FUTURE VISION BIE

One Stop for All Study Materials

& Lab Programs

Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page

Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU,
CSE – Computer Science Engineering,
ISE – Information Science Engineering,
ECE - Electronics and Communication Engineering
& MORE...

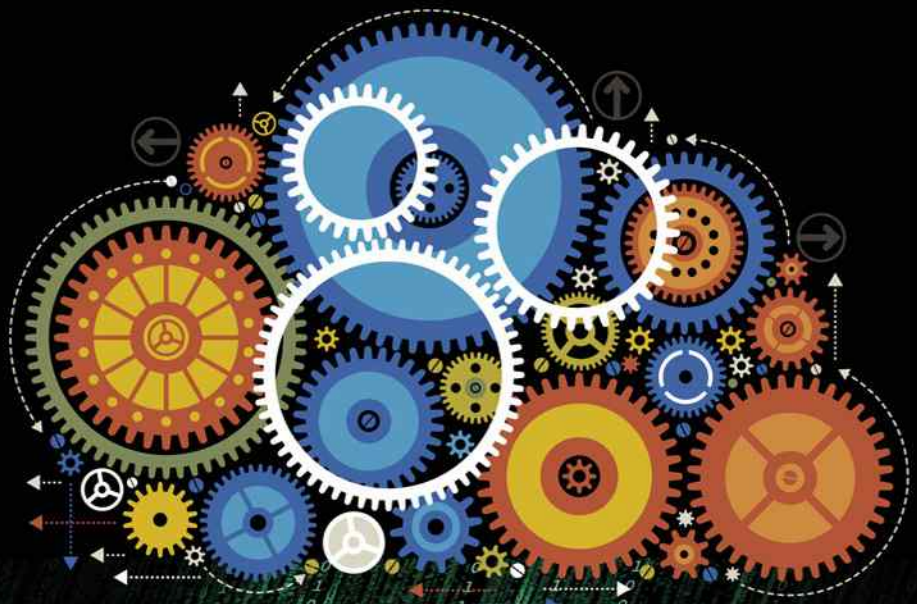
Join Telegram to get Instant Updates: https://bit.ly/VTU_TELEGRAM

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

WHATSAPP SHARE: https://bit.ly/FVBIESHARE



MASTERING CLOUD COMPUTING

FOUNDATIONS AND APPLICATIONS PROGRAMMING

MORGAN KAUFMANN

Rajkumar Buyya, Christian Vecchiola, S. Thamarai Selvi

CHAPTER

Concurrent Computing
Thread Programming

6

Throughput computing focuses on delivering high volumes of computation in the form of transactions. Initially related to the field of transaction processing [60], throughput computing has since been extended beyond that domain. Advances in hardware technologies led to the creation of multicore systems, which have made possible the delivery of high-throughput computations, even in a single computer system. In this case, throughput computing is realized by means of multiprocessing and multithreading. *Multiprocessing* is the execution of multiple programs in a single machine, whereas *multithreading* relates to the possibility of multiple instruction streams within the same program.

This chapter presents the concept of multithreading and describes how it supports the development of high-throughput computing applications. It discusses how multithreaded programming, originally conceived to be contained within the boundaries of a single machine, can be extended to a distributed context and which limitations apply. The Aneka Thread Programming Model will be taken as a reference model to review a practical implementation of a multithreaded model for computing clouds.

6.1 Introducing parallelism for single-machine computation

Parallelism has been a technique for improving the performance of computers since the early 1960′s, when Burroughs Corporation designed the D825, the first MIMD multiprocessor ever produced. From there on, a variety of parallel strategies have been developed. In particular, *multiprocessing*, which is the use of multiple processing units within a single machine, has gained a good deal of interest and gave birth to several parallel architectures.

One of the most important distinctions is made in terms of the symmetry of processing units. *Asymmetric multiprocessing* involves the concurrent use of different processing units that are specialized to perform different functions. *Symmetric multiprocessing* features the use of similar or identical processing units to share the computation load. Other examples are *nonuniform memory access (NUMA)* and *clustered multiprocessing*, which, respectively, define a specific architecture for accessing a shared memory between processors and the use of multiple computers joined together as a single virtual computer.

Symmetric and asymmetric multiprocessing are the techniques used to increase the performance of commodity computer hardware. The introduction of *graphical processing units (GPUs)*, which

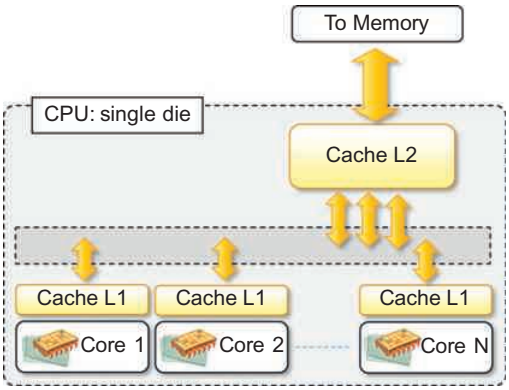


FIGURE 6.1

Multicore processor.

are *de facto* processors, is an application of asymmetric processing, whereas multicore technology is the latest evolution of symmetric multiprocessing. Multiprocessor and especially multicore technologies are now of fundamental importance because of the physical constraint imposed on frequency scaling,[1] which has been the common practice for performance gain in recent years. It became no longer possible to increase the frequency of the processor clock without paying in terms of power consumption and cooling, and this condition became unsustainable in May 2004, when Intel officially cancelled the development of two new microprocessors in favor of multicore development.[2] This date is generally considered the end of the frequency-scaling era and the beginning of multicore technology. Other issues also determined the end of frequency scaling, such as the continuously increasing gap between processor and memory speeds and the difficulty of increasing the instruction-level parallelism[3] in order to keep a single high-performance core busy.

Multicore systems are composed of a single processor that features multiple processing cores that share the memory. Each core has generally its own L1 cache, and the L2 cache is common to all the cores, which connect to it by means of a shared bus, as depicted in Figure 6.1. Dual- and quad-core configurations are quite popular nowadays and constitute the standard hardware configuration for commodity computers. Architectures with multiple cores are also available but are not designed for the commodity market. Multicore technology has been used not only as a support for processor design but also in other devices, such as GPUs and network devices, thus becoming a standard practice for improving performance.

[1] *Frequency scaling* refers to the practice of increasing the clock frequency of a processor to improve its performance. The increase of clock frequency leads to higher power consumption and a higher temperature on the die, which becomes unsustainable over certain values of the frequency clock. Also known as *frequency ramping*, this was the dominant technique for achieving performance gain from the mid-1980s to the end of 2004.

[2] www.nytimes.com/2004/05/08/business/08chip.html?ex = 1399348800&en = 98cc44ca97b1a562&ei = 5007.

[3] *Instruction-level parallelism* (ILP) is a measure of how many operations a computer program can perform at one time. There are several techniques that can be applied to increase the ILP at the microarchitectural level. One of these is *instruction pipelining*, which involves the division of instructions into stages so that a single processing unit can execute multiple instructions at the same time by carrying out different stages for each of them.

Multiprocessing is just one technique that can be used to achieve parallelism, and it does that by leveraging parallel hardware architectures. Parallel architectures are better exploited when programs are designed to take advantage of their features. In particular, an important role is played by the operating system, which defines the runtime structure of applications by means of the abstraction of *process* and *thread*. A process is the runtime image of an application, or better, a program that is running, while a thread identifies a single flow of the execution within a process. A system that allows the execution of multiple processes at the same time supports *multitasking*. It supports *multithreading* when it provides structures for explicitly defining multiple threads within a process.

Note that both multitasking and multithreading can be implemented on top of computer hardware that is constituted of a single processor and a single core, as was the common practice before the introduction of multicore technology. In this case, the operating system gives the illusion of concurrent execution by interleaving the execution of instructions of different processes and of different threads within the same process. This is also the case in multiprocessor/multicore systems, since the number of threads or processes is higher than the number of processors or cores. Nowadays, almost all the commonly used operating systems support multitasking and multithreading. Moreover, all the mainstream programming languages incorporate the abstractions of process and thread within their APIs, whereas direct support of multiple processors and cores for developers is very limited and often reduced and confined to specific libraries, which are available for a subset of the programming languages such as C/C++.

