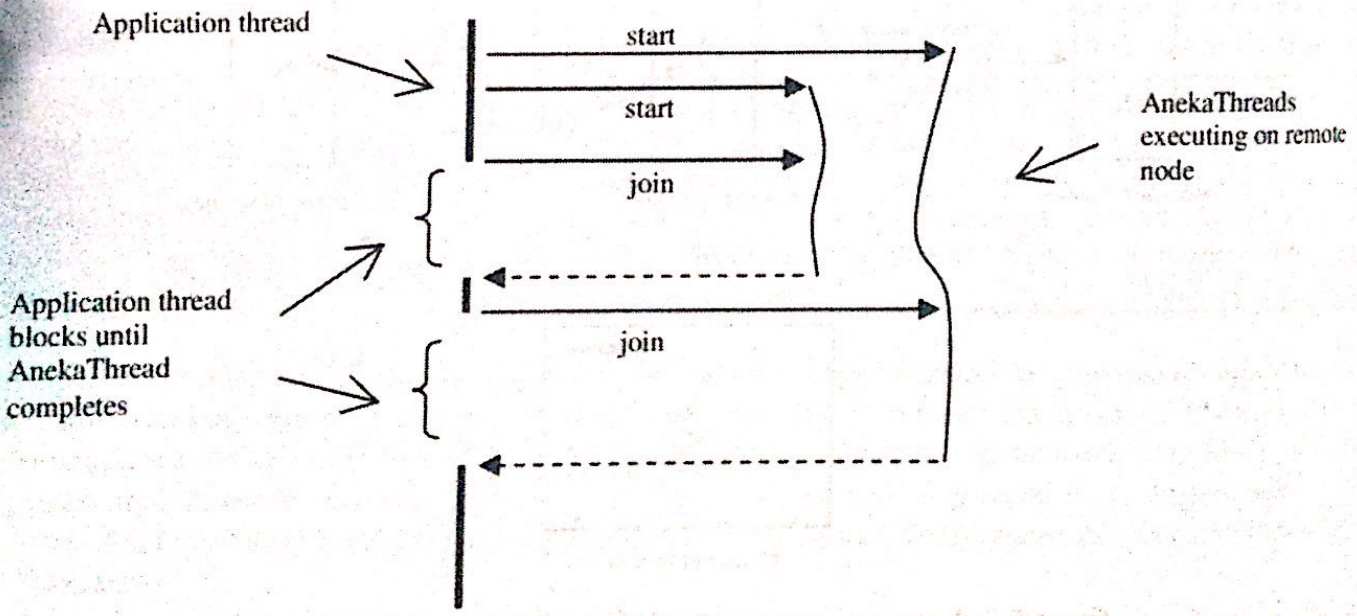


Figure 5.6: Basic synchronization provided by Aneka's threading library

Thread Priorities

The Thread class in .Net supports thread priorities.The scheduling priority can be one of Highest, AboveNormal, Normal, BelowNormal, and Lowest. Operating systems, on the other hand, are not compelled to respect a thread's priority. Aneka's present version does not support thread priority.

Thread Life Cycle

As Aneka threads exist and operate in a distributed environment, their life cycle differs from that of local threads. The following figure depicts the possible execution states for local threads supported by the .Net framework.

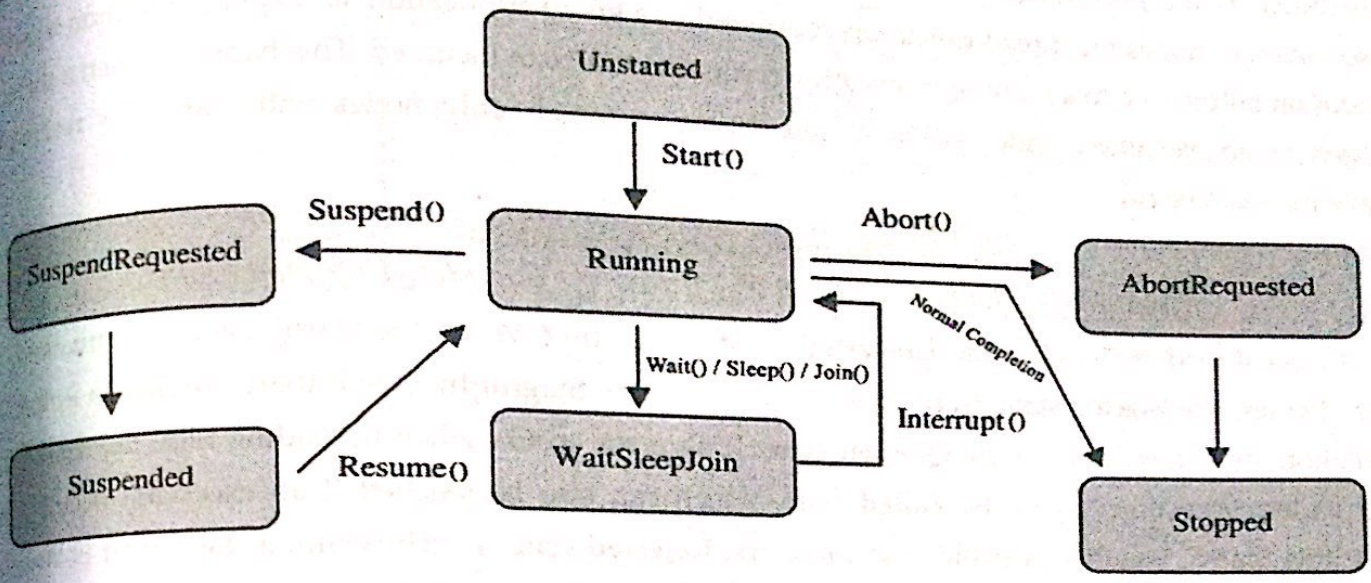


Figure 5.7: The life cycle of local threads in .Net (System.Threading.Threadlife)

A thread may be in one or more of those statuses at any given moment. When a new thread is created, its status is *Unstarted*, and when the Start method is called, it changes to *Running*. There is also a Background state that specifies whether a thread is operating in the background or the foreground. The diagram below depicts the life cycle of an AnekaThread. Because the two thread types are essentially different, one is local and the other is distributed, the possible states they might take vary from instantiation to termination.

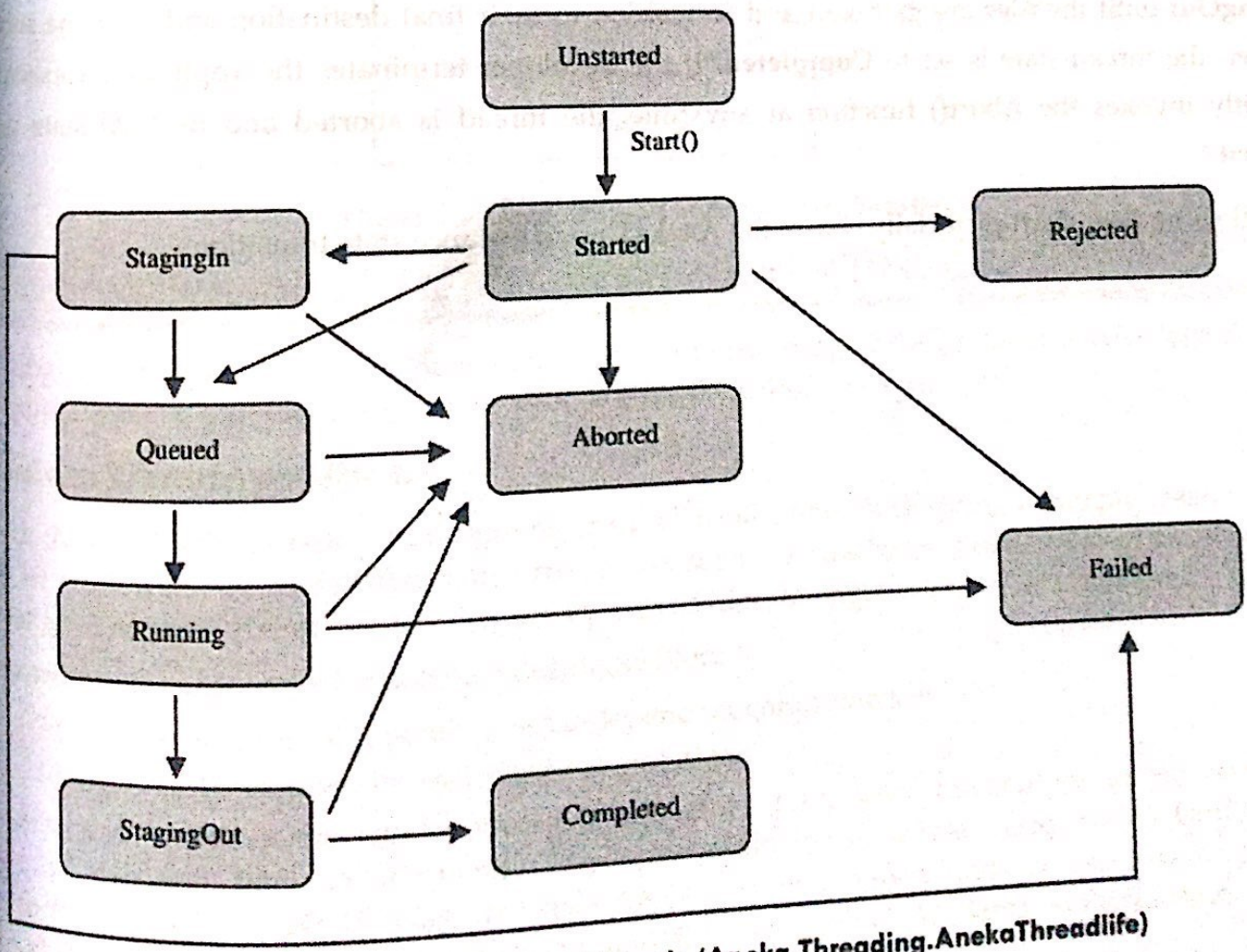


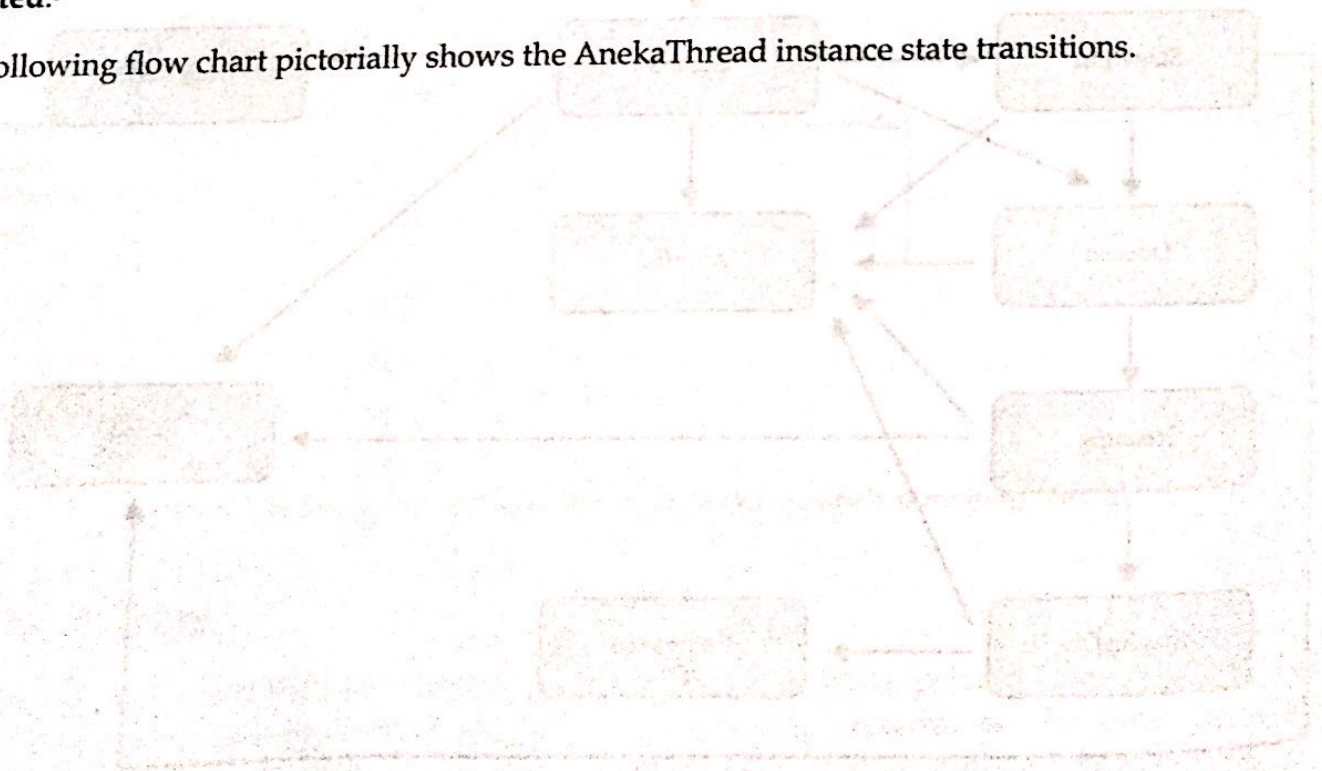
Figure 5.8: The life cycle of AnekaThreads (Aneka.Threading.AnekaThreadlife)

Most state transitions in local threads are managed by the developer, who causes the state transition by executing methods on the thread instance, but many state transitions in Aneka threads are handled by the middleware. Aneka threads have more states than local threads since they support file staging and are scheduled by the middleware, which might queue them for a long period. Because Aneka allows the reservation of nodes for thread execution connected to a given application, an explicit condition signaling execution failure due to a missing reservation credential has been included. This happens when a thread is routed to an execution node within a time frame in which only nodes with particular reservation credentials can be run.

The normal state transitions
Unstarted - [Started] - [Queued] – Running - Completed/Aborted/Failed.

An Aneka thread is found in the **Unstarted** condition at first. When the Start() method is invoked, the thread enters the **Started** state, from which it can go to the **StagingIn** state if there are files to upload for execution, or immediately to the **Queued** state. If an error occurs when uploading files, the thread fails and its execution finishes in the **Failed** state, which can also be reached if an exception occurs when executing Start(). Another possible outcome is the **Rejected** state, which occurs if the thread is launched with an incorrect reservation token. This is the last state, and it suggests that the execution will fail due to a lack of permissions. Once the thread is in the queue, if there is a free node where it can be executed, the middleware transports all of the object data and dependent files to the distant node and begins execution, setting the status to **Running**. If the thread throws an error or fails to create the anticipated output files, the execution is regarded as a failure, and the thread's end state is set to **Failed**. If the execution succeeds, the final state is changed to **Completed**.If there are any output files to retrieve, the thread state is set to **StagingOut** until the files are gathered and forwarded to their final destination and after the successful transfer, the thread state is set to **Completed**. If the developer terminates the application's execution or explicitly invokes the Abort() function at any time, the thread is aborted and its final state is set to **Aborted**.[5]

The following flow chart pictorially shows the AnekaThread instance state transitions.



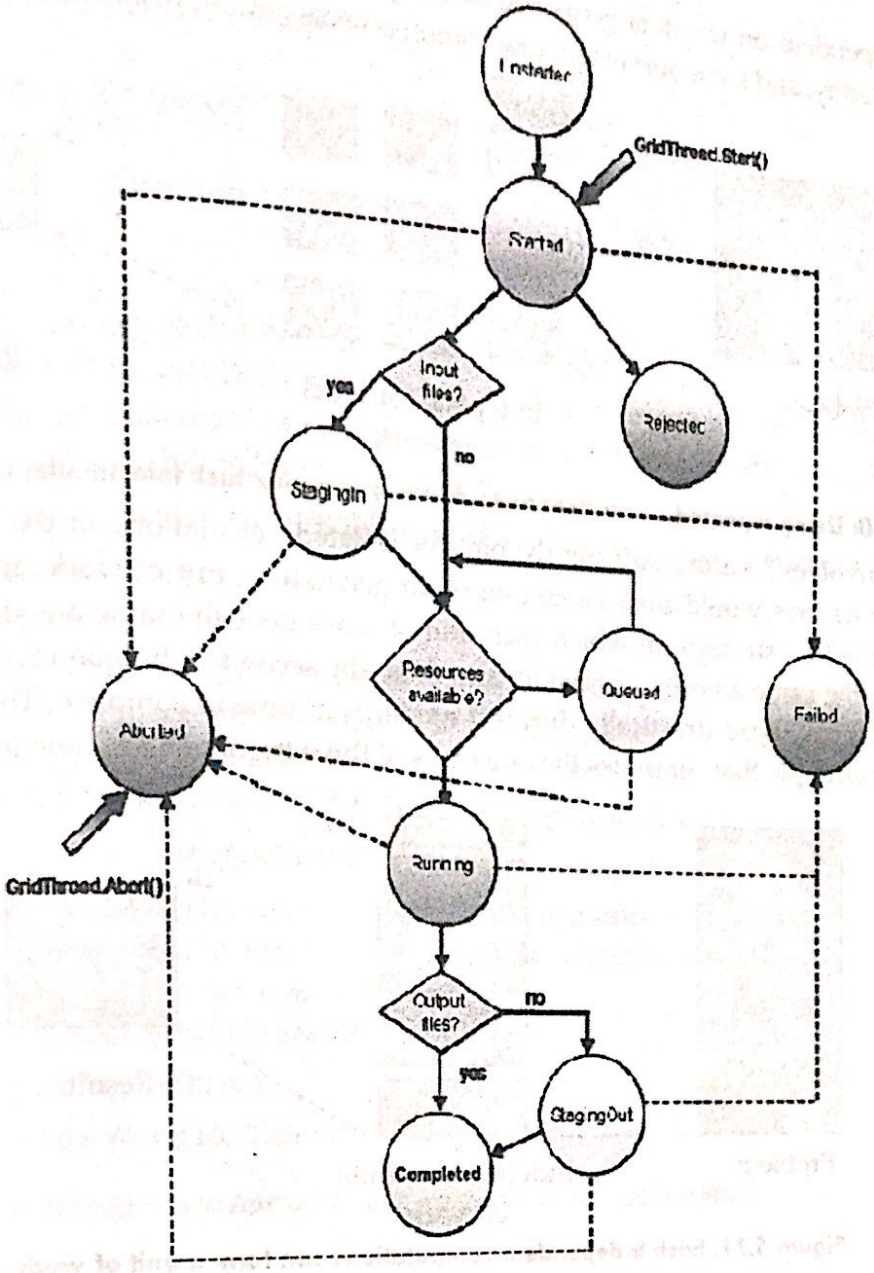


Figure 5.9: AnekaThread instance state transitions.

DEVELOPING PARALLEL APPLICATIONS WITH ANEKA THREADS

Creating parallel applications requires a thorough understanding of the problem and its logical structure. It is quite simple to approach the answer after you have worked this out.

Problem Decomposition

One of the most difficult issues in designing parallel applications is dividing a complex task into smaller units of work that can be handled concurrently on separate machines. Decomposing an issue may not appear to be obvious at first, but it is frequently a good idea, to begin with, a piece of paper.

There are two popular ways for problem decomposition:

- Identifying patterns of repetitive, but independent computations
- Identifying distinct, but independent computations

The first way, which is the most popular, is recognizing repeated calculations in the problem. In a sequential program, they are frequently implemented as for or while loops. Every loop iteration is therefore potentially a unit of work that may be computed independently of previous iterations. This methodology is used in the examples such as calculating the value of PI, Matrix Multiplication, etc.

If an iteration is dependent on the data generated in the prior iteration, the units of work can no longer be computed independently, and some sort of communication is necessary as illustrated in the following figure.

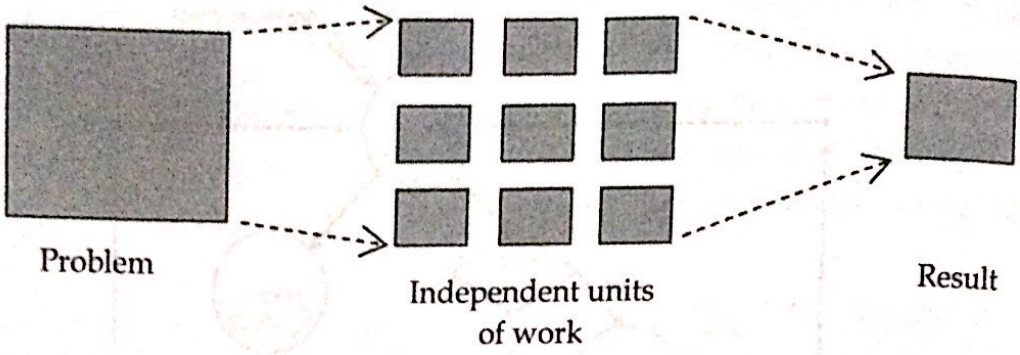


Figure 5.10: Using repeated computations to divide a complex task into smaller units of work

The second way involves locating sufficiently big yet isolated calculations in the issue. Each of these independent computations would then be combined to generate a unit of work suitable for concurrent execution. Unlike the first strategy, in which each unit of work does the same amount of calculation and thus takes roughly the same amount of time to complete, the second technique involves unique units of work, each of which may need drastically different amounts of time to complete. This method is used in the program of the example that computes the outcomes of three trigonometric functions.

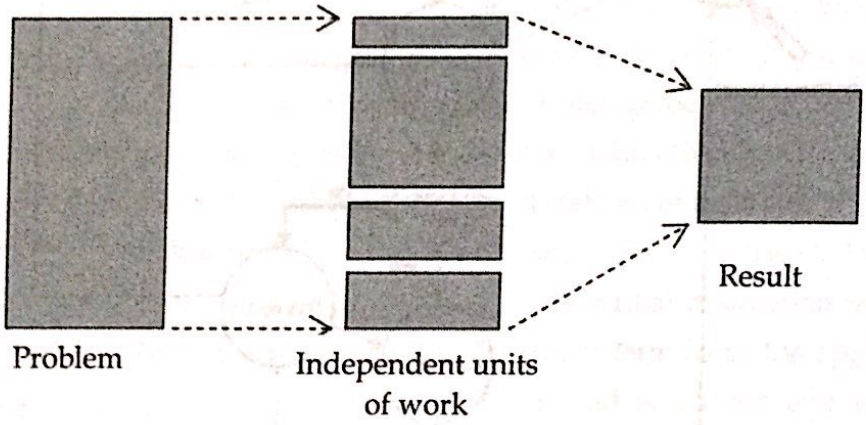


Figure *5.11*: Each independent computations can form a unit of work

CLASS DESIGN AND SERIALIZATION

The next stage is to build the classes after you have broken the problem into smaller pieces of work. Your class should ideally contain the data and methods needed to conduct the calculation on the remote node. To enable instances to be serialized and transported to the Aneka runtime for execution, this class must be annotated with the Serializable property.