In this chapter, we concentrate our attention on multithreaded programming, which now has full support and constitutes the simplest way to achieve parallelism within a single process, despite the underlying hardware architecture.

6.2 Programming applications with threads

Modern applications perform multiple operations at the same time. Developers organize programs in terms of threads in order to express intraprocess concurrency. The use of threads might be implicit or explicit. *Implicit threading* happens when the underlying APIs use internal threads to perform specific tasks supporting the execution of applications such as graphical user interface (GUI) rendering, or garbage collection in the case of virtual machine-based languages. *Explicit threading* is characterized by the use of threads within a program by application developers, who use this abstraction to introduce parallelism. Common cases in which threads are explicitly used are I/O from devices and network connections, long computations, or the execution of background operations for which the outcome does not have specific time bounds. The use of threads was initially directed to allowing asynchronous operations—in particular, providing facilities for asynchronous I/O or long computations so that the user interface of applications did not block or became unresponsive. With the advent of parallel architectures the use of multithreading has become a useful technique to increase the throughput of the system and a viable option for throughput computing. To this purpose, the use of threads strongly impacts the design of algorithms that need to be refactored in order to leverage threads. In this section, we discuss the use of threading as a support for the design of parallel and distributed algorithms.

6.2.1 What is a thread?

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

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

In a multitasking environment the operating system assigns different time slices to each process and interleaves their execution. The process of temporarily stopping the execution of one process, saving all the information in the registers (and in general the state of the CPU in order to restore it later), and replacing it with the information related to another process is known as a *context switch*. This operation is generally considered demanding, and the use of multithreading minimizes the latency imposed by context switches, thus allowing the execution of multiple tasks in a lighter fashion. The state representing the execution of a thread is minimal compared to the one describing a process. Therefore, switching between threads is a preferred practice over switching between processes. Obviously the use of multiple threads in place of multiple processes is justified if and only if the tasks implemented are logically related to each other and require sharing memory or other resources. If this is not the case, a better design is provided by separating them into different processes.

Figure 6.2 provides an overview of the relation between threads and processes and a simplified representation of the runtime execution of a multithreaded application. A running program is identified by a process, which contains at least one thread, also called the *main thread*. Such a thread is implicitly created by the compiler or the runtime environment executing the program. This thread is likely to last for the entire lifetime of the process and be the origin of other threads, which in general exhibit a shorter duration. As main threads, these threads can spawn other threads. There is no difference between the main thread and other threads created during the process lifetime. Each of them has its own local storage and a sequence of instructions to execute, and they all share the memory space allocated for the entire process. The execution of the process is considered terminated when all the threads are completed.

6.2.2 Thread APIs

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

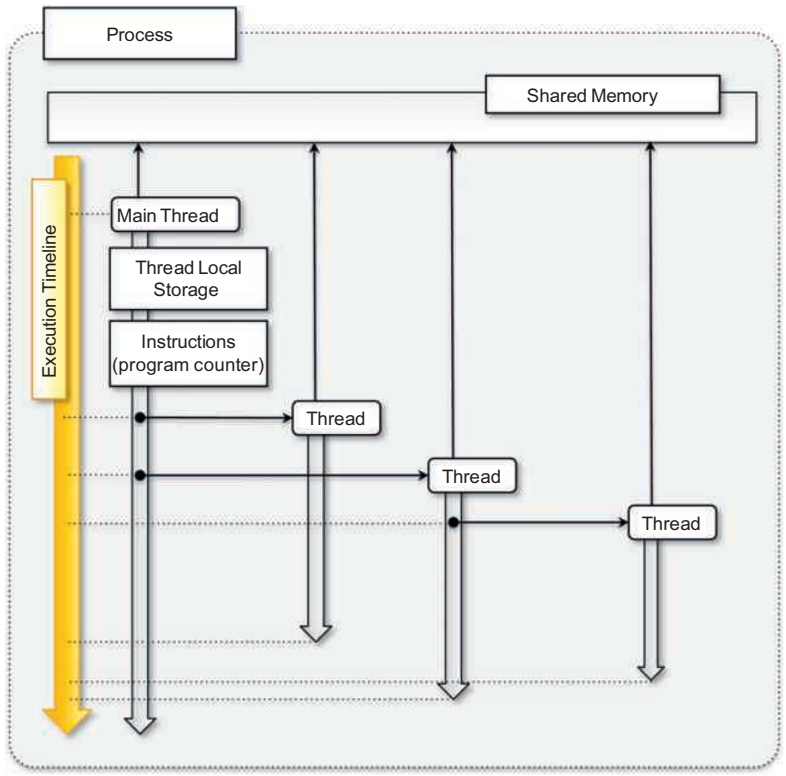


FIGURE 6.2

The relationship between processes and threads.

6.2.2.1 POSIX Threads

Portable Operating System Interface for Unix (POSIX) is a set of standards related to the application programming interfaces for a portable development of applications over the Unix operating system flavors. Standard POSIX 1.c (IEEE Std 1003.1c-1995) addresses the implementation of threads and the functionalities that should be available for application programmers to develop portable multithreaded applications. The standards address the Unix-based operating systems, but an implementation of the same specification has been provided for Windows-based systems.

The POSIX standard defines the following operations: creation of threads with attributes, termination of a thread, and waiting for thread completion (join operation). In addition to the logical structure of a thread, other abstractions, such as semaphores, conditions, reader-writer locks, and others, are introduced in order to support proper synchronization among threads.

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

- A thread identifies a logical sequence of instructions.
- A thread is mapped to a function that contains the sequence of instructions to execute.

- A thread can be created, terminated, or joined.
- A thread has a state that determines its current condition, whether it is executing, stopped, terminated, waiting for I/O, etc.
- The sequence of states that the thread undergoes is partly determined by the operating system scheduler and partly by the application developers.
- Threads share the memory of the process, and since they are executed concurrently, they need synchronization structures.
- Different synchronization abstractions are provided to solve different synchronization problems.

A default implementation of the POSIX 1.c specification has been provided for the C language. All the available functions and data structures are exposed in the *pthread.h* header file, which is part of the standard C implementations.

6.2.2.2 Threading support in java and .NET

Languages such as Java and C# provide a rich set of functionalities for multithreaded programming by using an object-oriented approach. Since both Java and .NET execute code on top of a virtual machine, the APIs exposed by the libraries refer to managed or logical threads. These are mapped to physical threads (i.e., those made available as abstractions by the underlying operating system) by the runtime environment in which programs developed with these languages execute. Despite such a mapping process, managed threads are considered, from a programming point of view, as physical threads and expose the same functionalities.

Both Java and .NET express the thread abstraction with the class *Thread* exposing the common operations performed on threads: *start*, *stop*, *suspend*, *resume*, *abort*, *sleep*, *join*, and *interrupt*. *Start* and *stop/abort* are used to control the lifetime of the thread instance, while *suspend* and *resume* are used to programmatically pause and then continue the execution of a thread. These two operations are generally deprecated in both of the two implementations that favor the use of appropriate techniques involving proper locks of the use of the *sleep* operation. This operation allows pausing the execution of a thread for a predefined period of time. This one is different from the *join* operation that makes one thread wait until another thread is completed. These waiting states can be interrupted by using the *interrupt* operation, which resumes the execution of the thread and generates an exception within the code of the thread to notify the abnormal resumption.

The two frameworks provide different support for implementing synchronization among threads. In general the basic features for implementing mutexes, critical regions, and reader-writer locks are completely covered by means of the basic class libraries or additional libraries. More advanced constructs than the thread abstraction are available in both languages. In the case of Java, most of them are contained in the *java.util.concurrent*[4] package, whereas the rich set of APIs for concurrent programming in .NET is further extended by the *.NET Parallel Extension* framework.[5]

[4] http://download.oracle.com/javase/6/docs/api/java/util/concurrent/package-summary.html.
[5] http://msdn.microsoft.com/en-us/concurrency/default.aspx.

6.2.3 Techniques for parallel computation with threads

Developing parallel applications requires an understanding of the problem and its logical structure. Understanding the dependencies and the correlation of tasks within an application is fundamental to designing the right program structure and to introducing parallelism where appropriate. *Decomposition* is a useful technique that aids in understanding whether a problem is divided into components (or tasks) that can be executed concurrently. If such decomposition is possible, it also provides a starting point for a parallel implementation, since it allows the breaking down into independent units of work that can be executed concurrently with the support provided by threads. The two main decomposition/partitioning techniques are *domain* and *functional* decompositions.

6.2.3.1 Domain decomposition

Domain decomposition is the process of identifying patterns of *functionally repetitive, but independent, computation on data*. This is the most common type of decomposition in the case of throughput computing, and it relates to the identification of repetitive calculations required for solving a problem.

When these calculations are identical, only differ from the data they operate on, and can be executed in any order, the problem is said to be *embarrassingly parallel* [59]. Embarrassingly parallel problems constitute the easiest case for parallelization because there is no need to synchronize different threads that do not share any data. Moreover, coordination and communication between threads are minimal; this strongly simplifies the code logic and allows a high computing throughput.

In many cases it is possible to devise a general structure for solving such problems and, in general, problems that can be parallelized through domain decomposition. The master-slave model is a quite common organization for these scenarios:

- The system is divided into two major code segments.
- One code segment contains the decomposition and coordination logic.
- Another code segment contains the repetitive computation to perform.
- A master thread executes the first code segment.
- As a result of the master thread execution, as many slave threads as needed are created to execute the repetitive computation.
- The collection of the results from each of the slave threads and an eventual composition of the final result are performed by the master thread.

Although the complexity of the repetitive computation strictly depends on the nature of the problem, the coordination and decomposition logic is often quite simple and involves identifying the appropriate number of units of work to create. In general, a *while* or a *for* loop is used to express the decomposition logic, and each iteration generates a new unit of work to be assigned to a slave thread. An optimization, of this process involves the use of thread pooling to limit the number of threads used to execute repetitive computations.

Several practical problems fall into this category; in the case of embarrassingly parallel problems, we can mention:

- Geometrical transformation of two (or higher) dimensional data sets
- Independent and repetitive computations over a domain such as Mandelbrot set and Monte Carlo computations

Even though embarrassingly parallel problems are quite common, they are based on the strong assumption that at each of the iterations of the decomposition method, it is possible to isolate an independent unit of work. This is what makes it possible to obtain a high computing throughput. Such a condition is not met if the values of all the iterations are dependent on some of the values obtained in the previous iterations. In this case, the problem is said to be *inherently sequential*, and it is not possible to directly apply the methodology described previously. Despite this, it can still be possible to break down the whole computation into a set of independent units of work, which might have a different granularity—for example, by grouping into single computation-dependent iterations. Figure 6.3 provides a schematic representation of the decomposition of embarrassingly parallel and inherently sequential problems.

To show how domain decomposition can be applied, it is possible to create a simple program that performs matrix multiplication using multiple threads.

Matrix multiplication is a binary operation that takes two matrices and produces another matrix as a result. This is obtained as a result of the composition of the linear transformation of the original matrices. There are several techniques for performing matrix multiplication; among them, the *matrix product* is the most popular. Figure 6.4 provides an overview of how a matrix product can be performed.

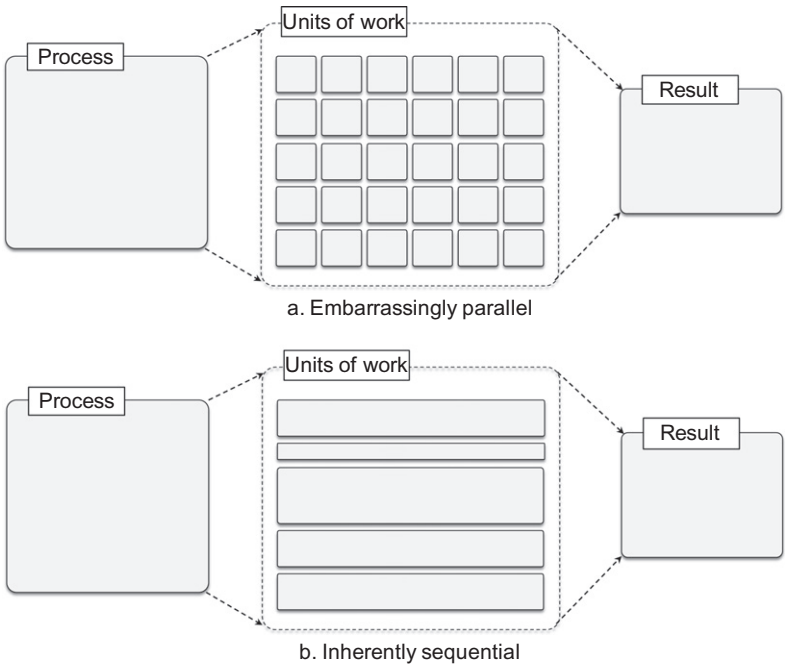


FIGURE 6.3

Domain decomposition techniques.

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

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

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

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

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

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

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

In order to give a practical example of the implementation of such a solution, we demonstrate the use of .NET threading. The .NET framework provides the *System.Threading.Thread* class that

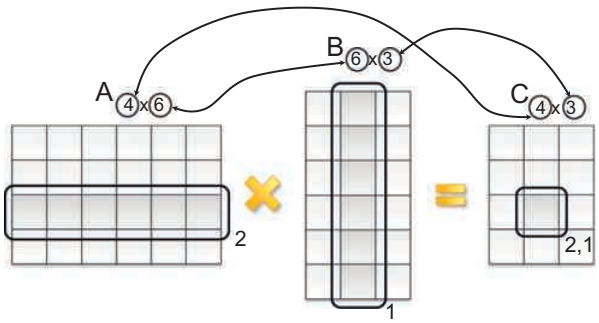


FIGURE 6.4

A matrix product.

can be configured with a function pointer, also known as a *delegate*, to execute asynchronously. Such a delegate must reference a defined method in some class. Hence, we can define a simple class that exposes as properties the row and the column to multiply and the result value. This class will also define the method for performing the actual computation. Listing 6.1 shows the class *ScalarProduct*.

The creation of the main thread of control is very simple. In this case, we skip the boilerplate code that is required to read the matrices from the standard input or from a file and concentrate our attention on the main control logic that decomposes the computation, creates threads, and waits for their completion in order to compose the resulting matrix.

To control the threads, we need to keep track of them so that we can query their status and obtain the result once they have completed the computation. We can create a simple program that reads the matrices, keeps track of all the threads in an appropriate data structure, and, once the threads have been completed, composes the final result. Listing 6.2 shows the content of the *MatrixProduct* class with some omissions.

Whereas the domain decomposition is quite simple, note that most of the complexity of the program resides in the management of threads. A few issues arise from the previous implementation:

- *Matrix layout.* Because of the way in which multidimensional arrays are stored, retrieving the column for the scalar product is not as straightforward as obtaining the row. This problem can be easily solved by memorizing the second matrix as *columns* × *rows* rather than *rows* × *columns*.
- *Result composition.* The composition of results is made on the master thread, and this requires keeping track of all the worker threads. Maintaining a reference to all the worker threads is in general a good programming practice, since it is necessary to terminate all of them before the application completes; but in this case it is possible to modify the application by using synchronization constructs that allow updating the resulting matrix from the worker threads. The new design implies storing the information about the indexes of rows and columns and a reference to the resulting matrix in the *ScalarProduct* class. As a result, there is no need to maintain a dictionary for threads, and we do not need the *ComposeResult* method in the master thread.

The example of a matrix product has been taken as a model to sketch the basic logic that is required to implement domain decomposition for an embarrassingly parallel problem and how to use threads in .NET to achieve throughput computing. This example can be taken as a reference to develop more sophisticated applications.

6.2.3.2 Functional decomposition

Functional decomposition is the process of identifying *functionally distinct but independent computations*. The focus here is on the type of computation rather than on the data manipulated by the computation. This kind of decomposition is less common and does not lead to the creation of a large number of threads, since the different computations that are performed by a single program are limited.