A sample class that encapsulates the work done by an AnekaThread

```
[Serializable]
public class Work
{
        private int data1;
        private string data2;
        private int result;
        public int Result
        {
                get { returnthis.result; }
        }
        public Work(int data1, string data2)
```

```
{
        this.data1 = data1;
        this.data2 = data2;
}
public void Compute()
{
        // perform computation and store result in the variable 'result'
}
}
```

The method named Compute() that performs the actual computation must be assignable to a *ThreadStart* delegate. This method will be invoked by the Aneka runtime on the execution node. The results of the computation can be stored in an internal variable or data structure and can be obtained from the instance after it has been serialized and shipped back to the client.

The constructor for AnekaThread takes a ThreadStart delegate as one of its parameters:

```
public AnekaThread(ThreadStart start, AnekaApplication<AnekaThread,
ThreadManager> application)
```

Above mentioned code snippet is the constructor for AnekaThread, which takes a ThreadStart delegate and an instance of AnekaApplication as parameters. The delegate is a pointer to an instance method that will be executed on the remote node, and the AnekaApplication instance contains the required configuration to forward AnekaThreads to the Aneka runtime environment.

Creating and configuring an AnekaApplication instance

```
AnekaApplication<AnekaThread, ThreadManager> application = new
AnekaApplication<AnekaThread, ThreadManager>(configuration);
Configuration configuration = new Configuration();
configuration.SchedulerUri = schedulerUri;
```

Creating and starting an AnekaThread

```
Work work = new Work(10, "Hello World");

AnekaThread thread = new AnekaThread(work.Compute, application);

thread.start();
```

Example: Calculating Pi Using A Dartboard[5]

This example stated by Buyya et al. is a bit complicated, but it yields a more useful result. It explains how to utilize *AnekaThreads* to calculate the value of **pi**. This is similar to shooting darts at random spots on a dartboard. Using this strategy, each thread individually approximates the value of pi, and the cumulative average derived from all threads is utilized to estimate the value of pi. Increasing the number of AnekaThreads enhances pi accuracy.

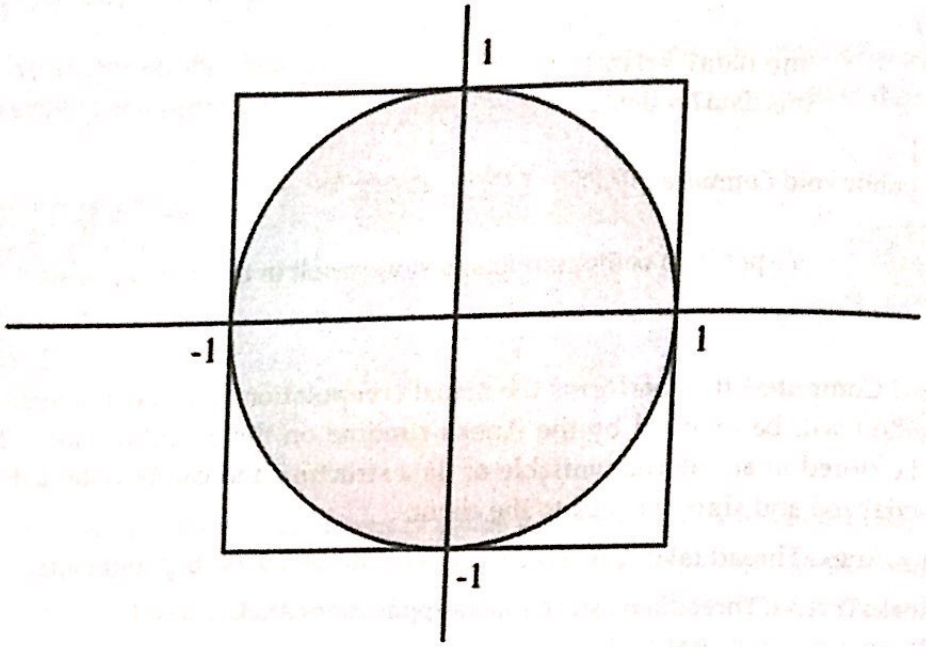


Figure 5.12: A circle inscribed in a square bounded by the coordinates (1,1), (1,-1), (-1,-1) and (-1,1)

Throwing darts at random spots within the square is used to calculate pi. A dart may land within or outside the circle, but within the square. A dart that lands inside the circle is called a hit.

The ratio of the area of the circle, πr^2, to the area of the square, $4r^2$, is $\pi/4$. The x-y coordinates of the darts are random numbers in the range [-1, 1]. The value of pi is given by the following equation:

$\pi/4$ = *(number of hits inside the circle) / (total number of throws)*

$\pi = 4$ * *(number of hits inside the circle) / (total number of throws)*

Each thread calculates pi by throwing darts a specified number of times. Increasing the number of threads maintains the work done by each thread while improving the overall accuracy of the cumulative average of pi.

```
/// <summary>
/// Class Dart. Represents a dart that is thrown for the given
/// number of iterations at random points on a dartboard.
/// </summary>
[Serializable]
public class Dart
{
    /// <summary>
    /// The number of iterations to throw the dart at random points.
    /// </summary>
    private int iterations;

    /// <summary>
    /// The approximation for the value of Pi.
    /// </summary>
    private double result;

    /// <summary>
    /// Gets or sets the approximation for the value of Pi.
    /// </summary>
    public double Result
    {
```

```
        get { return result; }
        set { result = value; }

    }

    /// <param name="iterations">The number of iterations to throw
    /// the dart at random points</param>
    public Dart(int iterations)
    {
        this.iterations = iterations;

    }
    /// <summary>
    /// Throws the dart at random points for the specified number of iterations.
    /// </summary>
    public void Fire()
    {
        int hit = 0;
        Random rand = new Random();
        for (int i = 0; i<this.iterations; i++)
        {
            double x = rand.NextDouble() * 2 - 1;
            double y = rand.NextDouble() * 2 - 1;
            if ((x * x) + (y * y) <= 1.0)
            {
            hit++;
            }
        }
        this.result = 4.0 * ((double)hit / (double)iterations);
    }
    }
    /// <summary>
    /// Class Dartboard. Represents a dartboard to which a collection of darts
    /// can be thrown. Each of the darts thrown in encapsulated in an AnekaThread
    /// instance and executed on the remote runtime environment.
    /// </summary>
public class Dartboard
{
    /// <summary>
    /// The application configuration
    /// </summary>
    private Configuration configuration;

    /// <summary>
    /// Creates an instance of Dartboard.
    /// </summary>
    /// <param name="schedulerUri">The uri to Aneka's scheduler</param>

    public Dartboard(Uri schedulerUri)
    {
        configuration = new Configuration();
        configuration.SchedulerUri = schedulerUri;
```

```
}

/// <summary>
/// Creates a list of AnekeThread instances for each of the darts
/// specified by <paramref name="noOfDarts"/>, submits them for execution
/// on the Aneka runtime, and composes the final result by calculating the
/// cumulative average value of pi.
/// </summary>
/// <param name="noOfDarts">The number of darts to throw. That is the number
/// of AnekaThread instances to execute on the remote runtime
/// environment</param>
/// <param name="iterations">The number of iterations for each of the darts</param>
/// <returns>The cumulative average of pi</returns>
public double ThrowDarts(int noOfDarts, int iterations)
{
        // Create application and computation threads
        AnekaApplication<AnekaThread, ThreadManager> application = new
        AnekaApplication<AnekaThread, ThreadManager>(configuration);
        IList<AnekaThread> threads = this.CreateComputeThreads(application,
        noOfDarts, iterations);

        // execute threads on Aneka
        this.ExecuteThreads(threads);

        // calcualate cumulative average of pi
        double pi = this.ComposeResult(threads);

        // stop application
        application.StopExecution();
        return pi;
}
/// <summary>
/// Creates AnekeThread instances for each of the specified number of
/// darts. These threads are initalized to execute the Dart.Fire() method on
/// the remote node.
/// </summary>
/// <param name="application">The AnekaApplication instance
/// containing the application configuration</param>
/// <param name="noOfDarts">The number of darts to throw. That is, the
/// number of instances of AnekaThread to create</param>
/// <param name="iterations">The number of iterations for each of the darts
/// thrown</param>
/// <returns>A list of AnekaThread instances</returns>

private IList<AnekaThread>CreateComputeThreads(
AnekaApplication<AnekaThread, ThreadManager> application,
int noOfDarts, int iterations)
{
        IList<AnekaThread> threads = new List<AnekaThread>();
        for (int x = 0; x <noOfDarts; x++)
```

```
        {
                Dart dart = new Dart(iterations);
                AnekaThread thread = new AnekaThread(dart.Fire, application);
                threads.Add(thread);
        }
        return threads;
}

/// <summary>
/// Executes the list of AnekaThread instances
/// on the Aneka runtime environment.
/// </summary>
/// <param name="threads">The list of AnekaThread
/// instances to execute</param>

private void ExecuteThreads(IList<AnekaThread> threads)
{
        for each (AnekaThread thread in threads)
        {
                thread.Start();
        }
}
/// <summary>
/// Composes the resulting value of pi by calculating the cumulative average
/// computed by each of the AnekaThread instances.
/// This method pauses until all threads have completed execution.
/// </summary>
/// <param name="threads">The list of instances that were submitted for
/// execution</param>
/// <returns>The average value of pi</returns>
private double ComposeResult(IList<AnekaThread> threads)
{
        // wait till all threads complete.
        foreach (AnekaThread thread in threads)
        {
                thread.Join();
        }
        double total = 0;
        foreach (AnekaThread thread in threads)
        {
                Dart dart = (Dart)thread.Target;
                total += dart.Result;
        }
        return total / threads.Count;
}
/// <summary>
/// The main entry point to the application.
/// </summary>
/// <param name="args">Currently requires no command line arguments. </param>
static void Main(string[] args)
```

```
    {
        Uri uri = new Uri("tcp://400w-ICT0217-09:9090/Aneka");
        Dartboard dboard = new Dartboard(uri);
        double pi = dboard.ThrowDarts(25, 3000);
        Console.WriteLine("Value of pi = " + pi);
        Console.ReadKey();
    }
}
```

The Dartboard class manages the execution of AnekaThread objects on distant nodes. The parameters for the ThrowDarts method are the number of darts and the number of repetitions per dart. For each dart, an AnekaThread is established and performed on a distant node for the specified number of repeats. Take note of how a List is utilized to keep track of the program's AnekaThread collection. The ExecuteThreads function iterates through this list to start all threads, and the ComposeResult method iterates through the list to connect all threads to the main application thread before computing the cumulative average for pi.

Output

Run 1:

Value of pi = 3.13797333333333

Run 2:

Value of pi = 3.15202666666667

Run 3:

Value of pi = 3.13845333333333

EXAMPLE: A PARALLEL MATH PROGRAM USING ANEKATHREADS[5]

The program below stated by Buyya et al. shows how to utilize AnekaThreads to do a simple mathematical computation. While the end product may be of little practical value, this application can be used to learn programming using AnekaThreads.

Consider the following mathematical equation:

$$p = \sin(x) + \cos(y) + \tan(z)$$

As these trigonometric functions are distinct operations, they may be run separately, and the results of these operations must be combined to obtain the final result after calculation.

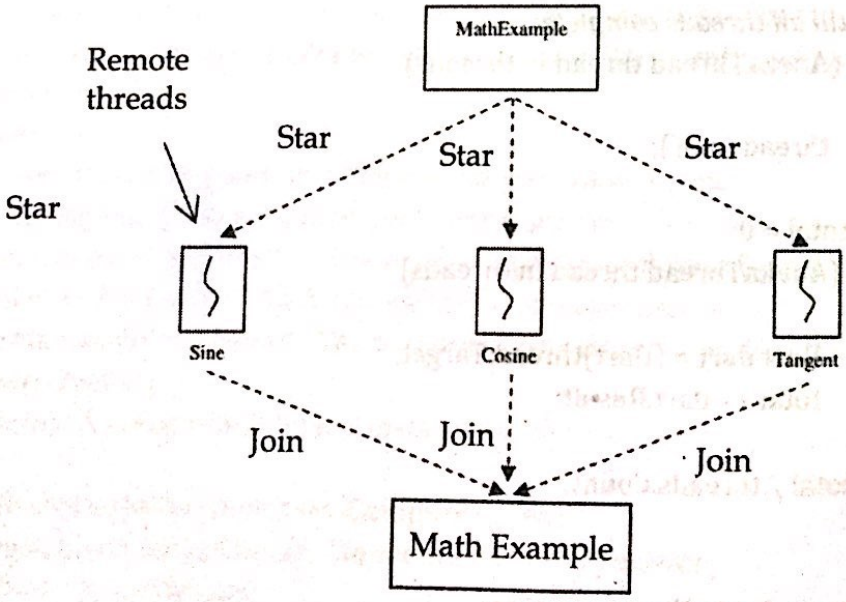


Figure 5.13: The flow control in an application using multiple distributed threads.

The class MathExample is a single-threaded client application that runs on the local machine. It generates three AnekaThreads, each to compute the sin, cos, and tan of a given angle. Each of these threads is then

dispatched to remote compute nodes for concurrent execution, by invoking the start operation on the threads. Although MathExample can now continue execution, it needs to wait until all three threads have completed before it can combine their results. To do this, it invokes the join operation on each of the threads which cause MathExample to block until the threads have completed their operation on the remote node.

The code segment mentioned below which shows the solution to the discussed problem.

```
/// <summary>
/// Class Sine. Computes the sin value of an angle.
/// </summary>
[Serializable]
public class Sine
{
        /// <summary>
        /// The angle in degrees
        /// </summary>
        private double angle;
        /// <summary>
        /// The sin value of the angle
        /// </summary>
        private double result;
        /// <summary>
        /// Gets or sets the sin value of the angle
        /// </summary>
        public double Result
        {
                get { return result; }
                set { result = value; }
        }
        /// <summary>
        /// Creates and instance of the Sine class
        /// </summary>
        /// <param name="angle">The angle in degrees to convert</param>
        public Sine(double angle)
        {
                this.angle = angle;
        }
        /// <summary>
        /// Computes the sin value of the specified angle
        /// </summary>
        public void Sin()
        {
                this.result = System.Math.Sin(Util.DegreeToRadian(this.angle));
        }
}
/// <summary>
/// Class Cosine. Computes the cos value of an angle.
/// </summary>
[Serializable]
public class Cosine
{
        /// <summary>
        /// The angle in degrees
        /// </summary>
        private double angle;
```

```
        /// <summary>
        /// The cos value of the angle
        /// </summary>
        private double result;
        /// <summary>
        /// Gets or sets the cos value of the angle
        /// </summary>
        public double Result
        {
                get { return result; }
                set { result = value; }
        }
        /// <summary>
        /// Creates and instance of Cosine class
        /// </summary>
        /// <param name="angle">The angle in degrees to convert</param>
        public Cosine(double angle)
        {
                this.angle = angle;
        }
        /// <summary>
        /// Computes the cos value of the specified angle
        /// </summary>
        public void Cos()
        {
                this.result = System.Math.Cos(Util.DegreeToRadian(this.angle));
        }
}

/// <summary>
/// Class Tangent. Computes the tan of an angle.
/// </summary>
[Serializable]
public class Tangent
{
        /// <summary>
        /// The angle in degrees
        /// </summary>
        private double angle;
        /// <summary>
        /// The tan value of the angle
        /// </summary>
        private double result;
        /// <summary>
        /// Gets or sets the tan value of the angle
        /// </summary>
        public double Result
        {
                get { return result; }
                set { result = value; }
        }
        /// <summary>
        /// Creates and instance of the Tangent class
        /// </summary>
        /// <param name="angle">The angle in degrees to convert</param>
```

```
public Tangent(double angle)
{       this.angle = angle;

}
/// <summary>
/// Computes the tan value of the specified angle
/// </summary>
public void Tan()
{       this.result = System.Math.Tan(Util.DegreeToRadian(this.angle));

}

}

/// <summary>
/// Class Util. A class for utility functions.
/// </summary>
public class Util
{
        /// <summary>
        /// Converts the angle in degrees to radians
        /// </summary>
        /// <param name="angle">The angle in degrees</param>
        /// <returns>The angle in radians</returns>
        public static double DegreeToRadian(double angle)
        {
                return System.Math.PI * angle / 180.0;
        }

}

/// <summary>
/// Class MathExample. Performs simple trigonometric calculations
/// on remote nodes using AnekaThreads
/// </summary>
public class MathExample
{
        /// <summary>
        /// The main entry point to the application
        /// </summary>
        /// <param name="args">Currently requires no arguments</param>
        static void Main(string[] args)
        {
                // Configuration for using runtime environment
                Configuration configuration = new Configuration();
                configuration.SchedulerUri = new Uri("tcp://400w-ICT0217-09:9090/Aneka");

                // Create AnekaApplication and remote threads
                AnekaApplication<AnekaThread, ThreadManager> application = new
                AnekaApplication<AnekaThread, ThreadManager>(configuration);

                Sine sine = new Sine(10);
                AnekaThreadsinThread = new AnekaThread(sine.Sin, application);

                Cosine cosine = new Cosine(10);
                AnekaThreadcosThread = new AnekaThread(cosine.Cos, application);
```

```
Tangent tangent = new Tangent(10);
AnekaThreadtanThread = new AnekaThread(tangent.Tan, application);

try
{
        // start executing all threads
        sinThread.Start();
        cosThread.Start();
        tanThread.Start();

        // wait until all threads complete
        sinThread.Join();
        cosThread.Join();
        tanThread.Join();

        // retrieve value for sin, cos and tan
        sine = (Sine)sinThread.Target;
        cosine = (Cosine)cosThread.Target;
        tangent = (Tangent)tanThread.Target;
}
finally
{
        // stop application
        application.StopExecution();
}
// compute sum
double sum = sine.Result + cosine.Result + tangent.Result;

// display result
Console.WriteLine("Sum = " + sum);
}
}
```

TASK PROGRAMMING

The Task Programming Methodology is a multithreaded high-level programming model. It is a broad topic of distributed system programming that encompasses multiple alternative models for architecting distributed systems, all of which are ultimately based on the same underlying abstraction: the task. In general, a task describes a program that may need input files and generate output files as a result of its execution. Applications are then made up of a series of tasks. These are submitted for execution, and their output data is gathered at the end. The task programming field encompasses how tasks are generated, the sequence in which they are executed, and whether or not they need data exchange to differentiate the application models.

Advantages of using the Task Programming Model

- No explicit threading, Tasks, are created not Threads.
- Schedules the Tasks to the processors so that the code scales to the number of available processors.

- Multiple algorithms written using the Task Programming Model can run at the same time without significant performance impact.
- Complex problems can be solved without requiring traditional synchronization tools such as Mutexes and Condition Variables.
- If such synchronization tools are not used, the function cannot be deadlocked.

TASK COMPUTING

The most obvious and frequent method for developing parallel and distributed computing applications is to manage an application in terms of tasks. A task denotes one or more processes which generate different outputs and may be separated as a distinct logical unit. A task is represented as a discrete unit of code, or a program, which may be split and performed in a remote runtime environment. Programs are the most frequent way to express tasks, particularly in the field of scientific computing, which has benefited from the power of distributed computing.

Multithreaded programming is primarily concerned to enabling parallelism inside a single system. Task computing distributes compute power by utilizing the computation capabilities of several computing nodes. As a result, the presence of a distributed infrastructure is crucial in this programming paradigm. Previously, the infrastructures used to accomplish tasks have been clusters, supercomputers, and computing grids. Now, cloud computing has been established as an appealing approach for obtaining massive computing capacity on demand for the execution of distributed applications. It is required to use appropriate middlewaresoftware layer that help to properly coordinate the use of multiple resources drawn from a data center or geographically distributed networked computers.

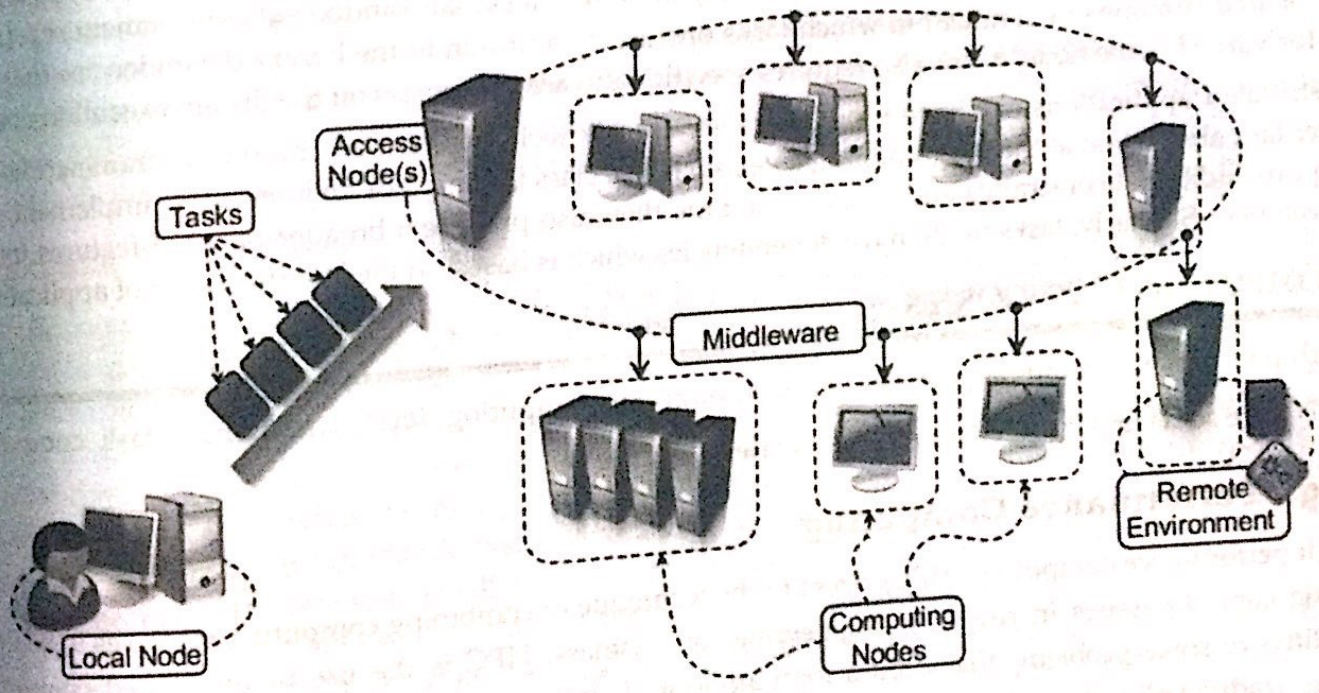


Figure 5.14: Task computing scenario

A user presents a group of tasks to the middleware's access point(s)/node(s), which will handle task scheduling and monitoring. Each computing resource offers a suitable runtime environment, which may differ from one implementation to the next (a simple shell, a sandboxed environment, or a virtual machine). The APIs supplied by the middleware, whether a Web or programming language interface, are often used for task submission. Appropriate APIs are also supplied to monitor task status and gather data when they are completed.