Functional decomposition leads to a natural decomposition of the problem in separate units of work because it does not involve partitioning the dataset, but the separation among them is clearly

```
///<summary>
/// Class ScalarProduct. Computes the scalar product between the row and the column
/// arrays.
///</summary>
public class ScalarProduct
{
   /// <summary>
   /// Scalar product.
   /// </summary>
   private double result;
   /// <summary>
   /// Gets the resulting scalar product.
   /// </summary>
   public double Result{ get { returnthis.result; } }

   /// <summary>
   /// Arrays containing the elements of the row and the column to multiply.
   /// </summary>
   private double[] row, column;

   /// <summary>
   /// Creates an instance of the ScalarProduct class and configures it with the given
   /// row and column arrays.
   /// </summary>
   /// <param name="row">Array with the elements of the row to be multiplied.</param>
   /// <param name="column">Array with the elements of the column to be multiplied.
   /// </param>
   public ScalarProduct(double[] row, double[] column)
   {
      this.row = row;
      this.colum = colum;
   }
   /// <summary>
   /// Executes the scalar product between the row and the colum.
   /// </summary>
   /// <param name="row">Array with the elements of the row to be multiplied.</param>
   /// <param name="column">Array with the elements of the column to be multiplied.
   /// </param>
   public void Multiply()
   {
      this.result = 0;
      for(int i=0; i<this.row.Length; i++)
      {
         this.result += this.row[i] * this.column[i];
      }
   }
}
```

LISTING 6.1

ScalarProduct Class.

```
using System;
using System.Threading;
using System.Collections.Generic;

///<summary>
/// Class MatrixProduct. Performs the matrix product of two matrices.
///</summary>
public class MatrixProduct
{
   ///<summary>
   /// First and second matrix of the produt.
   ///</summary>
   private static double[,]a, b;
   ///<summary>
   /// Result matrix.
   ///</summary>
   private static double[,] c;
   ///<summary>
   /// Dictionary mapping the thread instances to the corresponding ScalarProduct
   /// instances that are run inside.
   ///</summary>
   private static IDictionary<Thread, ScalarProduct>workers.

   ///<summary>
   /// Read the command line parameters and perform the scalar product.
   ///</summary>
   ///<param name="args">Array strings containing the command line parameters.</param>
   public static void Main(string[] args)
   {
       // reads the input matrices a and b.
       MatrixProduct.ReadMatrices();
       // executes the parallel matrix product.
       MatrixProduct.ExecuteProudct();
       // waits for all the threads to complete and
       // composes the final matrix.
       MatrixProduct.ComposeResult();
   }

   ///<summary>
   /// Executes the parallel matrix product by decomposing the problem in
   /// independent scalar product between rows and colums.
   ///</summary>
   private static void ExecuteThreads()
   {
       MatrixProduct.workers = newList<Thread>();
       int rows = MatrixProduct.a.Length;
       // in .NET matrices are arrays of arrays and the number of columns is
       // is represented by the length of the second array.
       int columns = MatrixProduct.b[0].Length;
       for(int i=0; i<rows; i++)
          for(int j=0; j<columns; j++)
          {
             double[] row = MatrixProduct.a[i];
             // because matrices are stored as arrays of arrays in order to
             // to get the columns we need to traverse the array and copy the
             // the data to another array.
```

LISTING 6.2

MatrixProduct Class (Main Program).

```
                double[] column = new double[common];
                for(int k=0; k<common; k++)
                {
                    column[j] = MatrixProduct.b[j][i];
                }
                // creates a ScalarProduct instance with the previous rows and
                // columns and starts a thread executing the Multiply method.
                ScalarProduct scalar = newScalarProduct(row, column);
                Thread worker = newThread(newThreadStart(scalar.Multiply));
                worker.Name =string.Format("{0}.{1}",row,column);
                worker.Start();
                // adds the thread to the dictionary so that it can be
                // further retrieved.
                MatrixProduct.workers.Add(worker, scalar);
            }
        }

        ///<summary>
        /// Waits for the completion of all the threads and composes the final
        /// result matrix.
        ///</summary>
        private static void ComposeResult()
        {
            MatrixProduct.c = new double[rows,columns];
            foreach(KeyValuePair<Thread,ScalarProduct>pair in MatrixProduct.workers)
            {
                Thread worker = pair.Key;
                // we have saved the coordinates of each  scalar product in the name
                // of the thread now we get them back by parsing the name .
                string[] indices = string.Split(worker.Name, new char[] {'.'});
                int i = int.Parse(indices[0]);
                int j = int.Parse(indices[1]);
                // we wait for the thread to complete
                worker.Join();
                // we set the result computed at the given coordinates.
                MatrixProduct.c[i,j] = pair.Value.Result;
            }
            MatrixProduct.PrintMatrix(MatrixProduct.c);
        }
        ///<summary>
        /// Reads the matrices.
        ///</summary>
        private static void ReadMatrices()
        {
            // code for reading the matrices a and b
        }
        ///<summary>
        /// Prints the given matrix.
        ///</summary>
        ///<param name="matrix">Matrix to print.</param>
        private static void PrintMatrices(double[,] matrix)
        {
            // code for printing the matrix.
        }
    }
```

LISTING 6.2

(Continued)

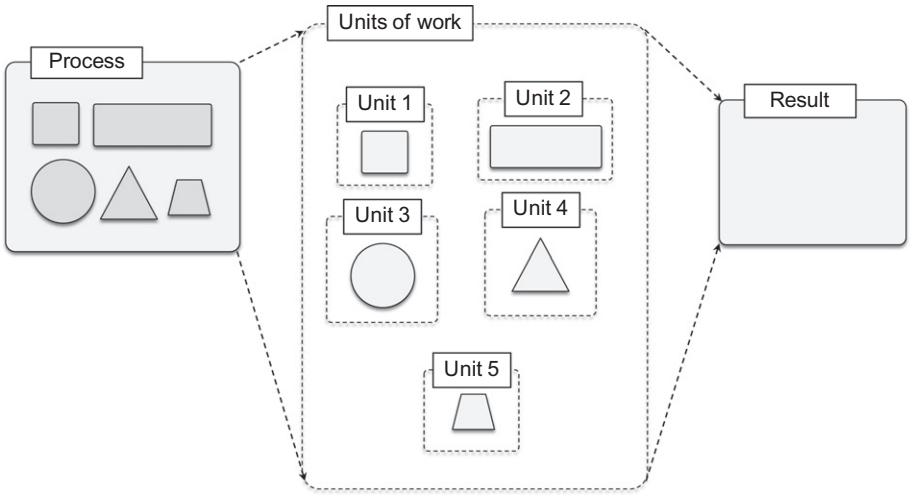


FIGURE 6.5

Functional decomposition.

defined by distinct logic operations. Figure 6.5 provides a pictorial view of how decomposition operates and allows parallelization.

As described by the schematic in Figure 6.5, problems that are subject to functional decomposition can also require a composition phase in which the outcomes of each of the independent units of work are composed together. In the case of domain decomposition, this phase often results in an aggregation process. The way in which results are composed in this case strongly depends on the type of operations that define the problem.

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

$$f(x) = \sin(x) + \cos(x) + \tan(x)$$

It is apparent that, once the value of x has been set, the three different operations can be performed independently of each other. This is an example of functional decomposition because the entire problem can be separated into three distinct operations. A possible implementation of a parallel version of the computation is shown in Listing 6.3.

The program computes the *sine*, *cosine*, and *tangent* functions in three separate threads and then aggregates the results. The implementation provided constitutes an example of the alternative technique discussed in the previous sample program. Instead of using a data structure for keeping track of the worker threads that have been created, a function pointer is passed to each thread so that it can update the final result at the end of the computation. This technique introduces a synchronization problem that is properly handled with the *lock* statement in the method referenced by the function pointer. The *lock* statement creates a critical section that can only be accessed by one thread at time and guarantees that the final result is properly updated.

```
using System;
using System.Threading;
using System.Collections.Generic;

/// <summary>
/// Delegate UpdateResult. Function pointer that is used to update the final result
/// from the slave threads once the computation is completed.
/// </summary>
/// <param name="x">partial value to add.</param>
public delegate void UpdateResult(double x);

/// <summary>
/// Class Sine. Computes the sine of a given value.
/// </summary>
public class Sine
{
   /// <summary>
   /// Input value for which the sine function is computed.
   /// </summary>
   private double x;
   /// <summary>
   /// Gets the input value of the sine function.
   /// </summary>
   public double X { get { return this.x; } }
   /// <summary>
   /// Result value.
   /// </summary>
   private double y;
   /// <summary>
   /// Gets the result value of the sine function.
   /// </summary>
   public double Y { get { return this.y; } }
   /// <summary>
   /// Function pointer used to update the result.
   /// </summary>
   private UpdateResult updater;
   /// <summary>
   /// Creates an instance of the Sine and sets the input to the given angle.
   /// </summary>
   /// <param name="x">Angle in radiants.</param>
   /// <param name="updater">Function pointer used to update the result.</param>
   public Sine(double x, UpdateResultupdater)
   {
      this.x = x;
      this.updater = updater;
   }
   /// <summary>
   /// Executes the sine function.
   /// </summary>
   public void Apply()
   {
      this.y = Math.Sin(this.x);
      if (this.updater != null)
      {
         this.updater(this.y);
```

LISTING 6.3

Mathematical Function.

```
        }
    }
}

///<summary>
/// Class Cosine. Computes the cosine of a given value.
///</summary>
public class Cosine
{
    /// <summary>
    /// Input value for which the cosine function is computed.
    /// </summary>
    private double x;
    /// <summary>
    /// Gets the input value of the cosine function.
    /// </summary>
    public double X { get { return this.x; } }
    /// <summary>
    /// Result value.
    /// </summary>
    private double y;
    /// <summary>
    /// Gets the result value of the cosine function.
    /// </summary>
    public double Y { get { return this.y; } }
    /// <summary>
    /// Function pointer used to update the result.
    /// </summary>
    private UpdateResultupdater;
    /// <summary>
    /// Creates an instance of the Cosine and sets the input to the given angle.
    /// </summary>
    /// <param name="x">Angle in radiants.</param>
    /// <param name="updater">Function pointer used to update the result.</param>
    public Cosine(double x, UpdateResultupdater)
    {
        this.x = x;
        this.updater = updater;
    }
    /// <summary>
    /// Executes the cosine function.
    /// </summary>
    public void Apply()
    {
        this.y = Math.Cos(this.x);
        if (this.updater != null)
        {
            this.updater(this.y);
        }
    }
}

///<summary>
/// Class Tangent. Computes the tangent of a given value.
///</summary>
public class Tangent
{
```

LISTING 6.3

(Continued)

```
    /// <summary>
    /// Input value for which the tangent function is computed.
    /// </summary>
    private double x;
    /// <summary>
    /// Gets the input value of the tangent function.
    /// </summary>
    public double X { get { return this.x; } }
    /// <summary>
    /// Result value.
    /// </summary>
    private double y;
    /// <summary>
    /// Gets the result value of the tangent function.
    /// </summary>
    public double Y { get { return this.y; } }
    /// <summary>
    /// Function pointer used to update the result.
    /// </summary>
    private UpdateResultupdater;
    /// <summary>
    /// Creates an instance of the Tangent and sets the input to the given angle.
    /// </summary>
    /// <param name="x">Angle in radiants.</param>
    /// <param name="updater">Function pointer used to update the result.</param>
    public Tangent(double x, UpdateResultupdater)
    {
        this.x = x;
        this.updater = updater;
    }
    /// <summary>
    /// Executes the cosine function.
    /// </summary>
    public void Apply()
    {
        this.y = Math.Tan(this.x);
        if (this.updater != null)
        {
            this.updater(this.y);
        }
    }
}
/// <summary>
/// Class Program. Computes the function sin(x) + cos(x) + tan(x).
/// </summary>
public class Program
{
    /// <summary>
    /// Variable storing the coputed value for the function.
    /// </summary>
    private static double result;
    /// <summary>
    /// Synchronization instance used to avoid keeping track of the threads.
    /// </summary>
    private static object synchRoot = new object();
    /// <summary>
    /// Read the command line parameters and perform the scalar product.
```

LISTING 6.3

(Continued)

```
        /// </summary>
        /// <param name="args">Array strings containing the command line parameters.</param>
        public static void Main(string[] args)
        {
            // gets a value for x
            double x = 3.4d;

            // creates the function pointer to the update method.
            UpdateResult updater = newUpdateResult(Program.Sum);

            // creates the sine thread.
            Sine sine = newSine(x, updater);
            Thread tSine =new Thread(new ThreadStart(sine.Apply);

            // creates the cosine thread.
            Cosine cosine = newCosine(x, updater);
            Thread tCosine =new Thread(new ThreadStart(cosine.Apply);

            // creates the tangent thread.
            Tangent tangent = newTangent(x, updater);
            Thread tTangent =new Thread(new ThreadStart(tangent.Apply);

            // shuffles the execution order.
            tTangent.Start();
            tSine.Start();
            tCosine.Start();

            // waits for the completion of the threads.
            tCosine.Join();
            tTangent.Join();
            tSine.Join();

            // the result is available, dumps it to console.
            Console.WriteLine("f({0}): {1}", x, Program.result);
        }
        /// <summary>
        /// Callback that is executed once the computation in the thread is completed
        /// and adds the partial value passed as a parameter to the result.
        /// </summary>
        /// <param name="partial">Partial value to add.</param>
        private static voidSum(double partial)
        {
           lock(Program.synchRoot)
           {
              Program.result += partial;
           }
        }
    }
```

LISTING 6.3

(Continued)

6.2.3.3 Computation vs. communication

In designing parallel and in general distributed applications, it is very important to carefully evaluate the communication patterns among the components that have been identified during problem decomposition. The two decomposition methods presented in this section and the corresponding sample applications are based on the assumption that the computations are *independent*. This means that:

- The input values required by one computation do not depend on the output values generated by another computation.
- The different units of work generated as a result of the decomposition do not need to interact (i.e., exchange data) with each other.

These two assumptions strongly simplify the implementation and allow achieving a high degree of parallelism and a high throughput. Having all the worker threads independent from each other gives the maximum freedom to the operating system (or the virtual runtime environment) scheduler in scheduling all the threads. The need to exchange data among different threads introduces dependencies among them and ultimately can result in introducing performance bottlenecks. For example, we did not introduce any queuing technique for threads; but queuing threads might potentially constitute a problem for the execution of the application if data need to be exchanged with some threads that are still in the queue. A more common disadvantage is the fact that while a thread exchanges data with another one, it uses some kind of synchronization strategy that might lead to blocking the execution of other threads. The more data that need to be exchanged, the more they block threads for synchronization, thus ultimately impacting the overall throughput.

As a general rule of thumb, it is important to minimize the amount of data that needs to be exchanged while implementing parallel and distributed applications. The lack of communication among different threads constitutes the condition leading to the highest throughput.

6.3 Multithreading with Aneka

As applications become increasingly complex, there is greater demand for computational power that can be delivered by a single multicore machine. Often this demand cannot be addressed with the computing capacity of a single machine. It is then necessary to leverage distributed infrastructures such as clouds. Decomposition techniques can be applied to partition a given application into several units of work that, rather than being executed as threads on a single node, can be submitted for execution by leveraging clouds.

Even though a distributed facility can dramatically increase the degree of parallelism of applications, its use comes with a cost in term of application design and performance. For example, since the different units of work are not executing within the same process space but on different nodes, both the code and the data need to be moved to a different execution context; the same happens for results that need to be collected remotely and brought back to the master process. Moreover, if there is any communication among the different workers, it is necessary to redesign the communication model eventually by leveraging the APIs, if any, provided by the middleware. In other words, the transition from a single-process multithreaded execution to a distributed execution is not transparent, and application redesign and reimplementation are often required.

The amount of effort required to convert an application often depends on the facilities offered by the middleware managing the distributed infrastructure. Aneka, as middleware for managing clusters, grids, and clouds, provides developers with advanced capabilities for implementing distributed applications. In particular, it takes traditional thread programming a step further. It lets you write multithreaded applications the traditional way, with the added twist that each of these threads can now be executed outside the parent process and on a separate machine. In reality, these "threads" are independent processes executing on different nodes and do not share memory or other resources, but they allow you to write applications using the same thread constructs for concurrency and synchronization as with traditional threads. Aneka threads, as they are called, let you easily port existing multithreaded compute-intensive applications to distributed versions that can run faster by utilizing multiple machines simultaneously, with minimum conversion effort.