There are several models of distributed applications that fall under task computing. Despite this diversity, a set of common operations that the middleware needs to allow the design and execution of task-based applications can be identified as:[5]

- Coordination and scheduling of activities for execution on a collection of distant nodes
- Transferring programs to remote nodes and maintaining their dependencies
- Creating an environment for distant node task execution
- Monitoring each task's execution and alerting the user of its status
- Access to the task's output

What is a task?

A task is a broad abstraction that identifies a program or a group of programs that comprise a computing unit of a distributed application and provide a physical output. A task is an application component that may be conceptually separated and run separately.Distributed applications are made up of tasks, the aggregate execution, and interrelationships of which constitute the application's nature. Different elements can be used to illustrate tasks such as a shell script that combines the execution of many apps, a single program, and a piece of Java/C++/.NET code that runs in the context of a specified runtime environment.

> According to the Task Model, an application is defined as a collection of tasks. Tasks are self-contained work units that the Scheduler can perform in any sequence. A task in the Task Model contains all of the components necessary for its execution on a grid node. When the distributed application comprises a group of separate jobs, they are executed on a grid, outcomes are gathered and combined, the Task Model is the ideal approach to utilize in such a scenario.

Commonly, a task is defined by input files, executable code such as Java or .Net programs or shell scripts, and output files. In many situations, the operating system or a similar sandboxed environment represents the shared runtime environment in which tasks operate. In addition to the library dependencies that may be forwarded to the node, a task also requires specific software appliances on the distant execution nodes.

Distributed applications can have multiple constraints as well. Distributed computing frameworks that offer task abstraction at the programming level, such as a class to inherit or an interface to implement, may require additional constraints and at the same time they also provide a broader range of features for the developers. Similarly, tasks might have dependencies which is based on the specific model of application.

COMPUTING CATEGORIES

High-performance computing (HPC), high-throughput computing (HTC), and many-task computing (MTC) are some important computing categories.

High-Performance Computing

High performance computing (HPC) refers to the technique of combining computing resources in order to tackle complex issues in research, engineering, or business. HPC is the use of distributed computing facilities to solve problems that need a high amount of processing power. Supercomputers and clusters have traditionally been built to enable HPC applications that are built to address challenging problems.Data drives revolutionary scientific discoveries, game-changing technologies, and improves the quality of life for billions of people across the world. HPC lays the groundwork for scientific, industrial, and societal progress. The size and volume of data that companies work with is rising quickly as technologies such as the Internet of Things (IoT), artificial intelligence (AI), and 3-D imaging advance. The capacity to analyze data in real time is critical for many purposes, including broadcasting a live sporting event, following a growing storm, testing new goods, and evaluating market patterns, etc.Organizations

must have lightning-fast, highly dependable IT infrastructure to process, store, and analyze vast volumes of data in order to stay ahead of the competition.

An HPC cluster is made up of hundreds or thousands of compute machines that are linked together through a network. Each server is referred to as a node. Each cluster's nodes operate in parallel with one another, increasing processing speed and delivering high-performance computing.The normal profile of HPC applications is made up of a significant number of compute-intensive activities that must be completed in a short period. Parallel and tightly connected processes are widespread, requiring the use of low-latency interconnection networks to reduce data interchange time. The metrics used to assess HPC systems are floating-point operations per second (FLOPS), now tera-FLOPS, or even peta-FLOPS, which identify the number of floating-point operations per second that a computing machine can execute.

High-Throughput Computing

For many experimental scientists, scientific advancement and research quality are inseparably tied to computational throughput. In other words, most scientists are more interested with how many floating-point operations they can extract from their computing environment each month or year than with how many such operations the environment can offer them per second or minute.High-throughput computing (HTC) is the utilization of distributed computing resources for applications that require a huge amount of processing power over a lengthy period. HTC relies on the efficient control and utilization of all available computer resources.HTC systems must be strong and dependable for lengthy periods. Traditionally, heterogeneous computing grids (clusters, workstations, and volunteer PCs) have been utilized to support HTC. HTC applications are made up of a large number of tasks whose execution may take weeks or months.Scientific simulations and statistical analysis are two examples of such applications. It is quite common to have independent tasks that can be scheduled in distributed resources because they do not need to communicate. HTC systems track success in terms of tasks performed every month.The primary problem that a typical HTC environment faces is maximizing the number of resources available to its consumers. The main barrier that such an ecosystem must overcome in order to enlarge the pool of resources from which it may draw is distributed ownership of computer resources.

Many-Task Computing

Many-task computing (MTC) is a parallel computing technique in computational science that seeks to bridge the gap between two computing paradigms: high-throughput computing (HTC) and high-performance computing (HPC).The many-task computing (MTC) concept has lately gained popularity and applies to a wide range of applications.MTC is similar to HTC, but it focuses on the usage of a large number of computing resources in a short amount of time to complete a large number of computational tasks. In a nutshell, MTC refers to high-performance calculations that include numerous independent activities that are linked together via file system operations. MTC is distinguished by the diversity of tasks, which may be radically different: small or big tasks, single-processor, or multiprocessor, compute-intensive or data-intensive, static, or dynamic, homogeneous, or heterogeneous are all possibilities.

MTC applications have a general profile that includes loosely coupled applications that are generally communication-intensive but are not naturally expressed using the message-passing interface found in HPC, drawing attention to the many computations that are heterogeneous but not perfectly parallel. Given the enormous number of jobs that frequently comprise MTC applications, MTC may be supported by any distributed facility with a significant availability of computing elements. Supercomputers, massive clusters, and emerging cloud infrastructures are examples of such facilities.

HPC vs. HTC vs. MTC

HPC activities need enormous amounts of computing power for short periods of time, whereas HTC workloads require large amounts of computing power for considerably longer periods of time (months

and years, rather than hours and days). The HTC is more concerned with operations per month or ye than with activities per second. As a result, the HTC field is more concerned with how many projects c be performed over a long period of time than how quickly. HTC is a computing paradigm that emphasi on the efficient execution of many loosely linked activities.HPC systems are often focused on tightly link parallel workloads.HTC systems are self-contained, sequential tasks that may be scheduled on a variet computer resources across several administrative boundaries. HTC systems do this through the use o different grid computing technologies and approaches.MTC's mission is to bridge the gap between HT and HPC. MTC is similar to HTC, but it differs in its emphasis on employing a large number of computi resources in a short amount of time to complete a large number of computational tasks, including bo dependent and independent activities. In contrast to operations per month, the major metrics are measured in seconds, such as FLOPS, tasks/s, MB/s I/O rates, and so on.

FRAMEWORKS FOR TASK COMPUTING

Several frameworks are available to allow the execution of task-based applications on distributed computing resources such as clouds. All of these systems have architectures that are comparable to the generic task computing scenario discussed previously. Condor, Globus Toolkit, Oracle Grid Engine (*previously* Sun Grid Engine), BOINC, Nimrod/G, and Aneka, etc. are some popular frameworks that support task-computing. These frameworks consist of two parts: a scheduling node and worker nodes. The system components' organization may differ. Multiple scheduling nodes can be grouped in hierarchical frameworks. This arrangement is fairly common in computing grid middleware, which utilizes a range of distributed resources from one or more companies or sites. Each of these locations may have its scheduling engine, particularly if the system contributes to the grid while simultaneously serving local consumers.

Condor is the most commonly used middleware for managing clusters, idle workstations, and cluster collections. Condor-G is a Condor variant that allows for integration with grid computing resources such as those handled by Globus. Condor supports common features of batch-queuing systems along with the capability to checkpoint jobs and manages overload nodes. It has a robust task resource-matching mechanism that only schedules jobs on resources that have a suitable runtime environment.Condor is capable of handling both serial and parallel workloads on a wide range of resources. Hundreds of organizations in business, government, and academia utilize it to manage infrastructures ranging from a few dozen to thousands of workstations.

Globus Toolkit is the software toolkit developed by the Globus Alliance to provide a collection of tools for Grid Computing middleware based on common grid APIs.It used to provide a broad set of tools for sharing computing power, databases, and other services across corporate, institutional, and geographic boundaries without sacrificing local autonomy, but the Globus Toolkit is no longer available as it retired in 2018.

Oracle Grid Engine, previously Sun Grid Engine (SGE), is a workload and distributed resource management middleware. It was originally designed to assist the execution of tasks on clusters, but it has since gained additional features and is now capable of managing heterogeneous resources and serving as grid computing middleware. It can run parallel, serial, interactive, and parametric jobs and has extensive scheduling capabilities such as budget-based and group-based scheduling, deadline-based and custom-policy scheduling, and reservation.

The Berkeley Open Infrastructure for Network Computing (BOINC) is a volunteer and grid computing architecture that enables the conversion of desktop workstations into volunteer computing nodes that may be used to perform jobs when they become inactive. BOINC is made up of two major parts: the BOINC server and the BOINC client. BOINC server is the central node that manages all available resources and schedules jobs; BOINC client is the software component that is installed on desktop computers and generates the BOINC execution environment for task submission. BOINC allows job checkpointing and

duplication because of the unpredictability of BOINC clients. Even though they are primarily geared for volunteer computing, BOINC systems may be easily configured to offer more steady assistance for job execution by establishing computing grids with dedicated computers. To use BOINC, you must first build an application project and while installing BOINC clients, users can select the application project to which they want to give their computer's CPU cycles. Several projects are now functioning on the BOINC infrastructure, spanning from health to astronomy to cryptography.

Nimrod/G is a tool for automated modeling and execution of parameter sweep applications on global computational grids. It offers a straightforward declarative parametric modeling language for expressing parametric experiments. A domain expert may simply build a plan for a parametric experiment and then utilize the Nimrod/G system to deliver tasks for execution on distributed resources. Over the years, it has been employed for a wide range of applications, spanning from quantum chemistry to policy and environmental effects.

Task-Based Application Models

There are numerous methods for constructing distributed applications that are based on the concept of the 'task' as the fundamental unit. The models are distinguished based on the approach in which tasks are created, the relationship they have with one another, and the presence of dependencies or other conditions that must be satisfied, such as a specified set of services in the runtime environment, which have to be made.

Most common and popular task-based models

- Embarrassingly parallel applications (EPA)
- Parameter sweep applications (PSA)
- Message Passing Interface applications (MPIA)
- Workflow applications with task dependencies (WATD)

Embarrassingly parallel applications (EPA)

In the field of parallel computing, embarrassingly parallel applicationsare such applications that can be divided into several parallel tasks with little or no effort. There is slight or no reliance or necessity for communication or outcomes between those simultaneous processes. These are different from distributed computing problems that need communication between tasks. Because of the nature of embarrassingly parallel algorithms, they are particularly suited to big, internet-based distributed platforms and are not affected by parallel slowness. Image and video rendering, evolutionary optimization, and model forecasting are some applications that can be computed using this model.

As mentioned earlier, in embarrassingly parallel applications, tasks do not need to communicate, there is a lot of flexibility in how to schedule them. Tasks can be completed in any sequence, and there is no requirement of completing all tasks at the same time. As a result, scheduling these applications is simpler, with the primary focus being the appropriate mapping of tasks to available resources. Globus Toolkit, BOINC, and Aneka are frameworks and tools that facilitate embarrassingly parallel applications.

Parameter sweep applications (PSA)

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

PSA is a special type of EPA in which the tasks are similar and just difference is in the precise parameter utilized to perform such tasks. Template tasks and a collection of parameters distinguish PSA. The template task specifies the operations that will be carried out on the remote node during task execution. The template task is parametric, and the parameter set indicates the variable combination whose assignments specialize the template task into a specific instance.

As the tasks that comprise the program may be run independently of each other, and distributed computing system that supports embarrassingly parallel applications may also allow the execution of parameter sweep applications. The main distinction is that the task that will be carried out are formed by iterating over all possible and permissible combinations of the parameters. This operation can be accomplished directly by frameworks or by tools included in distributed computing middleware. Frameworks that offer parameter sweep program execution frequently provide a range of helpful commands for manipulating or interacting with files. The following are some of the most commonly used commands:

- Execute - Executes a program on the remote node.
- Copy - Copies a file to/from the remote node.
- Substitute - Substitutes the parameter values with their placeholders inside a file.
- Delete - Deletes a file.

Aneka provides client-based tools for visually composing a template task, defining parameters, and iterating over all possible combinations of such parameters. Similarly, Nimrod/G is natively designed to support the execution of parameter sweep applications. Various applications such as evolutionary optimization algorithms, weather-forecasting models, computational fluid dynamics applications, Monte Carlo methods, etc. that are largely from the scientific computing domain fall under PSA.

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

```
individuals = {100, 200, 300, 500, 1000}
generations = {50, 100, 200, 400}
foreach indiv in individuals do
foreach generation in generations do
        task = generate_task(indiv, generation)
        submit_task(task)
```

The example defines a bidimensional domain comprised of discrete variables which are iterated over each combination of two variables - individuals and generations. This will generate 20 tasks composing the application. The generate_task method creates the task instance by substituting the values of *indiv* and *generation* to the corresponding variables in the template definition. Likewise, another method submit_task, which is specific to the middleware, submits the task.

MPI applications

The Message Passing Interface (MPI) is a de facto standard for creating parallel applications that interact by efficient message-passing. This standard can be used on current HPC clusters. The issue with a computer cluster is that although some CPUs share a memory, others have a dispersed memory architecture and are only connected by a network. Interface specifications have been defined and implemented for C/C++ and Fortran.

Developers these days can employ distributed memory, shared memory, or a hybrid system of the two, all with the power of MPI. There have been several MPI libraries available through the years, and some of the most popular libraries include MPICH, MVAPICH, Open MPI, and Intel MPI.

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

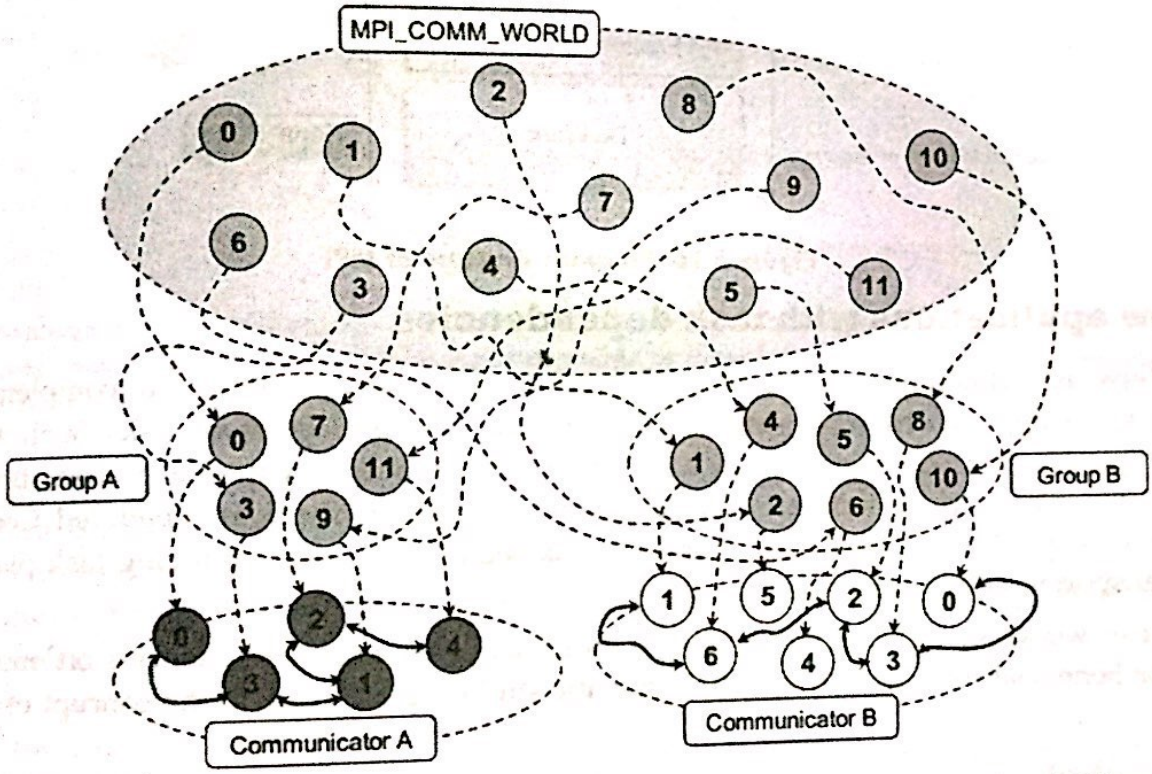


Figure 5.15: MPI reference scenario - A distributed application in MPI is made up of a set of MPI processes that run in parallel in an MPI-compatible distributed infrastructure.

MPI apps that use the same MPI runtime are automatically assigned to a global group named MPI_COMM_WORLD. In that group, the distributed processes have a unique identifier that allows the MPI runtime to localize and address them. Specific groups can also be formed as subsets of this global group. Individual MPI process has its rank within the associated group and the rank is a unique identifier that allows processes to communicate within the group. Communication is enabled through a communicator component that may be customized for each group.

To develop an MPI application, the code for the MPI process that will be performed in parallel must be defined. The parallel part of code is readily recognizable by two actions that set up and shut down the MPI environment, respectively. All MPI functions can be used to transmit or receive messages in either asynchronous or synchronous mode in the code block specified by these two actions.

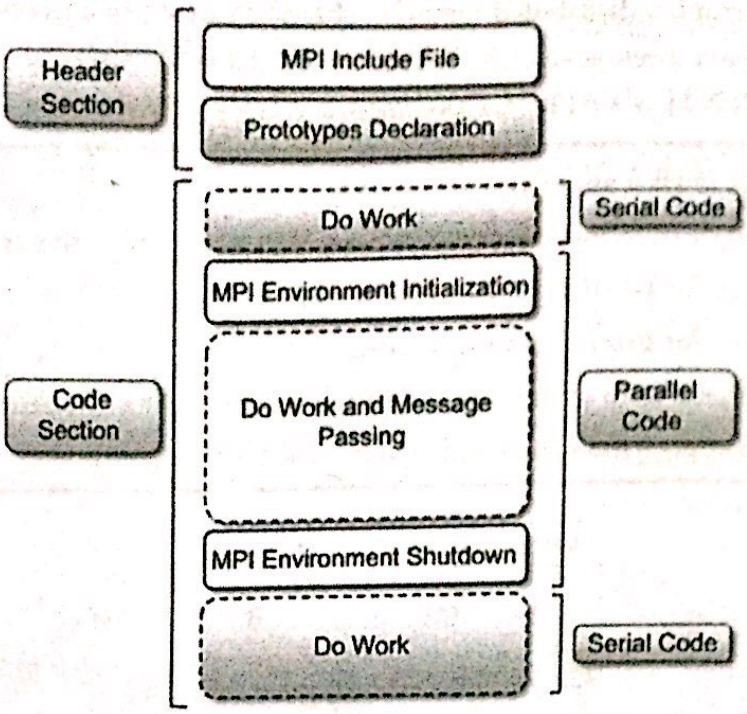


Figure 5.16: Program structure of MPI

Workflow applications with task dependencies

The Workflow is defined as a collection of services that work together to complete a business process.Workflow applications are distinguished by a group of interdependent tasks. Such dependencies, which are typically data dependencies, such as, the output of one activity is a precondition of another activity, determine how applications are scheduled along with where there are scheduled. Concerns in this scenario are connected to establishing a workable task sequencing and optimizing task placement such that data transportation is minimized.

The concept of workflow as a systematic execution of activities with dependencies on one another has proven to be beneficial for articulating many scientific studies, giving rise to the concept of the scientific workflow.

> "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." ↳Workflow Management Coalition

Numerous scientific experiments are made up of problem-solving components that, when combined in a precise order, define the essence of the experiment. When such studies demonstrate inherent parallelism and require the execution of a large number of operations or the handling of massive amounts of data, it makes sense to run them on a distributed infrastructure for the best performance. A scientific workflow often includes data management, analysis, simulation, and middleware to aid in workflow execution.

A scientific workflow is illustrated using a Directed Acyclic Graph (DAG), which depicts the relationships between tasks or procedures. The DAG nodes represent the tasks to be accomplished in a workflow application; the arcs linking those task nodes define task dependencies and data channels that link the tasks. The most frequent dependence realized by a DAG is data dependence. This dependence is depicted as an arc that starts at the node that identifies the first task and ends at the node that identifies the second task.

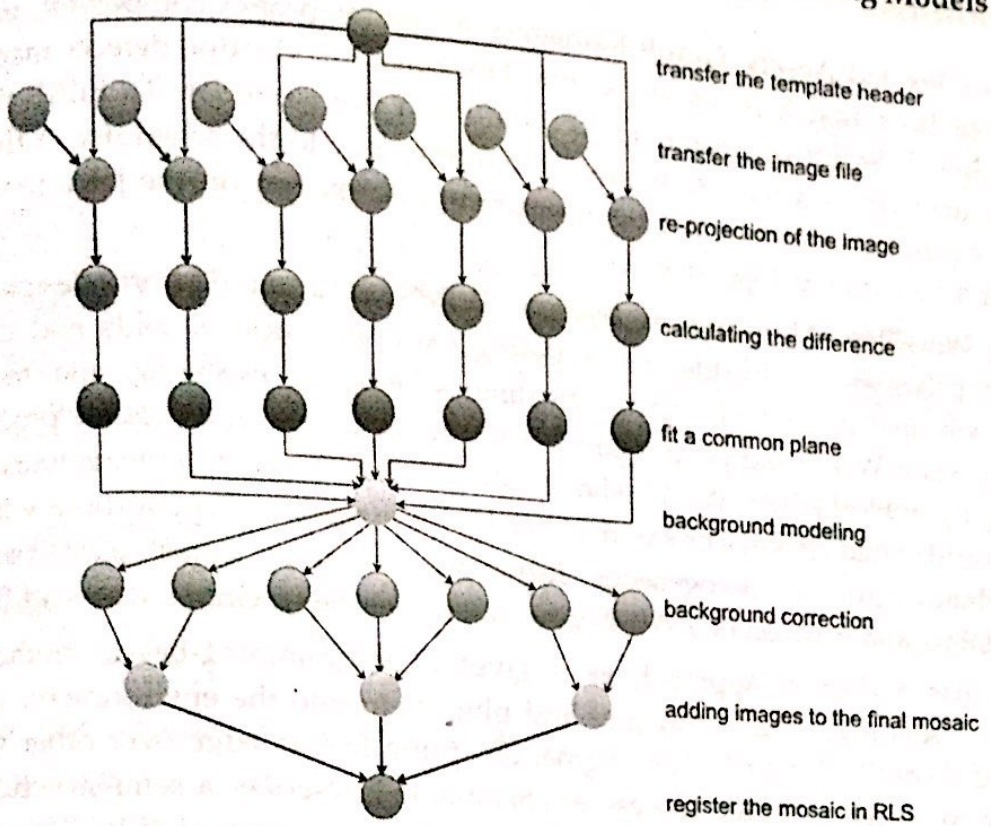


Figure 5.17: Montage workflow. Montage is a toolset for combining photos into mosaics; it assists astronomers in integrating photos acquired from several telescopes or points of view into a cohesive picture. The toolkit includes many programs for modifying photos and assembling them; some of the programs do background reprojection, perspective alteration, and brightness and color correction.