6.3.1 Introducing the thread programming model

Aneka offers the capability of implementing multithreaded applications over the cloud by means of the *Thread Programming Model*. This model introduces the abstraction of distributed thread, also called *Aneka thread*, which mimics the behavior of local threads but executes over a distributed infrastructure. The Thread Programming Model has been designed to transparently port high-throughput multithreaded parallel applications over a distributed infrastructure and provides the best advantage in the case of embarrassingly parallel applications.

As described in Section 5.4.1, each application designed for Aneka is represented by a local object that interfaces to the middleware. According to the various programming models supported by the framework, such an interface exposes different capabilities, which are tailored to efficiently support the design and the implementation of applications by following a specific programming style. In the case of the Thread Programming Model, the application is designed as a collection of threads, the collective execution of which represents the application run. Threads are created and controlled by the application developer, while Aneka is in charge of scheduling their execution once they have been started. Threads are transparently moved and remotely executed while developers control them from local objects that act like proxies of the remote threads. This approach makes the transition from local multithreaded applications to distributed applications quite easy and seamless.

The Thread Programming Model exhibits APIs that mimic the ones exposed by .NET base class libraries for threading. In this way developers do not have to completely rewrite applications in order to leverage Aneka; the process of porting local multithreaded applications is as simple as replacing the *System.Threading.Thread* class and introducing the *AnekaApplication* class. There are three major elements that constitute the object model of applications based on the Thread Programming Model:

- *Application.* This class represents the interface to the Aneka middleware and constitutes a local view of a distributed application. In the Thread Programming Model the single units of work are created by the programmer. Therefore, the specific class used will be *Aneka.Entity.AnekaApplication* $< T,M >$, with T and M properly selected.
- *Threads.* Threads represent the main abstractions of the model and constitute the building blocks of the distributed application. Aneka provides the *Aneka.Threading.AnekaThread* class, which represents a distributed thread. This class exposes a subset of the methods exposed by the

System.Threading.Thread class, which has been reduced to those operations and properties that make sense or can be efficiently implemented in a distributed context.

- *Thread Manager*. This is an internal component that is used to keep track of the execution of distributed threads and provide feedback to the application. Aneka provides a specific version of the manager for this model, which is implemented in the *Aneka.Threading.ThreadManager* class.

As a result, porting local multithreaded applications to Aneka involves defining an instance of the *AnekaApplication*<*AnekaThread, ThreadManager*> class and replacing any occurrence of *System.Threading.Thread* with *Aneka.Threading.AnekaThread*. Developers can start creating threads, control their life cycles, and coordinate their execution similarly to local threads.

Aneka applications expose additional other properties, such as events that notify the completion of threads, their failure, the completion of the entire application, and thread state transitions. These operations are also available for the Thread Programming Model and constitute additional features that can be leveraged while porting local multithreaded applications, where this support needs to be explicitly programmed. Also, the *AnekaApplication* class provides support for files, which are automatically and transparently moved in the distributed environment.

6.3.2 Aneka thread vs. common threads

To efficiently run on a distributed infrastructure, Aneka threads have certain limitations compared to local threads. These limitations relate to the communication and synchronization strategies that are normally used in multithreaded applications.

6.3.2.1 Interface compatibility

The *Aneka.Threading.AnekaThread* class exposes almost the same interface as the *System.Threading.Thread* class with the exception of a few operations that are not supported. Table 6.1 compares the operations that are exposed by the two classes. The reference namespace that defines all the types referring to the support for threading is *Aneka.Threading* rather than *System.Threading*.

The basic control operations for local threads such as *Start* and *Abort* have a direct mapping, whereas operations that involve the temporary interruption of the thread execution have not been supported. The reasons for such a design decision are twofold. First, the use of the *Suspend/Resume* operations is generally a deprecated practice, even for local threads, since *Suspend* abruptly interrupts the execution state of the thread. Second, thread suspension in a distributed environment leads to an ineffective use of the infrastructure, where resources are shared among different tenants and applications. This is also the reason that the *Sleep* operation is not supported. Therefore, there is no need to support the *Interrupt* operation, which forcibly resumes the thread from a waiting or a sleeping state. To support synchronization among threads, a corresponding implementation of the *Join* operation has been provided.

Besides the basic thread control operations, the most relevant properties have been implemented, such as name, unique identifier, and state. Whereas the name can be freely assigned, the identifier is generated by Aneka, and it represents a globally unique identifier (GUID) in its string form rather than an integer. Properties such as *IsBackground*, *Priority*, and *IsThreadPoolThread* have been provided for interface compatibility but actually do not have any effect on thread scheduling and always expose the values reported in the table. Other properties concerning the state of the

Table 6.1 Thread API Comparison

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

thread, such as *IsAlive* and *IsRunning*, exhibit the expected behavior, whereas a slightly different behavior has been implemented for the *ThreadState* property that is mapped to the *State* property. The remaining methods of the *System.Threading.Thread* class (.NET 2.0) are not supported.

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

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

6.3.2.2 Thread life cycle

Since Aneka threads live and execute in a distributed environment, their life cycle is necessarily different from the life cycle of local threads. For this reason, it is not possible to directly map the state values of a local thread to those exposed by Aneka threads. Figure 6.6 provides a comparative view of the two life cycles.

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

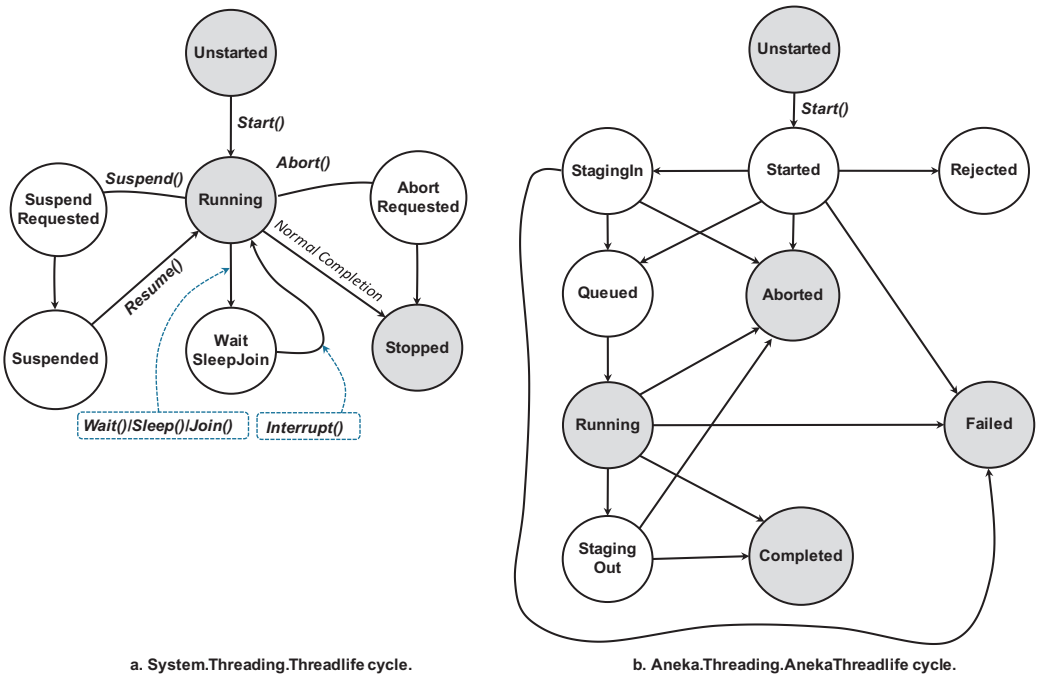


FIGURE 6.6

Thread life-cycle comparison.

by the middleware. As depicted in Figure 6.6, Aneka threads exhibit more states than local threads because Aneka threads support file staging and they are scheduled by the middleware, which can queue them for a considerable amount of time. As Aneka supports the reservation of nodes for execution of thread related to a specific application, an explicit state indicating execution failure due to missing reservation credential has been introduced. This occurs when a thread is sent to an execution node in a time window where only nodes with specific reservation credentials can be executed.

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

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

to *Completed*. At any point, if the developer stops the execution of the application or directly calls the *Abort()* method, the thread is aborted and its final state is set to *Aborted*.

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

6.3.2.3 Thread synchronization

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

6.3.2.4 Thread priorities

The *System.Threading.Thread* class supports thread priorities, where the scheduling priority can be one selected from one of the values of the *ThreadPriority* enumeration: *Highest*, *AboveNormal*, *Normal*, *BelowNormal*, or *Lowest*. However, operating systems are not required to honor the priority of a thread, and the current version of Aneka does not support thread priorities. For interface compatibility purposes the *Aneka.Threading.Thread* class exhibits a *Priority* property whose type is *ThreadPriority*, but its value is always set to *Normal*, and changes to it do not produce any effect on thread scheduling by the Aneka middleware.

6.3.2.5 Type serialization

Aneka threads execute in a distributed environment in which the object code in the form of libraries and live instances information are moved over the network. This condition imposes some limitations that are mostly concerned with the serialization of types in the .NET framework.

Local threads execute all within the same address space and share memory; therefore, they do not need objects to be copied or transferred into a different address space. Aneka threads are distributed and execute on remote computing nodes, and this implies that the object code related to the method to be executed within a thread needs to be transferred over the network. Since delegates can point to instance methods, the state of the enclosing instance needs to be transferred and reconstructed on the remote execution environment. This is a particular feature at the class level and goes by the term *type serialization*.

A .NET type is considered *serializable* if it is possible to convert an instance of the type into a binary array containing all the information required to revert it to its original form or into a possibly different execution context. This property is generally given for several types defined in the .NET framework by simply tagging the class definition with the *Serializable* attribute. If the class

exposes a specific set of characteristics, the framework will automatically provide facilities to serialize and deserialize instances of that type. Alternatively, custom serialization can be implemented for any user-defined type.

Aneka threads execute methods defined in serializable types, since it is necessary to move the enclosing instance to remote execution method. In most cases, providing serialization is as easy as tagging the class definition with the *Serializable* attribute; in other cases it might be necessary to implement the *ISerializable* interface and provide appropriate constructors for the type. This is not a strong limitation, since there are very few cases in which types cannot be defined as serializable. For example, local threads, network connections, and streams are not serializable since they directly access local resources that cannot be implicitly moved onto a different node.

6.4 Programming applications with Aneka threads

To show how it is possible to quickly port multithreaded application to Aneka threads, we provide a distributed implementation of the previously discussed examples for local threads.

6.4.1 Aneka threads application model

The *Thread Programming Model* is a programming model in which the programmer creates the units of work as Aneka threads. Therefore, it is necessary to utilize the *AnekaApplication* $<W,M>$ class, which is the application reference class for all the programming models falling into this category. The Aneka APIs make strong use of generics and characterize the support given to different programming models through template specialization. Hence, to develop distributed applications with Aneka threads, it is necessary to specialize the template type as follows:

AnekaApplication $<$ *AnekaThread, ThreadManager* $>$

This will be the class type for all the distributed applications that use the Thread Programming Model. These two types are defined in the *Aneka.Threading* namespace noted in the *Aneka.Threading.dll* library of the Aneka SDK.

Another important component of the application model is the *Configuration* class, which is defined in the *Aneka.Entity* namespace (*Aneka.dll*). This class contains a set of properties that allow the application class to configure its interaction with the middleware, such as the address of the Aneka index service, which constitutes the main entry point of Aneka Clouds; the user credentials required to authenticate the application with the middleware; some additional tuning parameters; and an extended set of properties that might be used to convey additional information to the middleware. The code excerpt presented in Listing 6.4 demonstrates how to create a simple application instance and configure it to connect to an Aneka Cloud whose index service is local.

Once the application has been created, it is possible to create threads by specifying the reference to the application and the method to execute in each thread, and the management of the application execution is mostly concerned with controlling the execution of each thread instance. Listing 6.5 provides a very simple example of how to create Aneka threads.

The rest of the operations relate to the common management of thread instances, similar to local multithreaded applications discussed earlier.

```
// namespaces containing types of common use
using System;
using System.Collections.Generic;
// common Aneka namespaces.
using Aneka;
using Aneka.Util;
using Aneka.Entity;
// Aneka Thread Programming Model user classes
using Aneka.Threading;

// ……

///<summary>
///Creates an instance of the Aneka Application configured to use the
/// Thread Programming Model.
/// </summary>
/// <returns>Application instance.</returns>
private AnekaApplication<AnekaThread,ThreadManager> CreateApplication();
{
   Configuration conf =new Configuration();
   // this is the common address and port of a local installation
   // of the Aneka Cloud.
   conf.SchedulerUri = newUri("tcp://localhost:9090/Aneka");
   conf.Credentials =newUserCredentials("Administrator", string.Empty);
   // we will not need support for file transfer, hence we optimize the
   // application in order to not require any file transfer service.
   conf.UseFileTransfer = false;
   // we do not need any other configuration setting

   // we create the application instance and configure it.
   AnekaApplication<AnekaThread,ThreadManager> app =
                  new AnekaApplication<AnekaThread,ThreadManager>(conf);
   return app;
}
```

LISTING 6.4

Application Creation and Configuration.

6.4.2 Domain decomposition: matrix multiplication

To port to Aneka threads the multithreaded matrix multiplication, we need to apply the considerations made in the previous section. Hence, we start reviewing the code by first making the proper changes to the *ScalarProduct* class. Listing 6.6 shows the modified version of *ScalarProduct*.

The class has been tagged with the *Serializable* attribute and extended with the methods required to implement custom serialization. Supporting custom serialization implies the following:

- Including the *System.Runtime.Serialization* namespace.
- Implementing the *ISerializable* interface. This interface has only one method that is void *GetObjectData(SerializationInfo, StreamingContext)*, and it is called when the runtime needs to serialize the instance.
- Providing a constructor with the following signature: *ScalarProduct(SerializationInfo, StreamingContext)*. This constructor is invoked when the instance is deserialized.

```
// ..... continues from the previous listing
///<summary>
///Thread worker method (implementation skipped).
///</summary>
private void WorkerMethod()
{
   // .....
}
///<summary>
///Creates a collection of threads that are executed in the context of the
/// the given application.
///</summary>
/// <param name="app">>Application instance.</param>
private void CreateThreads(AnekaApplication<AnekaThread,ThreadManager> app);
{
    // creates a delegate to the method to execute inside the threads.
    ThreadStart worker = newThreadStart(this.WorkerMethod);
    // iterates over a loop and creates ten threads.
    for(int i=0; i<10; i++)
    {
        AnekaThread thread = new AnekaThread(worker, app);
        thread.Start();
    }
}
```

LISTING 6.5

Thread Creation and Execution.

The *SerializationInfo* class has a central role of providing a repository where all the properties defining the serialized format of a class can be stored and referenced by name. With minimum changes to the class layout, it would be possible to rely on the default serialization provided by the framework. To leverage such capability, it is necessary that all the properties defining the state of an instance are accessible through both *get* and *set* methods. In that case, it would be possible to simply tag the class as serialization, since all the fields constituting the state of the instance are also serializable. It can be noted that, apart from serialization, there is no need to make any change to the way the class operates.

The second step is to change the *MatrixProduct* class to leverage Aneka threads. We need to first create a properly configured application and then substitute the occurrences of the *System.Threading.Thread* class with *Aneka.Threading.Thread* (see Listing 6.7).

As shown in Listing 6.7, the changes that need to be applied to the logic of the program are minimal, and most of the modifications are related to exception management and the proper use of Aneka logging facilities. The *MatrixProduct* class integrates the method discussed in the previous section for application creation and setup and introduces a *try...catch...finally* block to handle exceptions that occurred while the application was executing. The rest of the code, except for renaming the occurrences of the *Thread* class, is unchanged.