Looking at figure 5.17, you can see the overall method for creating a Mosaic. The labels on the right side of the graphic illustrate the several actions that must be completed to create a mosaic. In this case, seven large pictures are used. As numerous activities must be done concurrently, the mosaic generation process might benefit from the capability of distributed infrastructure. Each image file must go through the procedure of image file transfer, reprojection, difference computation, and common plane placement. As a result, for these activities, each of the photos may be handled in parallel. This emphasizes the significance of a distributed infrastructure in workflow execution.

WORKFLOW TECHNOLOGIES

Workflow technology is common ground for the development of various types of workflow-based applications. Computing for business workflow is specified as service compositions, and there are specialized languages and standards for workflow definition, such as Business Process Execution Language (BPEL). There is no common ground for describing processes in scientific computing, although multiple methods and workflow languages coexist. Kepler, DAGMan, Cloudbus Workflow Management System, and Offspring are some of the most relevant technologies for creating and implementing workflow-based systems.

1. **Kepler** is an open-source scientific workflow engine that was created via the collaboration of various research initiatives which offers a robust foundation for creating dataflow-oriented processes. Kepler offers a design environment based on the concept of actors, which are reusable and independent computing blocks such as Web services and database requests. Ports allow actors to communicate with one another. An actor consumes data from input ports and outputs results to output ports. Kepler's uniqueness stems from its ability to segregate data flow across components from the coordination logic which executes the workflow.As a result, Kepler allows many models for the same operation, such as synchronous and asynchronous models. XML is used to convey the process **definition.**

2. **DAGMan** (Directed Acyclic Graph Manager), a Condor project component, is an addition to the Condor scheduler that handles work interdependencies. Condor detects machines for program execution but does not provide job scheduling in a precise order. To fulfill that gap of Condor, DAGMan functions as a meta scheduler, submitting jobs to the scheduler in the correct sequence. DAGMan's input is a simple text file containing information on the jobs, references to their job submission files, and job dependencies.

3. **Cloudbus Workflow Management System** (WfMS) is a middleware designed to manage big application processes on distributed computing platforms such as grids and clouds. It comprises software tools that assist end-users in creating, scheduling, executing, and monitoring workflow applications via a Web-based control panel. The panel allows you to upload processes or create new ones using a graphical editor. WfMS relies on the Gridbus Broker to execute workflows. The Gridbus Broker is a grid/cloud resource broker that enables the execution of applications with quality-of-service (QoS) attributes across a heterogeneous distributed computing infrastructure, such as Linux-based clusters, Globus, and Amazon EC2.WfMS uses an XML for the specification of workflows.

4. **Offspring** has a distinct approach as it gives a programming-based method for developing workflows. Users may create strategies and plug them into the environment, which will execute them using a specified distribution engine. Offspring has an edge over other solutions in that it allows you to design dynamic processes. This method describes a semi-structured workflow that can vary its behavior at runtime based on the completion of specified tasks. This enables developers to control task dependencies dynamically during runtime rather than statically. Offspring is compatible with any distributed computing middleware capable of managing a simple bag-of-tasks application. It features a simulated distribution engine for testing tactics during development and has native integration with Aneka. Offspring does not employ any XML specifications because it delivers processes in the form of plug-ins.

ANEKA TASK-BASED PROGRAMMING[5]

Aneka supports different task-based programming via its Task Programming Model, which is the Aneka framework's core foundation for enabling the execution of bag-of-tasks applications. The Aneka abstraction is used to realize task programming. Task programming is accomplished through the abstraction of the *Aneka.Tasks.ITask*.Using this abstraction as a foundation, support for legacy application execution, parameter sweep apps, and workflows have been implemented into the framework.

Aneka Task Programming Model

The Task Programming Model provides an easy-to-use framework for rapidly constructing distributed applications on top of Aneka. It provides a minimal set of APIs, most of which are centered on the *Aneka.Tasks.ITask* interface. Interface, together with the services facilitates task execution in the middleware.

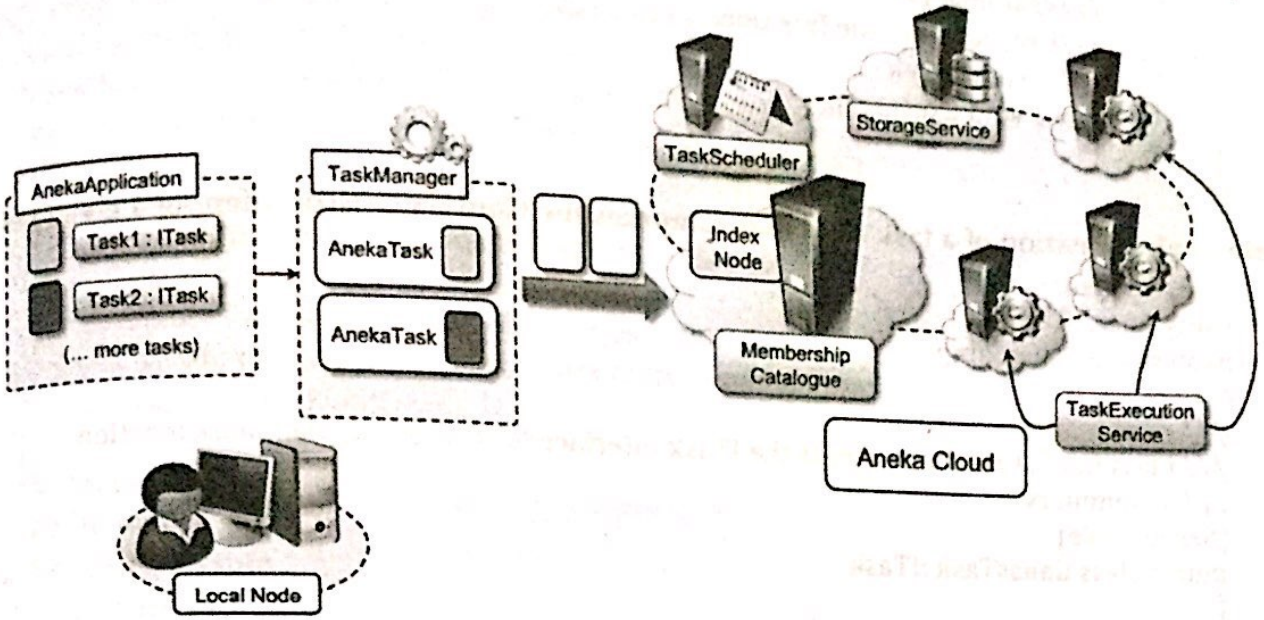


Figure 5.18: Overall view of the components of the Task Programming Model and their roles during application execution.

Distributed applications are created as ITask instances, and the combined execution of these instances forms a running application. The tasks and numerous required dependencies, such as data files and libraries, are gathered and handled by the AnekaApplication class, which supports task execution. The client-side view of a task-based application is made up of two additional components: AnekaTask and TaskManager. AnekaTask is the runtime wrapper that Aneka uses to represent a task within the middleware; and the TaskManager is the underlying component that communicates with Aneka, submits tasks, monitors their execution, and gathers results.

Four services in the middleware coordinate their actions to perform task-based applications. MembershipCatalogue, TaskScheduler, ExecutionService, and StorageService are examples.MembershipCatalogue is the cloud's primary access point, and it serves as a service directory for locating the TaskScheduler service, which is in charge of controlling the execution of task-based applications. Its primary role is to assign task instances to resources that use the ExecutionService for task execution and task state monitoring. If the application needs data transfer support in the form of data files, input files, or output files, and accessible StorageService will be used as a staging facility.

Developing applications with the task model

- Defining classes implementing the ITask interface
- Creating a properly configured AnekaApplication instance
- Creating ITask instances and wrapping them into AnekaTask instances
- Executing the application and waiting for its completion

The Aneka.Tasks namespace contains almost all of the client-side functionalities required for developing task-based applications. The ITask interface, which exposes only one method: Execute, is the most fundamental component for building tasks. The method is invoked to have the job executed on the remote node.

ITask interface

```
namespace Aneka.Tasks
{
///<summary>
///Interface ITask. Defines the interface for implementing a task.
///</summary>
public interface ITask
{
```

```
        ///<summary>
        ///Executes the sine function.
        ///</summary>
        public void Execute();

    }
    }
```

A simple implementation of a task class that computes the Gaussian distribution for a given point x

```
    using System;
    using Aneka.Tasks;
    namespace GaussSample
    {
    /// <summary>
    /// Class GaussTask. Implements the ITask Interface for computing the Gauss function.
    /// </summary>
    [Serializable]
    public class GaussTask :ITask
    {
            /// <summary>
            /// Input value.
            /// </summary>
            private double x;

            /// <summary>
            /// Gets the input value of the Gauss function.
            /// </summary>
            public double X { get { return this.x; } set { this.x = value; } }

            /// <summary>
            /// Result value.
            /// </summary>
            private double y;

            /// <summary>
            /// Gets the result value of the Gauss function.
            /// </summary>
            public double Y { get { return this.y; } set { this.y = value; } }

            /// <summary>
            /// Executes the Gauss function.
            /// </summary>
            public void Execute() {
                    this.y = Math.Exp(-this.x*this.x);
            }
    }
    }
```

Wrapping an ITask instance into an AnekaTask and adding input and output files specific to a given task

```
    // create a Gauss task and wraps it into an AnekaTaskinstance
    GaussTaskgauss = new GaussTask();
    AnekaTask task = new AnekaTask(gauss);

    // add one input and one output files
    task.AddFile("input.txt", FileDataType.Input, FileAttributes.Local);
    task.AddFile("result.txt", FileDataType.Output, FileAttributes.Local);
```

Static task submission

```
    // get an instance of the Configuration class from file
    Configuration conf = Configuration.GetConfiguration("conf.xml");
```

```
// specify that the submission of task is static (all at once)
conf.SingleSubmission = true;
AnekaApplication<AnekaTask, TaskManager> app =
new AnekaApplication<Task,TaskManager>(conf);
for(int i=0; i<400; i++)
{
        GaussTaskgauss = new GaussTask();
        gauss.X = i;
        AnekaTask task = new AnekaTask(gauss);
        // add the task to the bag of work units to submit
        app.AddWorkunit(task);
}
// submit the entire bag
app.SubmitExecution();
```

Dynamic Task Submission

```
///<summary>
///Main method for submitting tasks.
///</summary>
public void SubmitApplication()
{
        // get an instance of the Configuration class from file
        Configuration conf = Configuration.GetConfiguration("conf.xml");
        // specify that the submission of task is dynamic
        conf.SingleSubmission = false;
        AnekaApplication<AnekaTask, TaskManager> app =
        new AnekaApplication<Task,TaskManager>(conf);
        // attach methods to the event handler that notify the client code
        // when tasks are completed or failed
        app.WorkUnitFailed +=
        new EventHandler<WorkUnitEventArgs<AnekaTask>>(this.OnWorkUnitFailed);
        app.WorkUnitFinished +=
        new EventHandler<WorkUnitEventArgs<AnekaTask>>(this.OnWorkUnitFinished);
        for(int i=0; i<400; i++)
        {
                GaussTask gauss = new GaussTask();
                gauss.X = i;
                AnekaTask task = newAnekaTask(gauss);
                // add the task to the bag of work units to submit
                app.AddWorkunit(task);
        }
        // submit the entire bag
        app.SubmitExecution();
        }
        /// <summary>
        /// Event handler for task failure.
        /// </summary>
        /// <param name="sender">Event source: the application instance.</param>
        /// <param name="args">Event arguments.</param>
        private void OnWorkUnitFailed(object sender, WorkUnitEventArgs<AnekaTask>args)
        {
                // Do nothing, we are not interested in task failure at the moment
                // just dump to console the failure.
```

```
                    if (args.WorkUnit != null)
                    {
                    Exception error = args.WorkUnit.Exception;
                    Console.WriteLine("Task {0} failed – Exception: {1}",
                    args.WorkUnit.Name, (error == null ? "[Not given]" :error.Message);
            }
    }
    /// <summary>
    /// Event handler for task completion.
    /// </summary>
    /// <param name="sender">Event source: the application instance.</param>
    /// <param name="args">Event arguments.</param>
    private void OnWorkUnitFinished(object sender, WorkUnitEventArgs<AnekaTask>args)
    {
            // if the task is completed for sure we have a WorkUnit instance
            // and we do not need to check as we did before.
            GaussTask gauss = (GaussTask) args.WorkUnit.Task;
            // we check whether it is an initially submitted task or a task
            // that we submitted as a reaction to the completion of another task
            if (task.X – Math.Abs(task.X) == 0)
            {
                    // ok it was an original task, then we increment of 0.5 the
                    // value of X and submit another task
                    GaussTaskfraq = GaussTask();
                    fraq.X = gauss.X + 0.5;
                    AnekaTask task =new AnekaTask(fraq);
                    // we call the ExecuteWorkUnit method that is used
                    // for dynamic submission
                    app.ExecuteWorkUnit(task);
            }
            Console.WriteLine("Task {0} completed – [X:{1},Y:{2}]",
            args.WorkUnit.Name, gauss.X, gauss.Y);
    }
```

Aneka application configuration file

```
<?xml version="1.0"encoding="utf-8"?>
<Aneka>
        <UseFileTransfer value="true" />
        <Workspace value="." />
        <SingleSubmission value="false" />
        <ResubmitMode value="Manual" />
        <PollingTime value="1000" />
        <LogMessages value="true" />
        <SchedulerUri value="tcp://localhost:9090/Aneka"/>
        <UserCredential type="Aneka.Security.UserCredentials" assembly="Aneka.dll">
                <UserCredentials username="Administrator" password=""/>
        </UserCredential>
        <Groups>
                <Group name="StorageBuckets">
                        <Groups>
                                <Group name="FTPStore">
                                        <Property name="Scheme" value="ftp"/>
                                        <Property name="Host" value="www.remoteftp.org"/>
```

```
                    <Property name="Port" value="21"/>
                    <Property name="Username" value="anonymous"/>
                    <Property name="Password" value="nil"/>
                </Group>
                    <Group name="S3Store">
                    <Propertyname="Scheme" value="S3"/>
                    <Propertyname="Host"value="www.remoteftp.org"/>
                    <Propertyname="Port"value="21"/>
                    <Propertyname="Username"value="anonymous"/>
                    <Propertyname="Password"value="nil"/>
                </Group>
            </Groups>
        </Group>
    </Groups>
</Aneka>
```

Complete implementation of the task submission program, implementing dynamic submission and the appropriate synchronization logic[5]

```
using System;
using System.Threading;
using Aneka.Entity;
using Aneka.Tasks;
namespace GaussSample
{
/// <summary>
/// Class GaussApp. Defines the coordination logic of the
/// distributed application for computing the gaussian distribution.
/// </summary>
public class GaussApp
{
    /// <summary>
    /// Semaphore used to make the main thread wait while
    /// all the tasks are terminated.
    /// </summary>
    private ManualResetEventsemaphore;
    /// <summary>
    /// Counter of the running tasks.
    /// </summary>
    private int taskCount = 0;
    /// <summary>
    /// Aneka application instance.
    /// </summary>
    private AnekaApplication<AnekaTask, TaskManager>app;
    /// <summary>
    /// Main entry point for the application.
    /// </summary>
    /// <param name="args">An array of strings containing the command line.</param>
    public static void Main(string[] args)
    {
    try
```

```
{
// initialize the logging system
Logger.Start();
string confFile = "conf.xml";
if (args.Length> 0)
{
        confFile = args[0];
}
// get an instance of the Configuration class from file
Configuration conf = Configuration.GetConfiguration(confFile);
/* create an instance of the GaussApp and starts its execution
 with the given configuration instance */
GaussApp application = new GaussApp();
application.SubmitApplication(conf);
}
        catch(Exception ex)
{
IOUtil.DumpErrorReport(ex, "Fatal error while executing application.");
}
finally
{
        // terminate the logging thread
        Logger.Stop();
}
}
/// <summary>
/// Application submission method.
/// </summary>
/// <param name="conf">Application configuration.</param>
public void SubmitApplication(Configuration conf)
{
// initialize the semaphore and the number of task initially submitted
this.semaphore = new ManualResetEvent(false);
this.taskCount = 400;

// specify that the submission of task is dynamic
conf.SingleSubmission = false;
this.app = new AnekaApplication<Task,TaskManager>(conf);

/* attach methods to the event handler that notify the client code
 when tasks are completed or failed */
this.app.WorkUnitFailed +=
 new EventHandler<WorkUnitEventArgs<AnekaTask>>(this.OnWorkUnitFailed);
this.app.WorkUnitFinished +=
 new EventHandler<WorkUnitEventArgs<AnekaTask>>(this.OnWorkUnitFinished);
/* attach the method OnAppFinished to the Finished event so we can capture
 the application termination condition, this event will be fired in case of
both static application submission or dynamic application submission */

app.Finished += new EventHandler<ApplicationEventArgs>(this.OnAppFinished);
```

```
for(int i=0; i<400; i++)
{
        GaussTask gauss = new GaussTask();
        gauss.X = i;
        AnekaTask task = newAnekaTask(gauss);
        // add the task to the bag of work units to submit
        app.AddWorkunit(task);

}

// submit the entire bag
app.SubmitExecution();

// wait until signaled, once the thread is signaled the application is completed
this.semaphore.Wait();
}
/// <summary>
/// Event handler for task failure.
/// </summary>
/// <param name="sender">Event source: the application instance.</param>
/// <param name="args">Event arguments.</param>
private void OnWorkUnitFailed(object sender, WorkUnitEventArgs<AnekaTask>args)
{
        /* do nothing, we are not interested in task failure at the moment
         just dump to console the failure. */
        if (args.WorkUnit != null)
        {
                Exception error = args.WorkUnit.Exception;
                Console.WriteLine("Task {0} failed – Exception: {1}",
                args.WorkUnit.Name,
                (error == null ? "[Not given]" :error.Message);
        }
        /* we do not have to synchronize this operation because
        events handlers are run all in the same thread, and there
        will not be other threads updating this variable */
        this.taskCount--;
        if (this.taskCount == 0)
        {
        this.app.StopExecution();
        }
}
/// <summary>
/// Event handler for task completion.
/// </summary>
/// <param name="sender">Event source: the application instance.</param>
/// <param name="args">Event arguments.</param>
private void OnWorkUnitFinished(object sender, WorkUnitEventArgs<AnekaTask>args)
{
/* we do not have to synchronize this operation because
events handlers are run all in the same thread, and there
will not be other threads updating this variable */
```

```
this.taskCount--;
 /* if the task is completed for sure we have a WorkUnit instance
  and we do not need to check as we did before. */
GaussTask gauss = (GaussTask) args.WorkUnit.Task;

/* we check whether it is an initially submitted task or a task
that we submitted as a reaction to the completion of another task */
if (task.X - Math.Abs(task.X) == 0)
{
/* It was an original task, then we increase the value of X by 0.5
and submit another task */
        GaussTaskfraq = GaussTask();
        fraq.X = gauss.X + 0.5;
        AnekaTask task =new AnekaTask(fraq);
        this.taskCount++;
         // we call the ExecuteWorkUnit method that is used for dynamic submission
        app.ExecuteWorkUnit(task);
}
Console.WriteLine("Task {0} completed - [X:{1},Y:{2}]",
args.WorkUnit.Name, gauss.X, gauss.Y);
 if (this.taskCount == 0)
 {
this.app.StopExecution();
}
}
/// <summary>
/// Event handler for the application termination.
/// </summary>
/// <param name="sender">Event source: the application instance.</param>
/// <param name="args">Event arguments.</param>
private void OnWorkUnitFinished(object sender, ApplicationEventArgsargs)
{
/ *unblock the main thread because we have identified the termination
of the application */
this.semaphore.Set();
}
}
}
```

DATA-INTENSIVE COMPUTING

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

Data-intensive computations are common in a wide range of application disciplines. One of the most well-known is computational science. People who undertake scientific simulations and experiments are frequently eager to generate, evaluate, and analyze massive amounts of data. Telescopes scanning the sky

generate hundreds of terabytes of data per second; the collection of photographs of the sky easily exceeds petabytes over a year. Bioinformatics applications mine databases that can include terabytes of information. Earthquake simulators handle vast amounts of data generated by tracking the tremors of the Earth over the whole world. Aside from scientific computing, other IT business areas demand data-intensive computation assistance. Any telecom company's customer data will likely exceed 10-100 terabytes. This volume of data is mined not only to create bills, but also to uncover situations, trends, and patterns that assist these organizations to deliver better service.

BIG DATA

Big data is a term that refers to the vast volume of data – both structured and unstructured – that a business faces daily. However, it is not the quantity of data that is crucial. What organizations do with the data is important. Big data can be mined for insights that will lead to improved judgments and smart business movements.

Volume: Data is collected by organizations from several sources, including commercial transactions, smart (IoT) devices, industrial equipment, videos, social media, and more. Previously, storing it would have been a challenge, but cheaper storage on platforms such as data lakes and Hadoop has alleviated the strain.

Velocity: As the Internet of Things expands, data enters organizations at an unprecedented rate and must be processed promptly. RFID tags, sensors, and smart meters are boosting the demand for near-real-time data processing.

Variety: Data comes in a variety of forms, ranging from structured, numeric data in traditional databases to unstructured text documents, emails, films, audios, stock ticker data, and financial transactions.

Veracity: This refers to the degree of correctness and trustworthiness of data sets. Raw data acquired from a variety of sources might result in data quality concerns that are difficult to identify.

Variability: Big data may have various meanings or be formatted differently in different data sources.

Value: The business value of the collected data.

The significance of big data is determined not by how much data you have, but by what you do with it. When large data of different is combined with powerful analytics many benefits can be gained.

- Big data assists oil and gas businesses in identifying possible drilling areas and monitoring pipeline operations; similarly, utilities use it to track power networks.
- Big data systems are used by financial services businesses for risk management and real-time market data analysis.
- Big data is used by manufacturers and transportation businesses to manage supply chains and improve delivery routes.
- Emergency response, crime prevention, and smart city efforts are some of the other government applications.

TECHNOLOGIES FOR DATA-INTENSIVE COMPUTING

The creation of applications that are primarily focused on processing massive amounts of data is referred to as data-intensive computing. Storage systems and programming models form a natural categorization of the technologies that enable data-intensive computing.

Storage Systems

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

analytics. So, such increment of volume of data requires more efficient data management techniques. As we discussed earlier, Cloud computing provides on-demand access to enormous amounts of computing capability enabling developers to create software systems that expand progressively to arbitrary levels of parallelism as software applications and services run on hundreds or thousands of nodes. Traditional computing architectures are not capable to accommodate such a dynamic environment. So, new data management technologies are introduced to support data-intensive computing.

Distributed file systems play a vital role in big data management providing the interface to store information in the form of files and perform read and write operations later on. Some file system implementations in a cloud environment address the administration of large quantities of data on several nodes. Those file systems comprise data storage support for large computing clusters, supercomputers, parallel architectures, as well as storage and computing clouds.

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

The **Lustre** file system is an open-source, parallel file system that meets many of the needs of high-performance computing simulation settings. The Lustre file system grew from a research project at Carnegie Mellon University to a file system that supports some of the world's most powerful supercomputers. Lustre is meant to enable access to petabytes (PBs) of storage while serving thousands of clients at hundreds of gigabytes per second (GB/s). Lustre provides a POSIX-compliant file system interface and can grow to thousands of clients, petabytes of storage, and hundreds of gigabytes per second of I/O bandwidth. The Metadata Servers (MDS), Metadata Targets (MDT), Object Storage Servers (OSS), Object Server Targets (OST), and Lustre clients are the core components of the Lustre file system.

The **IBM General Parallel File System (GPFS)** is the high-performance distributed file system developed by IBM that provides support for the RS/6000 supercomputer and Linux computing clusters. It is a cluster file system that allows numerous nodes to access a single file system or collection of file systems at the same time. These nodes can be fully SAN-attached or a combination of SAN and network-attached. This offers high-performance access to this shared collection of data, which may be used to support a scale-out solution or to provide a high-availability platform. Beyond common data access, GPFS provides additional capabilities such as data replication, policy-based storage management, and multi-site operations. A GPFS cluster can be made up of AIX nodes, Linux nodes, Windows server nodes, or a combination of the three. GPFS may run on virtualized instances in contexts that use logical partitioning or other hypervisors to provide shared data access. Multiple GPFS clusters can share data inside a single place or over WAN networks.

Google File System (GFS) is the storage architecture that allows distributed applications to run on Google's computing cloud. GFS is designed to run on multiple, standard x86-based servers and it is intended to be a fault-tolerant, highly available distributed file system. The GFS was created with many of the same aims in mind as previous distributed file systems, such as scalability, performance, dependability, and resilience. GFS, on the other hand, was created by Google to achieve some specific aims motivated by certain significant observations of their workload. Google experienced regular failures of its cluster machines so, a distributed file system must be exceedingly faulted tolerant and have some type of automated fault recovery. As multi-gigabyte files are prevalent in today's computing environment, I/O and file block size must be adequately planned. Similarly, the majority of files are appended rather than rewritten or altered so optimization efforts should be directed at appending files. GFS, rather than being a generic implementation of a distributed file system, is tailored to Google's needs in terms of distributed storage for applications.

Sector/Sphere facilitates distributed data storage, dissemination, and processing over huge clusters of commodity computers, either inside a single data center or across numerous data centers. The sector is a distributed file system that is fast, scalable, and secure. The Sphere is a high-performance parallel data processing engine that can handle Sector data files on storage nodes using extremely simple programming interfaces. The Sector is one of the few file systems capable of supporting several data centers across wide area

networks. Sector, unlike other file systems, does not split files into blocks but instead replicates full files over several nodes, allowing users to tailor the replication technique for improved performance. The architecture of the system is made up of four nodes: a security server, one or more master nodes, slave nodes, and client machines. The security server stores all information regarding access control rules for users and files, whereas master servers coordinate and serve client I/O requests, which eventually interact with slave nodes to access files. The Sector employs UDT protocol to provide high-speed data transfer, and its data placement technique enables it to function efficiently as a content distribution network over WAN. For high reliability and availability, Sector does not require hardware RAID; instead, data is automatically replicated in Sector.

Amazon Simple Storage Service (Amazon S3) is storage for the Internet which is designed to make web-scale computing easier. Despite the internal specification is not disclosed, the system is said to enable high availability, dependability, scalability, security, high performance, limitless storage, and minimal latency at a cheap cost. The solution provides flat storage space grouped into buckets and linked to an Amazon Web Services (AWS) account. Each bucket can hold several objects, each of which is identifiable by a unique key. Objects are identifiable by unique URLs and accessible over HTTP, allowing for extremely easy get-put semantics. Because HTTP is used, no special library is required to access the storage system, the items of which may also be obtained via the Bit Torrent protocol.

Customers of all sizes and sectors may use it to store and safeguard any quantity of data for a variety of use cases, including data lakes, websites, mobile apps, backup and restore, archiving, business applications, IoT devices, and big data analytics. Amazon S3 offers simple administration capabilities that allow you to organize your data and establish fine-grained access restrictions to fit your unique business, organizational, and compliance needs.Amazon S3 is designed for 99.999999999% (11 9's) of durability, and stores data for millions of applications for companies all around the world.

After the file system, NoSQL systems also play a vital role in the field of intensive data processing. NoSQL is a database type that stores and retrieves data without previously defining its structure - an alternative to more strict relational databases. These days, the term NoSQL refers to any storage and database management technologies that differ from the relational paradigm in some way.Their overarching concept is to break free from the constraints imposed by the relational paradigm and deliver more efficient solutions. This frequently involves using tables with no defined schemas to enable a broader range of data types or avoiding joins to improve performance and grow horizontally.

The emergence of the NoSQL movement has been driven by two major factors: the first one, in many situations, basic data models are sufficient to describe the information utilized by applications, and another, the amount of information held in unstructured forms has increased significantly in the recent years.

A Classification of NoSQL Presented by *Stephen Yen*

Type	Notable examples of this type
Key-value cache	Apache Ignite, Couchbase, Coherence, eXtreme Scale, Hazelcast, Infinispan, Memcached, Redis, Velocity
Key-value store	Azure Cosmos DB, ArangoDB, Aerospike, Couchbase, Redis
Key-value store *(eventually consistent)*	Azure Cosmos DB, Oracle NoSQL Database, Dynamo, Riak, Voldemort
Key-value store *(ordered)*	FoundationDB, InfinityDB, LMDB, MemcacheDB
Tuple store	Apache River, GigaSpaces
Object database	Objectivity/DB, Perst, ZopeDB
Document store	Azure Cosmos DB, ArangoDB, BaseX, Clusterpoint, Couchbase, CouchDB, DocumentDB, eXist-db, IBM Domino, MarkLogic, MongoDB, Qizx, RethinkDB, Elasticsearch
Wide Column Store	Azure Cosmos DB, Amazon DynamoDB, Bigtable, Cassandra, Google Cloud Datastore, HBase, Hypertable, Scylla
Native multi-model database	ArangoDB, Azure Cosmos DB, OrientDB, MarkLogic

Apache CouchDB and MongoDB, Amazon Dynamo, Google Bigtable, Apache Cassandra, Hadoop HBase etc. are some notable implementations that enable data-intensive applications.

Apache CouchDB is a NoSQL document database that is open source that gathers and stores data in JSON-based document formats. In contrast to relational databases, CouchDB has a schema-free data architecture, which facilitates record maintenance across a variety of computer platforms, mobile phones and web browsers.Apache CouchDB stores data in JSON queries in JavaScript using MapReduce and provides an API over HTTP. In CouchDB, the fundamental unit of data is the document, which also includes metadata. Document fields are individually named and include a variety of values; there is no predefined limit on text size or element count. CouchDB ensures ACID properties on data.

MongoDB is a major NoSQL database and an open-source document database developed in C++. It is a document-oriented NoSQL database that is used for large-scale data storage. It uses JSON-like documents with optional schemas and employs collections and documents rather than tables and rows as in traditional relational databases. Documents are made up of key-value pairs, which are the fundamental unit of data and collections are the equivalent of relational database tables in that they include collections of documents and functions. MongoDB offers sharding, which is the ability to divide a collection's content over many nodes.

CouchDB and MongoDB both offer schema-less storage in which the primary objects are documents grouped into a set of key-value fields. Each field's value can be a text, integer, float, date, or an array of data. These databases provide a RESTful interface and data are stored in JSON format. Both of the DBs handle big files like documents and allow searching and indexing of the stored data using the MapReduce programming model. They also offer JavaScript as a base language for data searching and manipulation rather than SQL. These two systems also provide data replication and high availability.

Amazon DynamoDB is a key-value and document database that promises performance in single-digit milliseconds at any size. It's a fully managed, multi-region, multi-active, long-lasting database with built-in security, backup and restores, and in-memory caching for web-scale applications. DynamoDB can handle more than 10 trillion requests per day, with peaks of more than 20 million requests per second.Many of the world's fastest-growing companies, like Lyft, Airbnb, and Redfin, as well as corporations like Samsung, Toyota, and Capital One, rely on DynamoDB's scale and performance to serve mission-critical workloads.

Hundreds of thousands of Amazon Online Services customers have selected DynamoDB as their key-value and document database for mobile, web, gaming, ad tech, IoT, and other applications requiring low-latency data access at any scale.

Open challenges in data-intensive computing given by Ian Gorton et al.

- Scalable algorithms that can search and process massive datasets
- New metadata management technologies that can scale to handle complex, heterogeneous, and distributed data sources
- Advances in high-performance computing platforms aimed at providing better support for accessing in-memory multiterabyte data structures
- High-performance, highly reliable, petascale distributed file systems
- Data signature-generation techniques for data reduction and rapid processing
- New approaches to software mobility for delivering algorithms that can move the computation to where the data are located
- Specialized hybrid interconnection architectures that provide better support for filtering multi-gigabyte datastreams coming from high-speed networks and scientific instruments
- Flexible and high-performance software integration techniques that facilitate the combination of software modules running on different platforms to quickly form analytical pipelines

Google Bigtable is a distributed, column-oriented data store developed by Google to manage massive volumes of structured data related to the company's Internet search and Web services operations. It is a fully managed, scalable NoSQL database service for large analytical and operational workloads with up to 99.999% availability. Bigtable was created to help with applications that require large scalability. The system was designed to handle petabytes of data. The database was designed to run on clustered servers and has a basic data format described by Google as "*a sparse, distributed, permanent multi-dimensional sorted map.*" The data is placed in order by row key, and the map's indexing is sorted by row, column keys, and timestamps. Algorithms for compression aid in achieving high capacity.Google Bigtable is the database used by Google App Engine Datastore, Google Personalized Search, Google Earth, and Google Analytics. Google has kept the program as a proprietary, in-house technology. Nonetheless, Bigtable has had a significant effect on the architecture of NoSQL databases. Google software developers disclosed Bigtable information during a symposium in 2006.

As the published material revealed BigTable's inner workings, several other corporations and open-source development teams established Bigtable equivalents, such as the Apache HBase, Cassandra, Hypertable, etc.

Apache Cassandra is a distributed, open-source NoSQL database. It demonstrates a partitioned wide column storage approach with eventually consistent semantics. Apache Cassandra was created at Facebook utilizing a staged event-driven architecture to integrate Amazon's Dynamo distributed storage and replication techniques with Google's Bigtable data and storage engine paradigm. Both Dynamo and Bigtable were created to fulfill rising requirements for scalable, dependable, and highly available storage systems, yet they both have flaws.Cassandra was intended as a best-in-class mixture of both systems to satisfy increasing large-scale storage requirements, both in terms of data footprint and query traffic. Since applications began to need complete global replication and always-available low-latency read and write, it became clear that a new type of database architecture was required, as relational database solutions struggled to fulfill the new needs of global-scale applications.

HBase is a non-relational column-oriented database management system that works on top of the Hadoop Distributed File System (HDFS). HBase was inspired by Google Bigtable in its design. HBase provides a fault-tolerant method of storing sparse data sets, which are common in many big data applications. It's ideal for real-time data processing or random read/write access to enormous amounts of data. HBase, unlike relational database systems, does not support a structured query language such as SQL; as HBase is not a relational data store. HBase apps, like Apache MapReduce apps, are written in Java. HBase also supports the development of applications in Apache Avro, REST, and Thrift, etc.

HBase systems are intended to scale linearly. It is made up of regular tables with rows and columns, similar to a regular relational database. Each table must have a primary key declared, and all access attempts to HBase tables must utilize that key. As a component of HBase, Avro supports a diverse variety of fundamental data types such as numeric, binary data, and strings, as well as a variety of complex kinds such as arrays, maps, enumerations, and records. The data can also be sorted in a certain order.

After learning about file systems and databases required for data-intensive computing let's now discuss the programming of the data-intensive application. Platforms for programming data-intensive applications include abstractions that aid in the expression of computations over the big data, as well as runtime systems capable of effectively managing such massive amounts of data. The programming platforms emphasize data processing and shift transfer management into the runtime system, making data always available whenever needed. MapReduce programming platform follows the same approach and describes computing in two simple functions—map and reduce—and hides the difficulties of maintaining massive data files in the platform's distributed file system.

MAP-REDUCE PROGRAMMING

Data-intensive computing is concerned with enormous amounts of data. Several application domains, from computational research to social networking, generate vast amounts of data that must be effectively stored, accessed, indexed, and evaluated. Computing gets more difficult as the amount of information grows at a faster rate over time. Distributed computing may surely aid in solving the difficulties by providing more scalable and economical storage designs as well as improved data computation and processing capabilities. But using parallel and distributed approaches to enable data-intensive computing is not easy; there are various obstacles to overcome in the form of data representation, efficient algorithms, and scalable infrastructures. MapReduce is a common programming approach for developing data-intensive applications and deploying them on clouds.

MapReduce, a software framework allows to quickly write programs that process massive volumes of data (multi-terabyte datasets) in parallel across enormous clusters (thousands of nodes) in a fault-tolerant way. A MapReduce job typically divides the incoming data set into distinct pieces that are processed in parallel by the map jobs. The framework sorts the map outputs, which are subsequently fed into the reduction jobs. Typically, both the job's input and output are saved in a file system. The framework manages task scheduling, task monitoring, and task re-execution.

MapReduce is a programming engine is developed by Google for processing huge amounts of data. It expresses an application's computational logic in two basic functions: **map** and **reduce**. The distributed storage infrastructure, such as the Google File System, is in charge of giving access to data, duplicating files, and finally relocating them where needed. So, application developers have access to an interface that displays data at a higher level: as a collection of key-value pairs. MapReduce applications' processing is thus arranged into a workflow of the map and reduce operations that are completely controlled by the runtime system; developers need only specify how the map and reduce functions act on the key-value pairs.

The MapReduce model is stated as two functions, which are defined as follows:

map (k1, v1) → list (k2, v2)

reduce (k2, list (v2)) → list (v2)

The map function takes in a key-value pair and returns a list of key-value pairs of various kinds. The reduce function takes a key and a list of values and returns a list of items of the same type. The types (k1, v1, k2, kv2) used in the expressions of the two functions give indications as to how these two functions are linked and run to carry out a MapReduce job's computation: The output of map tasks is aggregated together by grouping the values according to their associated keys and becomes the input of reduce tasks, which reduces the list of associated values to a single value for each of the keys discovered. As a result, the input to a MapReduce computation is a collection of key-value pairs, <k1, v1>, and the final output is a list of values: list (v2)

Consider an eCommerce system, for instance, daraz.com, that receives a million payment requests each day. Several exceptions may be thrown during these queries, including "Payment refused by a payment gateway," "Out of inventory," and "Invalid address." etc. A developer can find out which exceptions were thrown and their frequency from the last week's logs implementing the MapReduce paradigm.

Input and Output types of a MapReduce job:

(input)<k1, v1>→ map →<k2, v2>→ combine →<k2, v2>→ reduce →<k3, v3>**(output)**

Let us discuss this mechanism with the help of a simple application that counts the words.

WordCount.java based on Hadoop

```
package org.myorg;
import java.io.IOException;
import java.util.*;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.*;
public class WordCount {

  public static class Map extends MapReduceBase implements Mapper<LongWritable, Text, Text,
IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output,
Reporter reporter) throws IOException {
      String line = value.toString();
    StringTokenizer tokenizer = new StringTokenizer(line);
      while (tokenizer.hasMoreTokens()) {
    word.set(tokenizer.nextToken());
    output.collect(word, one);
      }
     }
    }
  public static class Reduce extends MapReduceBase implements Reducer<Text, IntWritable, Text,
IntWritable> {
    public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text, IntWritable>
output, Reporter reporter) throws IOException {
      int sum = 0;
      while (values.hasNext()) {
       sum += values.next().get();
      }
    output.collect(key, new IntWritable(sum));
     }
    }

    public static void main(String[] args) throws Exception {
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("wordcount");

    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);

    conf.setMapperClass(Map.class);
    conf.setCombinerClass(Reduce.class);
    conf.setReducerClass(Reduce.class);

    conf.setInputFormat(TextInputFormat.class);
    conf.setOutputFormat(TextOutputFormat.class);

    FileInputFormat.setInputPaths(conf, new Path(args[0]));
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));
    JobClient.runJob(conf);
     }
    }
```

Assuming **HADOOP_HOME** is the root of the installation and **HADOOP_VERSION** is the Hadoop version installed, compile WordCount.java and create a jar:

```
$ mkdirwordcount_classes
$ javac -classpath ${HADOOP_HOME}/hadoop-${HADOOP_VERSION}-core.jar -d
wordcount_classes WordCount.java
$ jar -cvf /usr/kcc/wordcount.jar -C wordcount_classes/ .
```

Assume

- `/usr/kcc/wordcount/input` - input directory in HDFS
- `/usr/kcc/wordcount/output` - output directory in HDFS

Sample text-files as input:

```
$ bin/hadoopdfs -ls /usr/kcc/wordcount/input/
/usr/kcc/wordcount/input/file01
/usr/kcc/wordcount/input/file02

$ bin/hadoopdfs -cat /usr/kcc/wordcount/input/file01
Hello World Bye World
$ bin/hadoopdfs -cat /usr/kcc/wordcount/input/file02
Hello Hadoop Goodbye Hadoop
```

Run the application:

```
$ bin/hadoop jar /usr/kcc/wordcount.jar org.myorg.WordCount
/usr/kcc/wordcount/input /usr/kcc/wordcount/output
```

Output:

```
$ bin/hadoopdfs -cat /usr/kcc/wordcount/output/part-00000
Bye 1
Goodbye 1
Hadoop 2
Hello 2
World 2
```

In the program mentioned above, input is given through 2 files:

File01 contains: Hello World Bye World

File02 contains: Hello Hadoop Goodbye Hadoop

For the given sample input the first map produces:

< Hello, 1>

< World, 1>

< Bye, 1>

< World, 1>

The second map produces:

< Hello, 1>

< Hadoop, 1>

< Goodbye, 1>

< Hadoop, 1>

The output of each map is passed through the combiner (conf.setCombinerClass(Reduce.class);) for local aggregation, after being sorted on the keys:

The output of the first map:

< Bye, 1>
< Hello, 1>
< World, 2>

The output of the second map:

< Goodbye, 1>
< Hadoop, 2>
< Hello, 1>

Then the Reducer sums up the values, which are the occurrence counts for each key. Thus, the output of the *word count* job is:

< Bye, 1>
< Goodbye, 1>
< Hadoop, 2>
< Hello, 2>
< World, 2>

[*Source*: https://hadoop.apache.org]

MapReduce performs two critical functions: it filters and distributes work to various nodes within the cluster or map, a function known as the mapper, and it organizes and reduces the data from each node into a coherent answer to a query, known as the reducer.

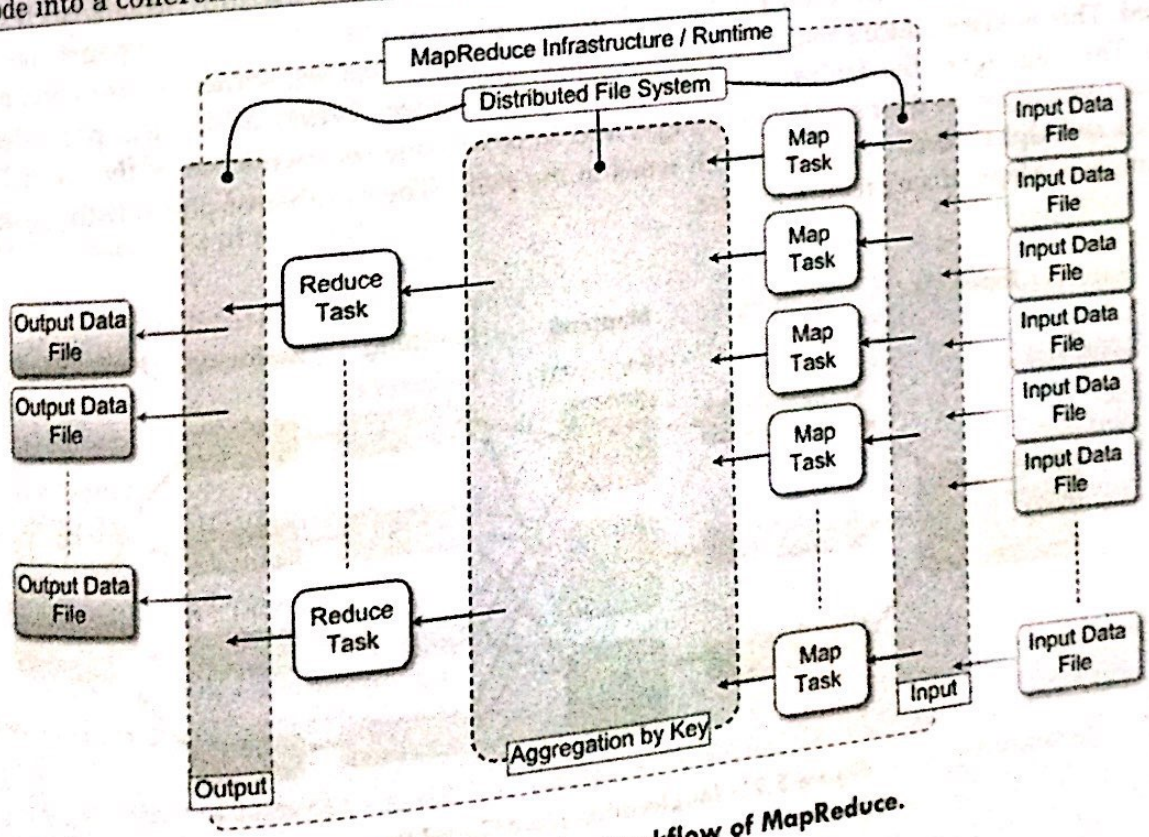


Figure 5.19: Computation workflow of MapReduce.

MapReduce runs in parallel over huge cluster sizes to spread input data and collate outcomes. Because cluster size does not affect the final output of a processing operation, workloads can be distributed among practically any number of computers. As a result, MapReduce makes software development easier. MapReduce is accessible in a variety of programming languages, including C, C++, Java, Ruby, Perl, and Python. MapReduce libraries allow programmers to build jobs without having to worry about communication or coordination between nodes.

MapReduce is also fault-tolerant, with each node sending its status to a master node regularly. If a node fails to reply as expected, the master node reassigns that portion of the work to other nodes in the cluster that are available. This enables resilience and makes it viable for MapReduce to run on affordable commodity servers.

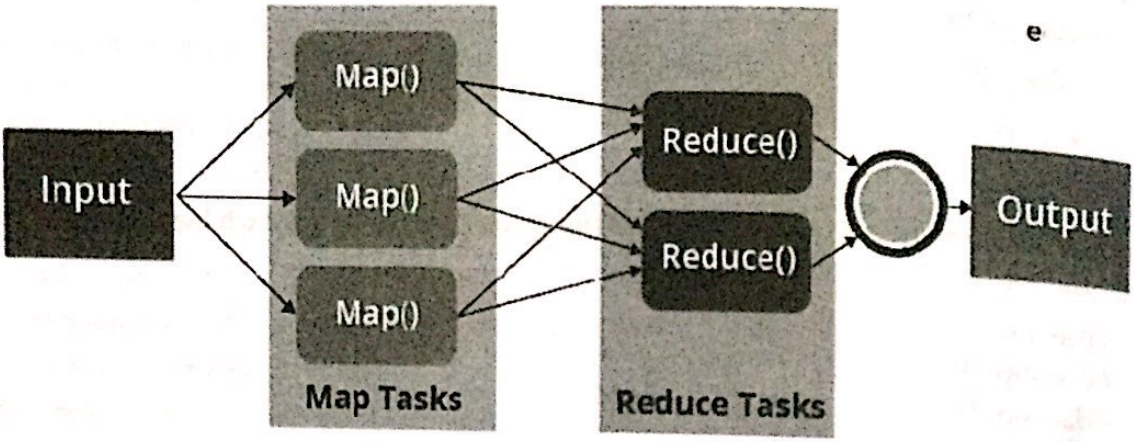


Figure 5.20: Map Reduce

MapReduce's strength is in its ability to handle large data sets by distributing processing over many nodes and then combining or reducing the outputs of those nodes. Users might, for example, use a single server application to list and tally the number of times each word appears in a novel, but this is time-consuming.Users, on the other hand, can divide the workload among 26 individuals, such that each person takes a page, writes a word on a separate piece of paper, and then takes a new page when they are completed. This is MapReduce's map component. And if someone leaves, someone else takes his or her position. This highlights the fault-tolerant nature of MapReduce. When all of the pages have been processed, users organize their single-word pages into 26 boxes, one for each letter of the word. Each user takes a box and alphabetically organizes each word in the stack. The number of pages with the same term is an example of MapReduce's reduce feature.

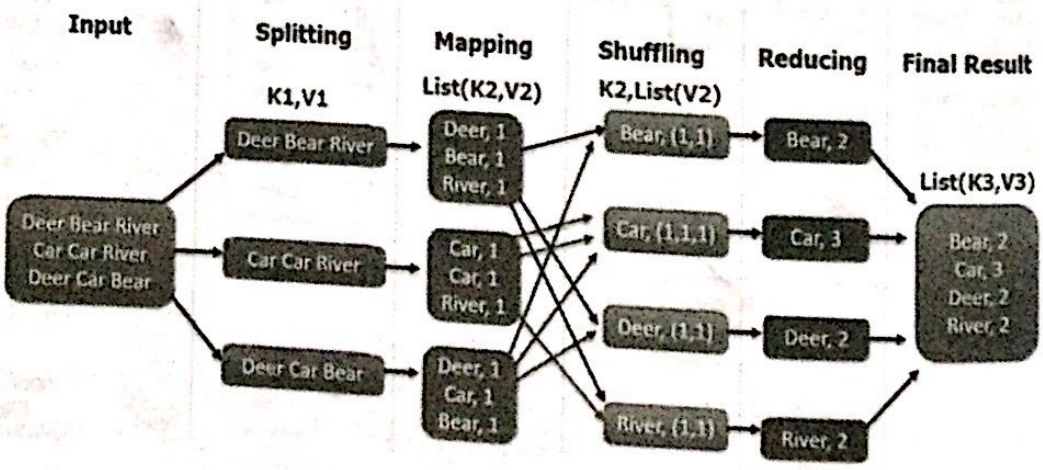


Figure 5.21: MapReduce Word Count Process

MapReduce might be used by a social networking site to assess users' possible friends, coworkers, and other contacts based on on-site activity, names, localities, employers, and a variety of other data variables. A booking website may utilize MapReduce to assess customers' search criteria and history activity, and then produce personalized options for each.

WordCount implementation using Hadoop and Java

```
package org.myorg;
import java.io.*;
import java.util.*;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.filecache.DistributedCache;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.*;
public class WordCount extends Configured implements Tool {
public static class Map extends MapReduceBase implements Mapper<LongWritable, Text, Text,
IntWritable> {
    static enum Counters { INPUT_WORDS }
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();
    private booleancaseSensitive = true;
    private Set<String>patternsToSkip = new HashSet<String>();
    private long numRecords = 0;
    private String inputFile;
    public void configure(JobConf job) {
  caseSensitive = job.getBoolean("wordcount.case.sensitive", true);
  inputFile = job.get("map.input.file");
     if (job.getBoolean("wordcount.skip.patterns", false)) {
  Path[] patternsFiles = new Path[0];
      try {
  patternsFiles = DistributedCache.getLocalCacheFiles(job);
      } catch (IOExceptionioe) {
  System.err.println("Caught exception while getting cached files: " +
StringUtils.stringifyException(ioe));
      }
      for (Path patternsFile :patternsFiles) {
  parseSkipFile(patternsFile);
      }
    }
  }
private void parseSkipFile(Path patternsFile) {
    try {
  BufferedReaderfis = new BufferedReader(new FileReader(patternsFile.toString()));
      String pattern = null;
      while ((pattern = fis.readLine()) != null) {
  patternsToSkip.add(pattern);
      }
    } catch (IOExceptionioe) {
  System.err.println("Caught exception while parsing the cached file '" + patternsFile + "' : " +
StringUtils.stringifyException(ioe));
```

```
    }
    }
    public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output,
Reporter reporter) throws IOException {
      String line = (caseSensitive) ?value.toString() : value.toString().toLowerCase();
for (String pattern :patternsToSkip) {
  line = line.replaceAll(pattern, "");
 }
StringTokenizer tokenizer = new StringTokenizer(line);
      while (tokenizer.hasMoreTokens()) {
    word.set(tokenizer.nextToken());
    output.collect(word, one);
    reporter.incrCounter(Counters.INPUT_WORDS, 1);
      }

      if ((++numRecords % 100) == 0) {
    reporter.setStatus("Finished processing " + numRecords + " records " + "from the input file: " +
inputFile);
      }
     }
    }
public static class Reduce extends MapReduceBase implements Reducer<Text, IntWritable, Text,
IntWritable> {
      public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text, IntWritable>
output, Reporter reporter) throws IOException {
       int sum = 0;
       while (values.hasNext()) {
        sum += values.next().get();
       }
    output.collect(key, new IntWritable(sum));
     }
    }

     public int run(String[] args) throws Exception {
    JobConf conf = new JobConf(getConf(), WordCount.class);
    conf.setJobName("wordcount");

    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);
    conf.setMapperClass(Map.class);
    conf.setCombinerClass(Reduce.class);
    conf.setReducerClass(Reduce.class);
    conf.setInputFormat(TextInputFormat.class);
    conf.setOutputFormat(TextOutputFormat.class);
      List<String>other_args = new ArrayList<String>();
      for (int i=0; i<args.length; ++i) {
       if ("-skip".equals(args[i])) {
    DistributedCache.addCacheFile(new Path(args[++i]).toUri(), conf);
    conf.setBoolean("wordcount.skip.patterns", true);
      } else {
    other_args.add(args[i]);
```

```
    }
  }
  FileInputFormat.setInputPaths(conf, new Path(other_args.get(0)));
  FileOutputFormat.setOutputPath(conf, new Path(other_args.get(1)));
  JobClient.runJob(conf);
  return 0;
 }
 public static void main(String[] args) throws Exception {
  int res = ToolRunner.run(new Configuration(), new WordCount(), args);
 System.exit(res);
 }
}
```

Running the Program

Sample text-files as input:

```
$ bin/hadoopdfs -ls /usr/kcc/wordcount/input/
/usr/kcc/wordcount/input/file01
/usr/kcc/wordcount/input/file02

$ bin/hadoopdfs -cat /usr/kcc/wordcount/input/file01
Hello World, Bye World!

$ bin/hadoopdfs -cat /usr/kcc/wordcount/input/file02
Hello Hadoop, Goodbye to hadoop.
```

Run the application:

```
$ bin/hadoop jar /usr/kcc/wordcount.jar org.myorg.WordCount
/usr/kcc/wordcount/input /usr/kcc/wordcount/output
```

Output:

```
$ bin/hadoopdfs -cat /usr/kcc/wordcount/output/part-00000
Bye 1
Goodbye 1
Hadoop, 1
Hello 2
World! 1
World, 1
hadoop. 1
to 1
```

Now, let us plug in a pattern file that lists the word patterns to be ignored, via the `DistributedCache`.

```
$ hadoopdfs -cat /user/kcc/wordcount/patterns.txt
\.
\,
\!
to
```

Run it again, this time with more options:

```
$ bin/hadoop jar /usr/kcc/wordcount.jar org.myorg.WordCount -
Dwordcount.case.sensitive=true /usr/kcc/wordcount/input
/usr/kcc/wordcount/output -skip /user/kcc/wordcount/patterns.txt
```

As expected, the output:

```
$ bin/hadoopdfs -cat /usr/kcc/wordcount/output/part-00000
Bye 1
Goodbye 1
Hadoop 1
Hello 2
```

```
World 2
hadoop 1
```

Run it once more, this time switch-off case-sensitivity:

```
$ bin/hadoop jar /usr/kcc/wordcount.jar org.myorg.WordCount -
Dwordcount.case.sensitive=false /usr/kcc/wordcount/input
/usr/kcc/wordcount/output -skip /user/kcc/wordcount/patterns.txt
```

Sure enough, the output:

```
$ bin/hadoopdfs -cat /usr/kcc/wordcount/output/part-00000
bye 1
goodbye 1
hadoop 2
hello 2
world 2
```

[*Source*: https://hadoop.apache.org]

The computation paradigm described by MapReduce is relatively simple, allowing for improved efficiency for the developers who write methods for processing massive amounts of data. In the case of Google, where the majority of the information that has to be processed is kept in textual form and is represented by Web pages or log files, this strategy has shown to be successful. Distributed grep, count of URL-access frequency, reverse web-link graph, term vector per host, inverted index, and distributed sort are some of the examples that show the flexibility of MapReduce. These examples are mostly focused on text-based processing. MapReduce may also be utilized to address a broader range of issues with some modifications. An exciting use is in the field of machine learning, where statistical algorithms such as Support Vector Machines (SVM), Linear Regression (LR), Neural Network (NN), etc. are expressed as a map and reduce functions. Other intriguing applications may be found in the realm of compute-intensive applications, such as the high-precision calculation of Pi.

GOOGLE MAPREDUCE INFRASTRUCTURE

The user submits MapReduce tasks for execution by utilizing client libraries, which are responsible for sending input data files, registering the map and reduce functions, and returning control to the user once the task is done. MapReduce applications may be executed on a general distributed infrastructure with job-scheduling and distributed storage capabilities. On the distributed infrastructure, two types of processes are run: **master** processes and **worker** processes.

The master process is responsible for directing the execution of map and reduce tasks, as well as partitioning and rearranging the map task's intermediate output to feed the reduce tasks. The worker processes are used to host the execution of map and reduce operations, as well as to offer fundamental I/O facilities for interacting with input and output files. In a MapReduce calculation, input files are divided into splits of usually 16 to 64 MB and stored in a distributed file system (i.e., HDFS). By balancing the load, the master process produces the map tasks and allocates input splits to each of them.

Input and output buffers are utilized by worker processes to optimize the efficiency of the map and reduce operations. Output buffers for map operations are dumped to disk regularly to produce intermediate files. To equally separate the output of map operations, intermediate files are partitioned using a user-defined function. The positions of these pairings are then sent to the master process, which passes this information to the *reduced* tasks, which may gather the needed input through a remote procedure call to read from the map tasks' local storage. The key range is then sorted, and any keys that have the same value are grouped. Finally, the reduction job is run to generate the final result, which is saved in the global file system. This procedure is fully automated; users may control it by providing (in addition to the map and reduce functions) the number of map jobs, the number of partitions into which the final output is divided, and the partition function for the intermediate key range.

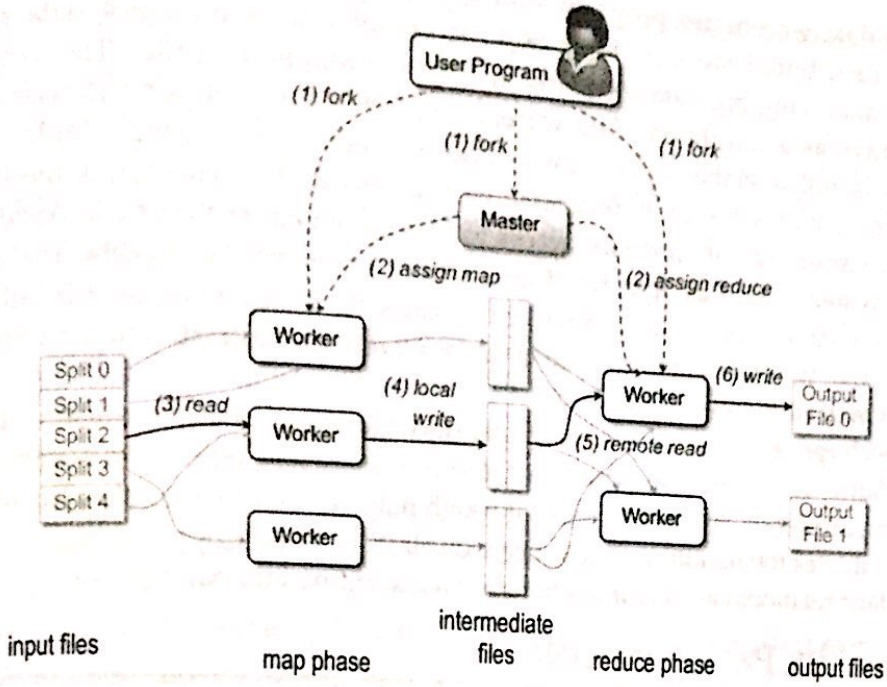


Figure 5.22: Google MapReduce infrastructure

The MapReduce runtime ensures application reliability by providing a fault-tolerant architecture. Faults of both master and worker processes, as well as machine failures that make intermediate outputs unavailable, are also addressed. Worker failures are addressed by rescheduling map work to another location. This is also the approach used to resolve machine failures when the legitimate intermediate output of map jobs is no longer available. Check pointing is used instead to address master process failure, allowing the MapReduce task to be restarted with little data and computation loss.

As discussed earlier, MapReduce is a simplified paradigm for the processing of big data. Although the paradigm is being used in a variety of scenarios, it puts limitations on how distributed algorithms should be structured to execute over a specified working mechanism of MapReduce architecture. The abstractions offered by MapReduce to process data are relatively simple but complex problems may take significant effort to be described in terms of map and reduce functions alone. As a result, several modifications and variants to the original MapReduce architecture have been proposed, to expand the MapReduce application area and give developers a more user-friendly interface for building distributed algorithms. Hadoop, Pig, Hive, Map-Reduce-Merge, etc. are some of those modified variants.

1. **Apache Hadoop** is a series of software projects that enable scalable and reliable distributed computing. Hadoop as a whole is an open-source implementation of the MapReduce architecture. It mainly consists of two projects: Hadoop Distributed File System (HDFS) and Hadoop MapReduce.HDFS is an implementation of the Google file system and Hadoop MapReduce offers the same functionality and abstraction as Google MapReduce. Originally developed and supported by Yahoo, Hadoop is currently the most mature and comprehensive data cloud application with a very active user community. has the support of developers and users. Yahoo operates the world's largest Hadoop cluster, consisting of 40,000 machines and more than 300, 000 cores, which can be used by academic institutions around the world.
2. **Apache Pig** is a platform for analyzing large data sets that consists of a high-level language for expressing data analysis programs and infrastructure for evaluating these programs. The outstanding property of Pig programs is that their structure is susceptible to significant parallelization, which in turn allows them to process very large data sets. Pig's infrastructure layer currently consists of a compiler that generates sequences of map-reduce programs for which large-scale parallel implementations are already in place. Pig's language layer currently consists of a text language called Pig Latin, which has key features such as simple programming, has optimization possibilities, and expandability.

3. **Apache Hive** data warehouse program simplifies the reading, writing, and management of huge data sets on distributed storage using SQL. It includes tools for quick data summary, ad hoc searches, and analyzing big datasets stored in Hadoop MapReduce files. The Hive architecture has identical features as a traditional data warehouse, but it does not perform well in terms of query latency, so it is not a viable option for online transaction processing. Built on top of Apache Hadoop, Hive provides the tools to enable easy access to data via SQL, a mechanism to impose structure on a variety of data formats, access to files stored either directly in Apache HDFS or other data storage systems such as Apache HBase, Query execution via Apache Tez, Apache Spark, or MapReduce, etc.Hive's main advantages are its capacity to scale out, as it is built on the Hadoop architecture, and its ability to provide a data warehouse infrastructure in situations where a Hadoop system is already running.
4. **Map-Reduce-Merge** is a change of the MapReduce paradigm that introduces a third phase to the traditional MapReduce pipeline, which is 'Merge' that lets in effectively combining data that has formerly been partitioned and taken care of through map and reduce modules. The Map-Reduce-Merge framework facilitates the handling of heterogeneous linked datasets by providing an abstraction that can express standard relational algebra operations as well as numerous join methods.

ANEKA MAPREDUCE PROGRAMMING

Aneka follows the reference model published by Google and implemented by Hadoop to give an implementation of the MapReduce abstractions. The MapReduce Programming Model specifies the abstractions and runtime support required to build MapReduce applications on top of Aneka. For MapReduce programming using Aneka, you have to have a basic understanding of object-oriented programming, cloud computing, distributed systems, and a good understanding of the .NET framework and C#. Similarly, Microsoft Visual Studio with C# and Aneka also need to be installed.

MapReduce.NET is a MapReduce solution for data centers that is similar to Google's MapReduce but focuses on the.NET and Windows platforms. MapReduce.NET offers a set of storage APIs to wrap input key/value pairs into starting files and extract result key/value pairs from results files, as well as an object-oriented interface for programming map and reduce functions.

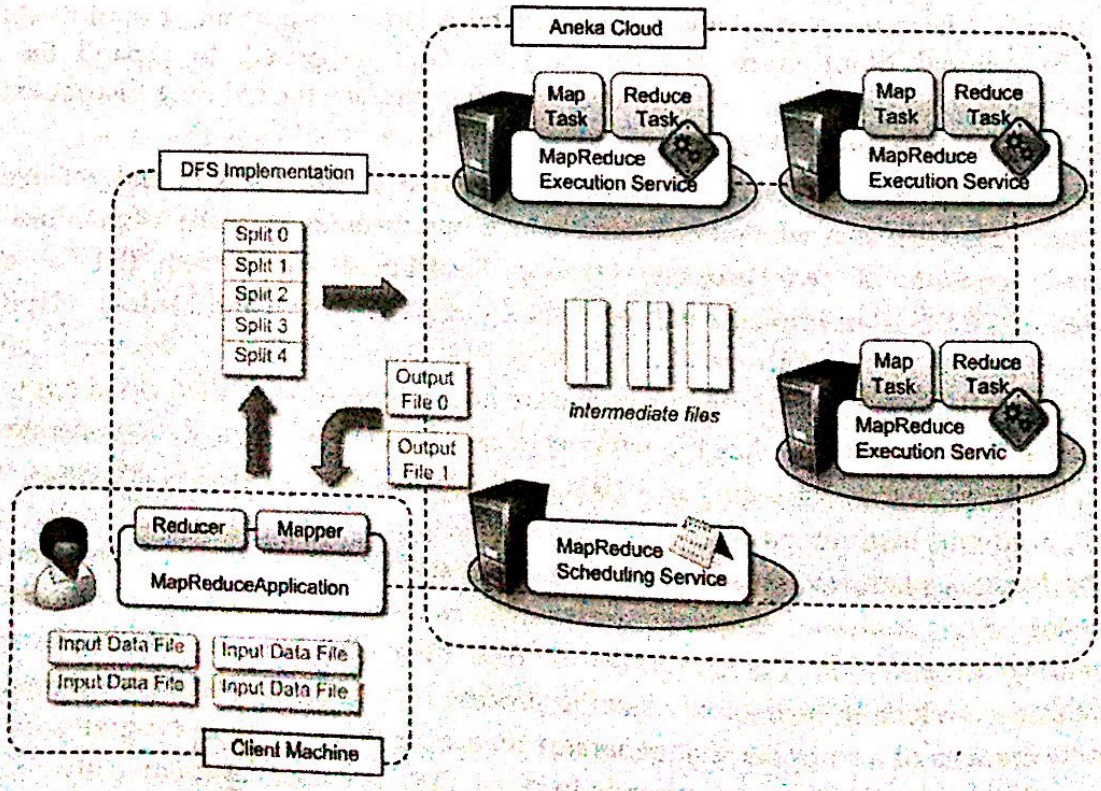


Figure 5.23: Aneka MapReduce infrastructure

MapReduce.NET is based on master-slave architecture. Its main components include manager, scheduler, executor, and storage.

1. **Manager:** The manager acts as a MapReduce computation agent. It sends applications to the MapReduce scheduler and gathers the final results after the execution is complete.
2. **Scheduler:** The scheduler assigns subtasks to available resources when users submit MapReduce.NET applications to it. During execution, it checks the progress of each job and performs task migration operations if certain nodes are much slower than others owing to heterogeneity.
3. **Executor:** Each executor awaits task execution orders from the scheduler. Normally, the input data for a Map job is located locally. Otherwise, the executor will have to collect input data from neighbors. Before executing a Reduce job, the executor must get all of the input and combine it. Furthermore, the executor checks the progress of the task execution and sends the status to the scheduler regularly.
4. **Storage:** the storage component of MapReduce.NET provides a distributed storage service over the .NET platform. It arranges disk spaces on all available resources as a virtual storage pool and provides an object-based interface with a flat namespace for managing data stored in it.

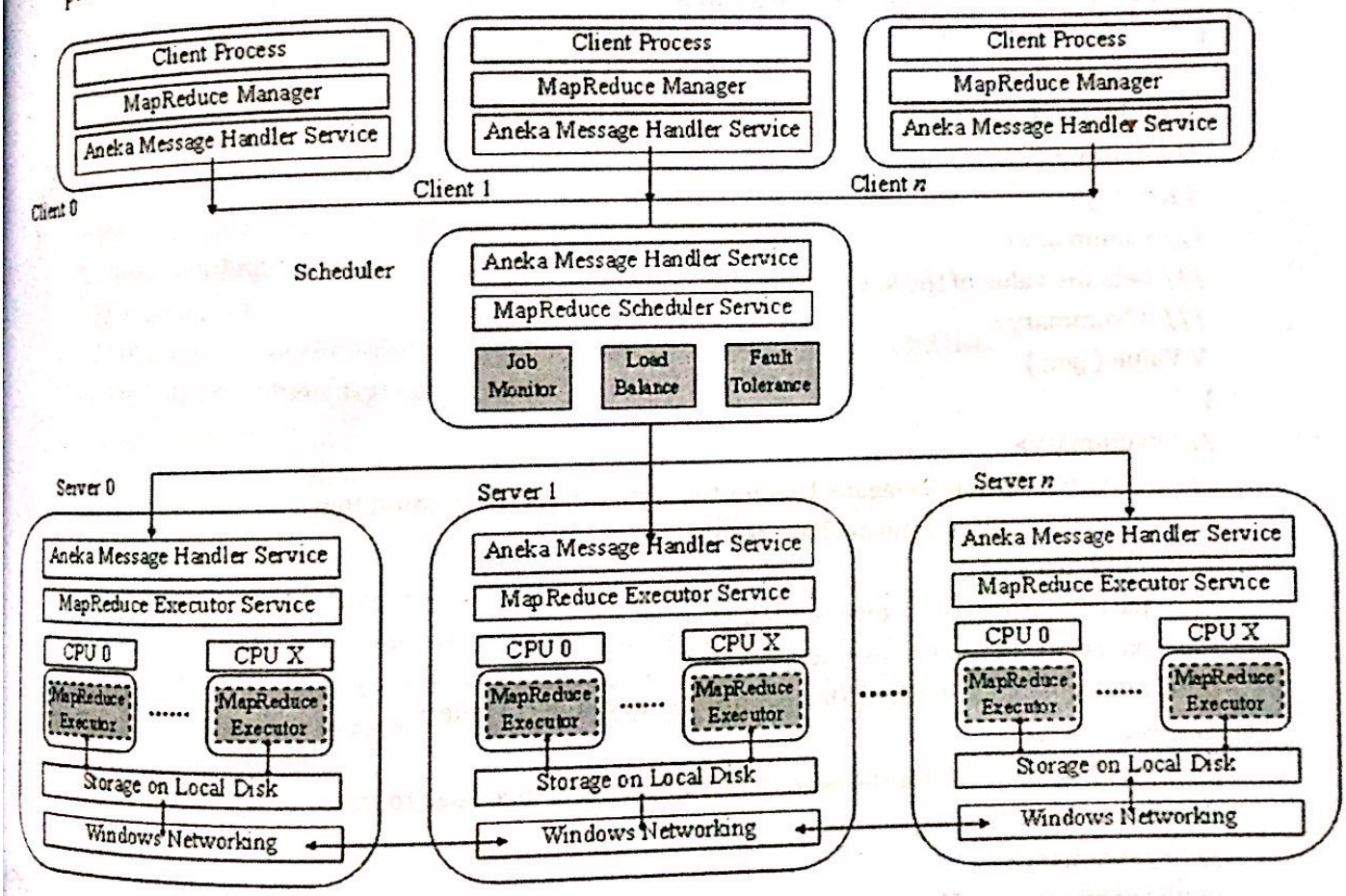


Figure 5.24: MapReduce System View

Figure 5.24 depicts an example MapReduce.NET deployment with Aneka setup. Client processes simply include the libraries necessary to connect to Aneka and submit MapReduce tasks for execution. A typical arrangement will have the MapReduce scheduler installed on the scheduler node and the MapReduce executors deployed on the executor node. Using the Windows Shared File Service, each executor node will access shared storage among all MapReduce executors.

MapReduce.NET API

The implementation of MapReduce.NET exposes APIs similar to Google MapReduce. A MapReduce.NET based application is then composed of three main components:

- **Mapper**: this class represents the base class that users can inherit to specify their map function.
- **Reducer**: it represents the base class that uses can inherit to specify their reduce function.
- **MapReduceApplication**: represents the driver of the entire application. This class is configured with the Mapper and the Reducer specific classes defined by the user and is in charge of starting the execution of MapReduce.NET, controlling it, and collecting the results.

Map API

The following code Map Related Classes and Interfaces

```
namespace Aneka.MapReduce
{
/// <summary>
/// Interface MapInput. Represents a key/value pair.
/// </summary>
public interface MapInput<K,V> : MapInput
{
 /// <summary>
 /// Gets the Key of the key/value pair.
 /// </summary>
 K Key { get; }
 /// <summary>
 /// Gets the value of the key/value pair.
 /// </summary>
 V Value { get; }
}
/// <summary>
/// Delegate MapEmitDelegate. Defines the signature of the method that
/// is called to submit intermediate results.
/// </summary>
/// <param name="key">key</param>
/// <param name="value">value</param>
public delegate void MapEmitDelegate(object key, object value);
/// <summary>
/// Class Mapper. Defines the base class that user can specialize to implement
/// the map function.
/// </summary>
public abstract class Mapper<K,V> : MapperBase
{
/// <summary>
/// Defines the Map function.
/// </summary>
/// <param name="input">input value</param>
protected abstract void Map(MapInput<K,V> input);
```

```
    }
    }
```

Users must inherit the Mapper class and then override the abstract Map method to construct a map function. The input parameter is a MapInput with a single key/value pair. Generic arguments, K and V, provide the type of key and value, respectively.

Users' specified map function is called once for each input key/value pair. Normally, the map function returns another key/value combination. Furthermore, each input key/value combination may produce a list of output key/value pairs. The MapReduce.NET runtime system collects the produced intermediate results using MapEmitDelegate. It takes as input a key and a value for the intermediate key/value combination.

All of the intermediate key/value pairs that have been gathered are sorted by key. Furthermore, all values linked with the same key are clustered together. Each group of intermediate values is sent to the reduction function, which typically performs an aggregate operation on the grouped data to generate a final result.

Reduce API

```
namespace Aneka.MapReduce
{
/// <summary>
/// Interface IReduceInputEnumerator. Defines an enumerator that iterates
/// over the values of the same key.
/// </summary>
public interface IReduceInputEnumerator<V> :IEnumerator<V>,
IreduceInputEnumerator;
/// <summary>
/// Delegate ReduceEmitDelegate. Defines the signature of the method that
/// is called to submit aggregate results.
/// </summary>
/// <param name="value">value</param>
public delegate void ReduceEmitDelegate(object value);
/// <summary>
/// Class Reducer. Defines the base class that user can specialize to implement
/// the reduce function. The reduce function defined by users will be invoked
/// to perform an aggregation operation on all the values associated with same
/// intermediate key, which are generated by the map function.
/// </summary>
public abstract class Reducer<K,V> : ReducerBase
{
/// <summary>
/// Defines the Reduce function.
/// </summary>
/// <param name="input">input value</param>
protected abstract void Reduce(IReduceInputEnumerator<V> input);
}
}
```

Users must inherit the Reducer class and then override the abstract Reduction method to specify a reduce function. The input parameter is an IReduceInputEnumerator, which is an enumerator that iterates

through each member in a single set of values associated with the same key. The IEnumerator interface is implemented by the IReduceInputEnumerator interface. Generic arguments, K and V, provide the type of key and value, respectively.

Users take each element of input values within the reduction function and then conduct an aggregate operation on all of the input values. Using ReduceEmitDelegate, the aggregated final result is delivered to the MapReduce.NET runtime system. It takes an object as an input parameter. As a result, it may take any form of data as the result.

All of the final results are divided into parts. A hash function is used on each key in the final result to divide it. The keyspace is partitioned into many parts based on user specifications, and values located in the same piece of keyspace comprise one partition. Users get the final results when the MapReduce calculation is completed.

MapReduceApplication API

Users must configure a MapReduce application using the MapReduceApplication class before it can be executed.

```
namespace Aneka.MapReduce
{
 /// <summary>
 /// Class MapReduceApplication. Configures a MapReduce application.
 /// </summary>
 public class MapReduceApplication<M, R> :
ApplicationBase<MapReduceManager<M, R>>
 where M :MapReduce.Internal.MapperBase
 where R :MapReduce.Internal.ReducerBase
 {
 /// <summary>
 /// Gets, sets a the number of partitions of the final results.
 /// </summary>
 public int Partitions { get { ... } set { .. } }
 /// <summary>
 /// Gets, sets a boolean value that indicates whether the application
 /// should download the final result files to the local machine or not.
/// </summary>
 public bool FetchResults{ get { ... } set { .. } }
 /// <summary>
 /// Gets, sets a boolean value that indicates whether the application
 /// should synchronize the execution of reducer task or not.
 /// </summary>
 public bool SynchReduce{ get { ... } set { .. } }
 /// <summary>
 /// Gets, sets a boolean value that indicates whether to combine results
 /// from the map phase in order to reduce the number of intermediate
 /// values.
 /// </summary>
 public bool UseCombiner{ get { ... } set { .. } }
 /// <summary>
 /// Gets, sets a boolean value that indicates whether the input for the
 /// map is already loaded into the storage or not.
 /// </summary>
 public bool IsInputReady{ get { ... } set { .. } }
 /// <summary>
```

```
/// Gets, sets a the number of times to re-execute a failed task.
/// </summary>
public int Attempts { get { ... } set { .. } }
/// <summary>
/// Gets, sets a the name of the log file for the application.
/// </summary>
public string LogFile{ get { ... } set { .. } }
/// <summary>
/// Fires when the MapReduce application is finished.
/// </summary>
public event EventHandler<ApplicationEventArgs>ApplicationFinished
// <summary>
/// Initializes a MapReduce application.
/// </summary>
/// <param name="configuration">Configuration information</param>
public MapReduceApplication(Configuration configuration) :
base("MapReduceApplication", configuration) { ... }
// <summary>
/// Initializes a MapReduce application.
/// </summary>
/// <param name="displayName">Application name</param>
/// <param name="configuration">Configuration information</param>
public MapReduceApplication(string displayName,
Configuration configuration) :
base(displayName, configuration) { ... }
// from the ApplicatioBase class...
/// <summary>
/// Starts the execution of the MapReduce application.
/// </summary>
public override void SubmitApplication() { ... }
/// <summary>
/// Starts the execution of the MapReduce application.
/// </summary>
/// <param name="args">Application finish callback</param>
public override void InvokeAndWait(EventHandler<ApplicationEventArgs>args)
{ ... }
}
}
```

Three different elements compose the interface of MapReduceApplication.

Type Parameters

- **MapperBase** class specifies the Mapper class defined by users.
- **ReducerBase** class specifies the Reducer class defined by users.

Configuration Parameters

- **Partitions**: The intermediate results are partition into r pieces, which correspond to r reduce tasks. ReducePartitionNumber specifies r.
- **Attempts**: Each map or reduce task may have errors during its execution. The errors may be caused by various reasons. Users can specify a number n, by using Attempts, to force the MapReduce.NET to re-execute its failed task n times for tolerating those faults that are not caused by the MapReduce application itself.

- **UseCombiner** and **SynchReduce** are parameters that control the behavior of the internal execution of MapReduce and are generally set to true.
- **LogFile**(*optional*): it is possible to store all the log messages generated by the application into a file for analyzing the execution of the application.

ApplicationBase Methods

- SubmitApplication() is called to submit the MapReduce application and input files to the MapReduce scheduler and start execution.
- ApplicationFinished is used to specify a function handler that is fired after the application is completed.
- InvokeAndWait(EventHandler<ApplicationEventArgs>args) is a convenient method that is used to start the application and wait for its termination.

It is important to define the particular types of ***Mapper*** and ***Reduce,*** which will be utilized by the application when creating a MapReduceApplication. This is accomplished by focusing on the MapReduceApplication class's template specification.

Furthermore, to construct an instance of this class, an instance of the Configuration class must be supplied, which includes all of the application's customization options. Configuration is a generic class that supports all of Aneka's programming models and includes various capabilities to help with the development of custom parameters.

The public interface of the Configuration class

```
namespace Aneka.Entity
{
 /// <summary>
 /// Class Configuration. Wraps the configuration parameters required
 /// to run distributed applications.
 /// </summary>
 [Serializable]
 public class Configuration
 {
 /// <summary>
 /// Gets, sets the user credentials to authenticate the client to Aneka.
 /// </summary>
 public virtual ICredentialUserCredential{ get { ... } set { .. } }
 /// <summary>
 /// If true, the submission of jobs to the grid is performed only once.
 /// </summary>
 public virtual bool SingleSubmission{ get { ... } set { ... } }
 /// <summary>
 /// If true, uses the file transfer management system.
 /// </summary>
 public virtual bool UseFileTransfer{ get { ... } set { ... } }
 /// <summary>
 /// Specifies the resubmission strategy to adopt when a task fails.
 /// </summary>
public virtual ResubmitModeResubmitMode{ get { ... } set { ... } }
```

```
/// <summary>
/// Gets and sets the time polling interval used by the application to query
/// the grid for job status.
/// </summary>
public virtual int PollingTime{ get { ... } set { ... } }
/// <summary>
/// Gets, sets the Uri used to contact the Aneka scheduler service which is
/// the gateway to Aneka grids.
/// </summary>
public virtual Uri SchedulerUri{ get { ... } set { ... } }
/// <summary>
/// Gets or sets the path to the local directory that will be used
/// to store the output files of the application.
/// </summary>
public virtual string Workspace { get { ... } set { ... } }
/// <summary>
/// If true all the output files for all the work units are stored
/// in the same output directory instead of creating sub directory
/// for each work unit.
/// </summary>
public virtual bool ShareOutputDirectory{ get { ... } set { ... } }
/// <summary>
/// If true activates logging.
/// </summary>
public virtual bool LogMessages{ get { ... } set { ... } }
/// <summary>
/// Creates an instance of the Configuration class.
/// </summary>
public Configuration() { ... }
/// <summary>
/// Loads the configuration from the default config file.
/// </summary>
/// <returns>Configuration class instance</returns>
public static Configuration GetConfiguration() { ... }
/// <summary>
/// Loads the configuration from the given config file.
/// </summary>
/// <param name="confPath">path to the configuration file</param>
/// <returns>Configuration class instance</returns>
public static Configuration GetConfiguration(string confPath) { ... }
/// <summary>
/// Gets or sets the value of the given property.
/// </summary>
/// <param name="propertyName">name of the property to look for</param>
/// <returns>Property value</returns>
public string this[string propertyName] { get { ... } set { ... } }
/// <summary>
/// Gets or sets the value of the given property.
/// </summary>
/// <param name="propertyName">name of the property to look for</param>
```

```
/// <param name="bStrict">boolean value indicating whether to raise
/// exceptions if the property does not exist</param>
/// <returns>Property value</returns>
public string this[string propertyName, bool bStrict]
{ get { ... } set { ... } }
/// <summary>
/// Gets or sets the value of the given property.
/// </summary>
/// <param name="propertyName">name of the property to look for</param>
/// <returns>Property value</returns>
public string this[string propertyName] { get { ... } set { ... } }
/// <summary>
/// Gets the property group corresponding to the given name.
/// </summary>
/// <param name="groupName">name of the property group to look for</param>
/// <returns>Property group corresponding to the given name, or
/// null</returns>
public PropertyGroupGetGroup(string groupName) { ... }
/// <summary>
/// Adds a property group corresponding to the given name to the
/// configuration if not already present.
/// </summary>
/// <param name="groupName">name of the property group to look for</param>
/// <returns>Property group corresponding to the given name</returns>
public PropertyGroupAddGroup(string groupName) { ... }
/// <summary>
/// Adds a property group corresponding to the given name to the
/// configuration if not already present.
/// </summary>
/// <param name="group">name of the property group to look for</param>
/// <returns>Property group corresponding to the given name</returns>
public PropertyGroupAddGroup(PropertyGroup group) { ... }
/// <summary>
/// Removes the group of properties corresponding to the given name from the
/// configuration if present.
/// </summary>
/// <param name="groupName">name of the property group to look for</param>
/// <returns>Property group corresponding to the given name if successfully
/// removed, null otherwise</returns>
public PropertyGroupRemoveGroup(string groupName) { ... }
/// <summary>
/// Checks whether the given instance is a configuration object and
/// whether it contains the same information of the current instance.
/// </summary>
/// <param name="other">instance to compare with</param>
/// <returns>true if the given instance is of type Configuration
/// contains the same information of the current instance.</returns>
public override bool Equals(object other) { ... }
}
}
```

An instance of the Configuration can be created programmatically or by reading the application configuration file that comes with any .NET executable application. In case we provide the configuration parameters through the application configuration file it is possible to get the corresponding Configuration instance simply by calling the static method *Configuration.GetConfiguration()* or by using the overloaded version that allows us to specify the path of the configuration file. These methods expect an XML file following the structure as below.

The MapReduce.NET application's setup is an example of the Configuration class's extensibility feature. User properties may be integrated into the configuration by keeping them into groups and adding the groups to the configuration file or the Configuration instance. Once added to the file, the runtime will import them into the configuration object and expose them in the format: GroupName.PropertyName.

For example, if we wish to get the MapReduce group's UseCombiner, we may use the following statements:

```
Configuration conf = Configuration.GetConfiguration();
string useCombinerProperty = conf["MapReduce.UseCombiner"];
```

MapReduce Configuration File

```
<?xml version="1.0" encoding="utf-8" ?>
<Aneka>
    <UseFileTransfer value="false" />
    <Workspace value="WordCounter/workspace" />
    <SingleSubmission value="false" />
    <ResubmitMode value="AUTO" />
    <PollingTime value ="1000" />
    <LogMessages value="true" />
    <ScheduleUri value="tcp://localhost:9099/Aneka" />
    <UserCredential type= "Aneka.Security.Windows.WindowsCredentials"
     assembly="Aneka.dll">
        <WindowsCredentials username="aneka" password="aneka" domain="" />
    </UserCredential>

    <Groups>
        <Group name="MapReduce">
            <property name="LogFile" value="WordCount.log" />
            <property name="FetchResults" value="true" />
            <property name="UseCombiner" value="true" />
            <property name="SynchReduce" value="true" />
            <property name="IsInputReady" value="false" />
            <property name="Partitions" value="1" />
            <property name="Attempts" value="3" />
        </Group>
    </Groups>
</Aneka>
```

A particular tag *<Group name="MapReduce"> ... </Group>* contains the precise parameters required by the MapReduce paradigm. This tag has the same attributes that the *MapReduceApplication* class exposes. The user may modify the MapReduce model's execution by simply changing the values of the attributes in the configuration file. The runtime will automatically read these configuration settings and associate them with the MapReduceApplication class's attributes. It is possible to leave out some of the properties mentioned inside this tag, or even the entire tag. In these instances, the runtime will update the configuration with the missing parameters by supplying default values.

The only other parameters in the Configuration that are relevant are the SchedulerUri and the Workspace. The first identifies the Aneka scheduler's URI, while the second is the link to the local directory where the output files are stored. All of the other parameters explicitly supplied by the Configuration class are of broad use for the other models but are unimportant for MapReduce.NET operation.

BENEFITS OF MAPREDUCE

It is fault-tolerant; during the middle of a map-reduce job, if a machine carrying a few data blocks fails, the architecture handles the failure.

Each node sends a status update to the master node regularly. If a slave node fails to transmit its notice, the master node reassigns the slave node's current job to another available node in the cluster.

Because MapReduce employs HDFS as its storage system, data processing is fast. MapReduce processes gigabytes of unstructured huge quantities of data in a matter of minutes.

By splitting the jobs, MapReduce tasks may process several portions of the same dataset concurrently. This has the advantage of completing tasks in less time.

Multiple copies of the same data are delivered to various network nodes. As a result, in the event of a failure, alternative copies are quickly available for processing with no loss.

Hadoop is an extremely scalable platform. Traditional RDBMS systems are not scalable as data volumes rise. MapReduce allows you to run programs over a large number of nodes, with data sizes ranging from terabytes to petabytes.

Hadoop's scale-out functionality, along with MapReduce programming, allows you to store and process data in a very efficient and cost-effective manner. Savings on terabytes of data may be in the hundreds of dollars.

PARALLEL EFFICIENCY OF MAP-REDUCE

It is worth trying to figure out what is the parallel efficiency of MapReduce. Now let us assume that the data produced after the map phase is **σ** times the original data size **D (σD)**, further, we assume that there are **P** processors which in turn perform map and reduce operations depending on which phase they are used in, so we do not have any wastage of processors. Also, the algorithm itself is assumed to do **wD** useful work, **w** is not necessarily a constant it could be D^2 or something like that, but the point is that there is some amount of useful work being done even if you have a single processor and that's **wD**. Now let us look at the overheads of doing the computation **wD** using MapReduce. After the map operation, instead of **D** data items, we have **σD** data items in P pieces, so each mapper writes $\frac{\sigma D}{P}$ data to their local disk. So, there is some overhead associated with writing this data. Next, this data has to be read by each reducer before it can begin 'reduce' operations, so each reducer has to read $\frac{\sigma D}{P^2}$ that is one P^{th} of the data from a particular mapper. Since there are **P** different reducers, one P^{th} of the data in a map goes to each reducer and it has to read **P** of these from **P** different mappers once again getting us the communication time that a reducer spends getting the data it requires from the different mappers as $\frac{\sigma D}{P}$.

$$\textbf{Overheads: } \frac{\sigma D}{P}$$

So, the total overhead that is work that would not have to be done if we did not have parallelism, that is writing data to disk and reading data from remote mappers is $\frac{2D}{P}$. Now if you look at the efficiency using this overhead, we get the **wD** is the time it takes on one processor which is the useful work that needs to be done; $\frac{wD}{P}$ if we had **P** processors but we have this extra overhead of $\frac{2\sigma D}{P}$. We have got a constant **c** which essentially measures how much time it takes to write one data item to disk or to read a data item remotely from a different mapper. Now you assumed both of these to be the same constant.

$$\cdot MR = \frac{wD}{P\left(\frac{wD}{P} + 2C\frac{sD}{P}\right)} = \frac{1}{1 + \frac{2c}{w}s}$$

Simplifying this parallel efficiency, we get an expression that is surprisingly independent of **P**, dependent on **w** and **σ** in an important way though. It is nice that it is independent of **P** because it indicates that the MapReduce process is scalable and you can get large efficiencies even with a large number of processors, there is no dependency on the processors with the efficiency. However, it is dependent on **σ** and **w**. In particular, as long as the amount of useful work that you do per data item grows, the efficiency ends up going close to 1, which is nice. On the other hand, if the extra data that you produce after the map phase grows then efficiency can suffer and reduce considerably. So, it is very important to use things like combiners to ensure that too much data does not get blown up after the map phase is done. In particular, if **σ** ends up being dependent on **P** then the statement here that efficiency is independent of **P** no longer holds so be careful when computing this expression. **σ** is quite likely to be dependent on **P** even though the expression hides that dependence.

This is quite insightful and will compute the parallel efficiency of our word-counting problem using this formula. When we were counting words we had **n** documents, **m** total possible words, each occurring say, **f** times per document on the average, so the total volume of data to be processed is **nmf**. Now, after the map phase using combiners, we produce **P** into **m** partial counts, that is **m** different partial counts per mapper and their **P** mappers. At worst, there will be **mP** partial counts, some mappers may not produce counts for words that do not occur there, but we do not worry about that. We assume that at worst there will be **mP** partial counts which give us **σ** as the data after the map, as compared to the data before it and there you get $\frac{P}{nf}$.

$$s = \frac{mP}{nmf} = \frac{P}{nf}$$

Now you see the ratio $\frac{n}{P}$ creeping in through the **σ** so the fact that you had shrinkage of data will end up helping us. Let us see, if we plug this into our equation, we get the parallel efficiency is the expression as below:

$$\frac{1}{1 + \frac{2cP}{wnf}}$$

Assuming **w** and **c** are the same, which is a simplifying assumption in our case, we get an expression that looks like this:

$$\frac{1}{1 + \frac{2P}{nf}}$$

Notice here that as the ratio $\frac{n}{P}$ goes very large, which means you have large volumes of data large number of documents then the efficiency goes to 1 as we would expect for a scalable parallel algorithm.

$$^{e}MR = \frac{1}{1 + \frac{2cP}{wnf^2}} = \frac{1}{\frac{2P}{nf^2}}$$

To summarize, we have seen that MapReduce is indeed scalable in the previous case we thought that our expression was independent of **D** and **P**, but we found that that creeps in because of the **σ**. So, the Sigma is actually, extremely important the expansion or shrinkage factor after the map phase is critical for the efficiency of MapReduce algorithms since we have to make sure that is as smallest possible if you want a

scalable implementation. In particular, it should certainly not grow with the number of processors otherwise we will have a blow-up inefficiency. If σ was growing with P, then you would end up having efficiency going down to 0 as a large number of processors were used. It appears that this is indeed the case here but notice that if you use a large number of processors and keep the same problem size you will get low efficiency as you have a lot of processors doing no work. But if you used larger and larger document sets then naturally, you would get better efficiencies which is what a scalable algorithm is supposed to be.

ENTERPRISE BATCH PROCESSING USING MAP-REDUCE

Data is the new money in the contemporary world. The data generated by today's enterprises has been increasing at exponential rates in size from the most recent couple of years. Data-intensive calculations are widespread in many application areas. Computational science is one of the most well-known. People who conduct scientific simulations and experiments are often eager to generate, review, and analyze large volumes of data. Telescopes scanning the sky create hundreds of terabytes of data each second; the collection of sky images often reaches petabytes over a year. Bioinformatics applications mine databases containing terabytes of data. Earthquake simulators manage massive volumes of data generated by detecting the Earth's earthquakes all across the planet.Other IT business fields, in addition to scientific computing, also require help from the data-intensive computation. The transaction data of an eCommerce site may exceed millions per month, customer data for any telecom firm will most certainly reach 60-100 terabytes. This volume of data is mined not only for billing purposes, but also to find events, trends, and patterns that help these firms provide better service. Businesses that effectively maximize their value will have a significant influence on their worth as well as the success of their consumers.

With such a large data volume, it will be difficult for a single server – or node – to handle. As a result, code that runs on several nodes is required. Because writing distributed systems present an infinite number of challenges. MapReduce is a framework that lets users develop code that runs on numerous nodes without worrying about fault tolerance, dependability, synchronization, or availability. Batch processing is a type of automated task that performs computations regularly. It executes the processing code on a group of inputs known as a batch. The task will often read batch data from a database and save the results in the same or a separate database. A batch processing job can consist of reading all of the sale logs from an online store for a single day and aggregating them into statistics for that day such as the number of users per country, the average spent amount, etc. Doing this periodically provide insights into the data patterns. Depending upon the size of the data that need to be processed the processing time varies and higher than the normal data processing.

As the batch processing is done with the cumulative transactions in a group so once batch processing has begun, no user participation is necessary. This distinguishes batch processing from normal transaction processing. While batch processing may be done at any time, it is best suited for end-of-day processing, such as processing a bank's reports at the end of the day or creating monthly or bimonthly payrolls.

The batch processing mechanism processes the data blocks that have previously been stored over some time. For example, a large financial business may complete all of its transactions in a single week. This data comprises millions of records for a single day, which may be saved as a file or record. This file will be processed at the end of the day for different analyses that the company wishes to do. Processing that file will take a significant amount of time so it is not feasible to perform it very frequently. In the point of performance, the latency of batch processing will be in a minute to hours. So, batch processing is useful when you do not require real-time analytics results and it is more necessary to analyze vast amounts of data to gain more deep insights than it is to acquire quick analytics results.

Batch processing frameworks are great for handling exceedingly big datasets that need a substantial amount of computation. Such datasets are generally limited and persistent, that is, they are kept in some form of permanent storage. Batch processing is appropriate for non-time-sensitive processing since processing a big dataset takes time. Apache Hadoop's MapReduce is the most widely used batch processing framework. Hadoop MapReduce is the finest framework for batch data processing. The diagram below explains in detail how Hadoop processes data using MapReduce.

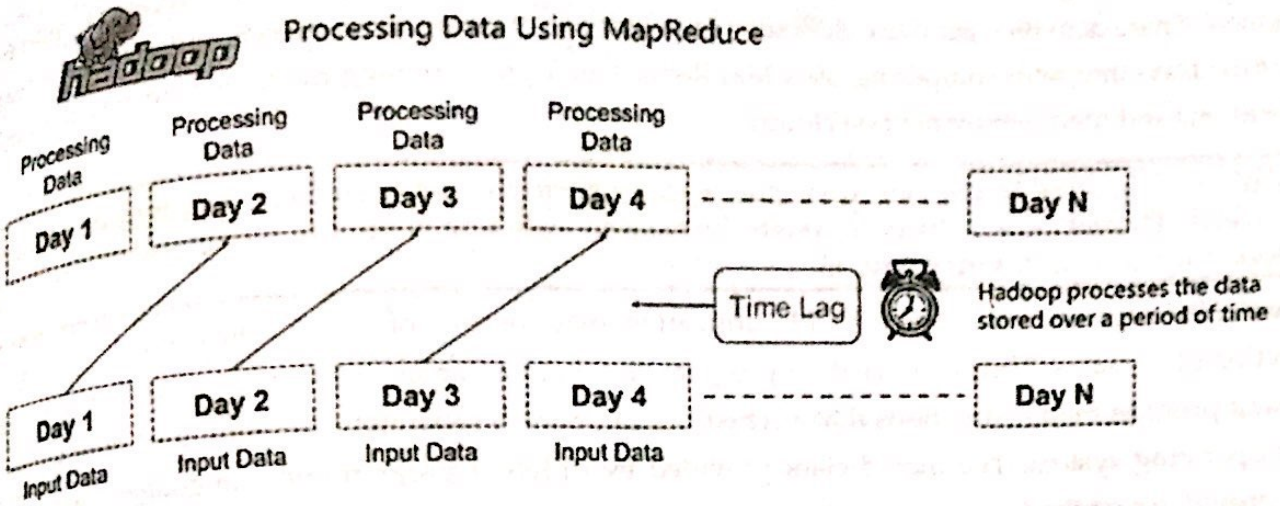


***Figure 5.25:* Data processing using Hadoop MapReduce**

Hadoop MapReduce is a Java-based system for processing large datasets. It reads data from the HDFS and divides the dataset into smaller pieces. Each piece is then scheduled and distributed for processing among the nodes available in the Hadoop cluster. Each node performs the required computation on the chunk of data and the intermediate results obtained are written back to the HDFS. These intermediate outputs may then be assembled, split, and redistributed for further processing until final results are written back to HDFS.

As already discussed above, the MapReduce data processing programming model consists of two different jobs executed by programs: a Map job and a Reduce job. Typically, the Map operation begins by turning a collection of data into another set of data in which individual pieces of the data are broken down into tuples consisting of key-value pairs. One or more Map tasks can then shuffle, sort, and process these key-value pairs. The Reduce task typically takes as input the results of a Map task and merges those data tuples into a smaller collection of tuples.

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

Comparisons Between Thread, Task and Map-Reduce

As previously mentioned in this section, there are several computing categories such as high-performance computing (HPC), high-throughput computing (HTC), etc. Various programming methods may be employed to take advantage of these computing techniques. High throughput computing is concerned with delivering large amounts of compute in the form of transactions, which is accomplished by multithreading, and multithreaded programming may now be extended to a distributed environment as well. Similarly, High-throughput computing (HTC) refers to the usage of a large number of computing resources over a lengthy period to complete a computational job. The most obvious and frequent method for designing parallel and distributed computing applications and gain the advantages of high-throughput computing is through task programming. A task describes a program that may need input files and generate output files as a result of its execution and applications are a collection of tasks.Tasks are submitted for execution, and their output data is gathered at the conclusion. The way tasks are produced, the sequence in which they are executed, and whether they need data interchange to distinguish the application models that come under the task programming umbrella. Similarly, another important computing category is data-intensive computing which deals with enormous amounts of data. Several application domains, from computational research to

social networking, generate vast amounts of data that must be effectively stored, accessed, indexed, and evaluated. These activities get more difficult as the amount of information collects and grows at a faster rate over time.Data-intensive computing uses MapReduce as a programming model for creating data-intensive applications and their deployment on clouds.

A Task may be used to indicate what you want to perform, and then that Task may be attached to a Thread. Threads are utilized to finish the task by splitting it up into pieces and executing them individually in a distributed system.

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

A task is anything that you want to be completed that is a higher-level abstraction on top of threads. It is a collection of software instructions stored in memory. When a software instruction is placed into memory, it is referred to as a process or task. The Task can inform you if it has been completed and whether the procedure has produced a result. A task will use the Threadpool by default, which saves resources because creating threads is costly as a large block of memory has to be allocated and initialized for the thread stack and system calls need to be made to create and register the native thread with the host OS. When requests are frequent and lightweight, as they are in most server applications, establishing a new thread for each request might take substantial computer resources.

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

Multithreaded programming and multiprocessing technologies, as well as multi-core technology, aid in attaining parallelism on a single computer that may be used to accelerate programs.Currently, all of the most common operating systems allow multithreading, regardless of whether the underlying hardware expressly supports actual parallelism or not. If many processors or cores are available, real parallelism can be achieved by using them at the same time; otherwise, multithreading is achieved by interleaving the execution of many threads on the same processing unit. Multithreaded programming enables parallelism within the confines of a single processor. Applications that require a high level of parallelism cannot be handled by traditional multithreaded programming and must rely on distributed infrastructures such as clusters, grids, or clouds. The usage of these facilities necessitates the development of programs and the usage of certain APIs, which may need considerable changes to existing programs. To overcome this issue, Aneka provides the Thread Programming Model, which extends the multithreaded programming philosophy beyond the bounds of a single node and enables the use of heterogeneous distributed infrastructure.

Programming languages define the abstractions of process and thread in their class libraries to facilitate multithreaded programming. POSIX is a prominent standard for thread operations and thread synchronization that is supported by all Linux/UNIX operating systems and is offered as an extra library for the Windows operating system family. A typical implementation of POSIX is provided as a function library in C/C++. New-generation languages, such as Java and C# (.NET), provide a set of abstractions for thread management and synchronization that is compliant and adheres to the object-oriented architecture as closely as possible.Aneka provides the Thread Programming Model, which extends the concept of multithreaded programming beyond the limitations of a single node and allows execution to be performed on heterogeneous distributed infrastructure.

Similarly, the most natural technique of dividing an application's computation among a group of nodes is task-based programming. The idea of a task, which represents a series of actions that may be isolated and executed as a single unit, is the core abstraction of task-based programming. A job might be as basic as a shell program or as sophisticated as a piece of code that requires a certain runtime environment to execute. Tasks frequently need input files for execution and generate output files as a result.The task can return a result. There is no direct mechanism to return the result from a thread. It is usually recommended to utilize tasks rather than threads since they are formed on the thread pool, which already includes system-generated threads to boost performance.A task by default runs in the background, and the task cannot be in the foreground. On the other hand, a thread can be background as well as the foreground.Aneka supports task-based programming and serves as a real example of a framework that facilitates the development and execution of task-based distributed applications.

As mentioned earlier, MapReduce is associated with data-intensive applications which process or generate large amounts of data and may also be compute-intensive. The volumes of data that prompted the concept of data-intensive computation have varied over time. Data-intensive computing is a discipline that began with high-speed WAN applications but nowadays it is the province of storage clouds, with data dimensions reaching terabytes, if not petabytes, that is referred to as Big Data which represents data in a semi-structured or unstructured form. As a result, traditional techniques based on relational databases are incapable of serving data-intensive applications efficiently. To address such issues, new techniques and storage models have been developed. The major efforts in the area of storage systems have been devoted to the creation of high-performance distributed file systems, storage clouds, and NoSQL-based systems. The most significant improvement in the assistance of programming data-intensive applications has been the advent of MapReduce.Google presented MapReduce as a straightforward technique to processing massive amounts of data based on the creation of two functions, map and reduce, which are applied to the data in a two-phase process. The map step first retrieves useful information from the data and saves it in key-value pairs, which are then aggregated together in the reduction step. Despite its limitations, this paradigm is effective in a variety of application settings. MapReduce can be implemented in the Apache Hadoop framework using programming languages such as Java, Python, or C++. Equally, Aneka, like with thread and task programming models, provides APIs for designing and implementing MapReduce applications as well. The .NET framework and C# are preferred for developing task-based applications using Aneka.

OBJECTIVE QUESTIONS

1. **What type of computing technology refers to services and applications that typically run on a distributed network through virtualized resources?**
   a. Distributed Computing
   b. Cloud Computing
   c. Soft Computing
   d. Parallel Computing
2. **Multithreading is also called as _____**
   a. Concurrency
   b. Simultaneity
   c. Crosscurrent
   d. Recurrent
3. **A single sequential flow of control within a program is _____**
   a. Process
   b. Task
   c. Thread
   d. Structure
4. **Java extension used in threads?**
   a. java.lang.Thread
   b. java.language.Thread
   c. java.lang.Threads
   d. java.Thread
5. **A method that must be overridden while extending threads.**
   a. run()
   b. start()
   c. stop()
   d. paint()
6. **An interface that is implemented while using threads.**
   a. java.lang.Run
   b. java.lang.Runnable
   c. java.lang.Thread
   d. java.lang.Threads
7. **A thread becomes non-runnable when?**
   a. Its stop method is invoked
   b. Its sleep method is invoked
   c. Its finish method is invoked
   d. Its init method is invoked
8. **A method used to temporarily release time for other threads.**
   a. yield()
   b. set()
   c. release()
   d. start()
9. **A method used to force one thread to wait for another thread to finish.**
   a. join()
   b. connect()
   c. combine()
   d. concat()
10. **_____ makes it possible for two or more activities to execute in parallel on a single processor.**
    a. Multithreading
    b. Threading
    c. SingleThreading
    d. Both Multithreading and SingleThreading
11. **The threads created using the Thread class are called the?**
    a. child threads
    b. B. parent threads
    c. base threads
    d. None of the above
12. **Which method for making a thread pause for a specific period.**
    a. push()
    b. B. wait()
    c. C. sleep()
    d. D. stack()
13. **Choose the correct statement about process-based multitasking.**
    a. Feature that allows our computer to run two or more programs concurrently
    b. A program that acts as a small unit of code that can be dispatched by the scheduler
    c. Only A program that acts as a small unit of code that can be dispatched by the scheduler
    d. Both A feature that allows our computer to run two or more programs concurrently & A program that acts as a small unit of code that can be dispatched by the scheduler

14. **Choose the namespace which supports multithreading programming?**
    a. System.net b. System.Linq c. System.Threading d. All of the above
15. **What is synchronization about a thread?**
    a. It is a process of handling situations when two or more threads need access to a shared resource
    b. It is a process by which many threads can access the same shared resource simultaneously
    c. It is a process by which a method can access many different threads simultaneously
    d. It is a method that allows too many threads to access any information they require
16. **A _____ node acts as the Slave and is responsible for executing a task assigned to it by the JobTracker.**
    a. MapReduce b. Mapper c. TaskTracker d. JobTracker
17. **The use of distributed computing facilities for applications requiring large computing power over a long period refers to:**
    a. High-performance computing b. High-throughput computing
    c. Data-intensive computing d. Many-task computing
18. **Point out the correct statement.**
    a. MapReduce tries to place the data and the compute as close as possible
    b. Map Task in MapReduce is performed using the Mapper() function
    c. Reduce Task in MapReduce is performed using the Map() function
    d. All of the mentioned
19. **_____ part of the MapReduce is responsible for processing one or more chunks of data and producing the output results.**
    a. Map task b. Mapper c. Task execution d. All of the mentioned
20. **_____ function is responsible for consolidating the results produced by each of the Map() functions/tasks.**
    a. Reduce b. Map c. Reducer d. All of the mentioned
21. **Use of distributed computing facilities for solving problems that need large computing power refers to:**
    a. High-performance computing b. High-throughput computing
    c. Data-intensive computing d. Many-task computing
22. **Point out the wrong statement.**
    a. A MapReduce job usually splits the input data set into independent chunks which are processed by the map tasks in a completely parallel manner
    b. The MapReduce framework operates exclusively on <key, value> pairs\
    c. Applications typically implement the Mapper and Reducer interfaces to provide the map and reduce methods
    d. None of the mentioned
23. **Although the Hadoop framework is implemented in Java, MapReduce applications need not be written in _____**
    a. Java b. C
    c. C# d. None of the mentioned
24. **_____ is a utility that allows users to create and run jobs with any executables as the mapper and/or the reducer.**
    a. Hadoop Strdata b. Hadoop Streaming c. Hadoop Stream d. None of the mentioned

25. _____ maps input key/value pairs to a set of intermediate key/value pairs.
    a. Mapper
    b. Reducer
    c. Both Mapper and Reducer
    d. None of the mentioned

26. The number of maps is usually driven by the total size of _____
    a. inputs
    b. outputs
    c. tasks
    d. one of the mentioned

27. Running a _____ program involves running mapping tasks on many or all of the nodes in our cluster.
    a. MapReduce
    b. Map
    c. Reducer
    d. All of the mentioned

28. The specific class of embarrassingly parallel applications for which the tasks are identical and differ only by the specific parameters used to execute them is known as:
    a. Embarrassingly parallel applications
    b. Parameter sweep applications
    c. MPI applications
    d. Workflow applications with task dependencies

29. What are the programming models supported by Aneka?
    a. Thread
    b. Task
    c. MapReduce
    d. All of the above

30. This provides distribution by harnessing the computing power of several computing nodes
    a. Task Programming
    b. Thread Programming
    c. Map Programming
    d. Object-Oriented Programming

31. Which of the following is not the category of task programming?
    a. High-performance computing
    b. High-throughput computing
    c. Data-intensive computing
    d. Many-task computing

32. Which of the following is not the framework for task computing?
    a. Condor
    b. Globus Toolkit
    c. VMWare
    d. Aneka

33. It is concerned with delivering large amounts of computing in the form of transactions
    a. Data-intensive computing
    b. Throughput Computing
    c. Concurrent Computing
    d. Performance Computing

34. Which one of the following options can be considered as the Cloud?
    a. Hadoop
    b. Intranet
    c. Web Applications
    d. All of the mentioned

35. Which of the following is a task-based application model?
    a. Embarrassingly parallel applications
    b. Parameter sweep applications and MPI applications
    c. Workflow applications with task dependencies
    d. All of the above

36. A specification for developing parallel programs that communicate by exchanging messages is called:
    a. Message Passing Interface
    b. Parameter Sweep Application
    c. Workflow Control Application
    d. Parallel Applications

37. Which of the following is not the step of developing application with Task Model:
    a. Define Classes WithItask Interface
    b. Create a properly Configured AnekaApplication instance
    c. Create ITask instances, wrap them into AnekaTask instances and Execute it
    d. All of the above

38. AnekaApplication Class Provides which of the following operations:
    a. Static and Dynamic task submission
    b. Application state and task state monitoring
    c. Event-based notification of tasks completion or failure
    d. All of the above mentioned

QUESTION

1. What are threads? Briefly explain the differences between a thread and a process.
2. Briefly explain the differences between local and distributed threads.
3. What are the challenges in developing parallel applications?
4. Explain how you would go about designing and developing a parallel application.
5. What are the limitations of distributed threads compared to local threads?
6. Modify the example for calculating Pi so that the number of darts and repetitions are passed as command-line arguments to the program.
7. Remodel and implement the example for trigonometric calculations, such that MathExample functions as a multi-threaded server that listens to computation requests from clients via sockets, and forwards them to the Aneka runtime for execution.
8. Write a program to print "Hello World" using the Aneka Thread Programming model use Single Thread.
9. Write a program to sum the two numbers using the Aneka Task Programming model.
10. Write a program to compute the matrix addition using Aneka Thread Programming Model.
11. What is throughput computing and what does it aim to achieve?
12. What is multiprocessing? Describe the different techniques for implementing multiprocessing.
13. What is multicore technology and how does it relate to multiprocessing?
14. What is multithreading and how does it relate to multitasking?
15. Describe the relationship between a process and a thread.
16. Does the parallelism of applications depend on parallel hardware architectures?
17. Describe the principal characteristics of a thread from a programming point of view and the uses of threads for parallelizing application execution.
18. Explain is POSIX. Describe the support given for programming with threads in new-generation languages such as Java or C#.
19. What do the terms logical thread and physical thread refer to?
20. What are the common operations implemented for a thread?
21. Which kind of support does Aneka provide for multithreading?
22. Describe the major differences between Aneka threads and local threads.
23. What are the limitations of the Thread Programming Model?
24. . What is a task? How does task computing relate to distributed computing?
25. List and explain the computing categories that relate to task computing.
26. What are the main functionalities of a framework that supports task computing?
27. List some of the most popular frameworks for task computing.
28. Explain parameter sweep application.

29. What is MPI? What are its main characteristics? Describe.
30. What is a workflow? What are the additional properties of this application model concerning an embarrassingly parallel application?
31. How does Aneka support task computing?
32. What are the main components of the Task Programming Model?
33. Discuss the differences between ITask and AnekaTask.
34. Discuss the differences between static and dynamic task submission.
35. Discuss the facilities and the general architecture provided by Aneka for the movement of data for task-based applications.
36. How it is possible to run a legacy application using the Task Programming Model?
37. Does Aneka provide any feature for leveraging the Task Programming Model from other technologies and platforms?
38. What is data-intensive computing? Describe the characteristics that define this term.
39. What are the characterizing features of Big Data?
40. List some of the important storage technologies that support data-intensive computing and describe one of them.
41. What are the requirements of a programming platform that supports data-intensive computations?
42. What is MapReduce? Describe the kinds of problems MapReduce can solve and give some real examples.
43. What are the major components of the Aneka MapReduce Programming Model?
44. What do you understand by MapReduce? Explain how MapReduce works.
45. Design and implement a simple program that uses MapReduce for the computation of Pi.

□□□