There is only one important change to note: Once the Aneka thread instance is completed, the updated reference to the object containing the remotely executed method is exposed by the

```
using System.Runtime.Serialization;

///<summary>
/// Class ScalarProduct. Computes the scalar product between the row and the column
/// arrays. The class uses custom serialization. In order to do so it implements the
/// the ISerializable interface.
///</summary>
[Serializable]
public class ScalarProduct : ISerializable
{
   /// <summary>
   /// Scalar product.
   /// </summary>
   private double result;
   /// <summary>
   /// Gets the resulting scalar product.
   /// </summary>
   public double Result{get { returnthis.result; }}

   /// <summary>
   /// Arrays containing the elements of the row and the column to multiply.
   /// </summary>
   private double[] row, column;

   /// <summary>
   /// Creates an instance of the ScalarProduct class and configures it with the given
   /// row and column arrays.
   /// </summary>
   /// <param name="row">Array with the elements of the row to be multiplied.</param>
   /// <param name="column">Array with the elements of the column to be multiplied.
   /// </param>
   public ScalarProduct(double[] row, double[] column)
   {
      this.row = row;
      this.colum = colum;
   }
   /// <summary>
   /// Deserialization constructor used by the .NET runtime to recreate instances of
   /// of types implementing custom serialization.
   /// </summary>
   /// <param name="info">Bag containing the serialized data instance.</param>
   /// <param name="context">Serialization context (not used).</param>
   public ScalarProduct(SerializationInfo info, StreamingContext context)
   {
      this.result = info.GetDouble("result");
      this.row = info.GetValue("row", typeof(double[])) as double[];
      this.column = info.GetValue("column", typeof(double[])) as double[];
   }
```

LISTING 6.6

ScalarProduct Class (Modified Version).

```
    /// <summary>
    /// Executes the scalar product between the row and the colum.
    /// </summary>
    /// <param name="row">Array with the elements of the row to be multiplied.</param>
    /// <param name="column">Array with the elements of the column to be multiplied.
    /// </param>
    public void Multiply()
    {
       this.result = 0;
       for(int i=0; i<this.row.Length; i++)
       {
           this.result += this.row[i] * this.column[i];
       }
    }
    /// <summary>
    /// Serialization method used by the .NET runtime to serialize instances of
    /// of types implementing custom serialization.
    /// </summary>
    /// <param name="info">Bag containing the serialized data instance.</param>
    /// <param name="context">Serialization context (not used).</param>
    public ScalarProduct(SerializationInfo info, StreamingContext context)
    {
       this.result = info.GetDouble("result");
       this.row = info.GetValue("row", typeof(double[])) as double[];
       this.column = info.GetValue("column", typeof(double[])) as double[];
    }
    /// <summary>
    /// Executes the scalar product between the row and the colum.
    /// </summary>
    /// <param name="row">Array with the elements of the row to be multiplied.</param>
    /// <param name="column">Array with the elements of the column to be multiplied.
    /// </param>
    public void Multiply()
    {
       this.result = 0;
       for(int i=0; i<this.row.Length; i++)
       {
           this.result += this.row[i] * this.column[i];
       }
    }
    /// <summary>
    /// Serialization method used by the .NET runtime to serialize instances of
    /// of types implementing custom serialization.
    /// </summary>
    /// <param name="info">Bag containing the serialized data instance.</param>
    /// <param name="context">Serialization context (not used).</param>
    public void GetObjectData(SerializationInfo info, StreamingContext context)
    {
       info.AddValue("result",this.result);
       info.AddValue("row", this.row,typeof(double[]));
       info.AddValue("column", this.column,typeof(double[]));
    }
}
```

LISTING 6.6

(Continued)

```
using System;
// we do not anymore need the reference to the threading namespace.
// using System.Threading;
using System.Collections.Generic;

// reference to the Aneka namespaces of interest.
// common Aneka namespaces.
using Aneka;
using Aneka.Util;
using Aneka.Entity;
// Aneka Thread Programming Model user classes
using Aneka.Threading;

/// <summary>
/// Class MatrixProduct. Performs the matrix product of two matrices.
/// </summary>
public class MatrixProduct
{
   /// <summary>
   /// First and second matrix of the produt.
   /// </summary>
   private static double[,]a, b;
   /// <summary>
   /// Result matrix.
   /// </summary>
   private static double[,] c;
   ///<summary>
   /// Dictionary mapping the thread instances to the corresponding ScalarProduct
   ///instances that are run inside.The occurrence of the Thread class has been
   ///substituted with AnekaThread.
   ///</summary>
   private static IDictionary<AnekaThread, ScalarProduct> workers.
   /// <summary>
   /// Reference to the distributed application the threads belong to.
   /// </summary>
   private static AnekaApplication<AnekaThread, ThreadManager> app;

   /// <summary>
   /// Read the command line parameters and perform the scalar product.
   /// </summary>
   /// <param name="args">Array strings containing the command line parameters.</param>
   public static void Main(string[] args)
   {
      try
      {
         // activates the logging facility.
         Logger.Start();

         // creates the Aneka application instance.
         MatrixProduct.app =Program.CreateApplication();

         // reads the input matrices a and b.
         MatrixProduct.ReadMatrices();
         // executes the parallel matrix product.
         MatrixProduct.ExecuteProudct();
         // waits for all the threads to complete and
         // composes the final matrix.
```

LISTING 6.7

MatrixProduct Class (Modified Version).

```
            MatrixProduct.ComposeResult();
        }
        catch(Exception ex)
        {
            IOUtil.DumpErrorReport(ex, "Matrix Multiplication - Error executing " +
                                       "the application");
        }
        finally
        {
            try
            {
                // checks whether the application instance has been created
                // stops it.
                if (MatrixProduct.app != null)
                {
                    MatrixProduct.app.Stop();
                }
            }
            catch(Exception ex)
            {
                IOUtil.DumpErrorReport(ex, "Matrix Multiplication - Error stopping " +
                                           "the application");
            }
            // stops the logging thread.
            Logger.Stop();
        }
    }

    /// <summary>
    /// Executes the parallel matrix product by decomposing the problem in
    /// independent scalar product between rows and colums.
    /// </summary>
    private static void ExecuteThreads()
    {
        // we replace the Thread class with AnekaThread.
        MatrixProduct.workers = new Dictionary<AnekaThread, ScalarProduct>();
        int rows = MatrixProduct.a.Length;
        // in .NET matrices are arrays of arrays and the number of columns is
        // is represented by the length of the second array.
        int columns = MatrixProduct.b[0].Length;

        for(int i=0; i<rows; i++)
        for(int j=0; j<columns; j++)
        {
            double[] row = MatrixProduct.a[i];
            // beacause matrices are stored as arrays of arrays in order to
            // to get the columns we need to traverse the array and copy the
            // the data to another array.
            double[] column = new double[common];
            for(int k=0; k<common; k++)
            {
                column[j] = MatrixProduct.b[j][i];
            }
            // creates a ScalarProduct instance with the previous rows and
            // columns and starts a thread executing the Multiply method.
            ScalarProduct scalar = newScalarProduct(row, column);
            // we change the System.Threading.Thread class with the corresponding
```

LISTING 6.7

(Continued)

```
            // Aneka.Threading.AnekaThread class and reference the application instance.
            AnekaThread worker = newAnekaThread(newThreadStart(scalar.Multiply), app);
            worker.Name =string.Format("{0}.{1}",row,column);
            worker.Start();
            // adds the thread to the dictionary so that it can be
            // further retrieved.
            MatrixProduct.workers.Add(worker, scalar);
        }
    }
    /// <summary>
    /// Waits for the completion of all the threads and composes the final
    /// result matrix.
    /// </summary>
    private static void ComposeResult()
    {
        MatrixProduct.c = new double[rows,columns];
        // we replace the Thread class with AnekaThread.
        foreach(KeyValuePair<AnekaThread,ScalarProduct>pair in MatrixProduct.workers)
        {
            AnekaThread worker = pair.Key;
            // we have saved the coordinates of each scalar product in the name
            // of the thread now we get them back by parsing the name.
            string[] indices = string.Split(worker.Name, new char[] {'.'});
            int i = int.Parse(indices[0]);
            int j = int.Parse(indices[1]);
            // we wait for the thread to complete
            worker.Join();
            // instead of using the local value of the ScalarProduct instance
            // we use the one that has is stored in the Target property.
            // MatrixProduct.c[i,j] = pair.Value.Result;
            MatrixProduct.c[i,j] = ((ScalarProduct) worker.Target).Result;
        }
        MatrixProduct.PrintMatrix(MatrixProduct.c);
    }
    /// <summary>
    /// Reads the matrices.
    /// </summary>
    private static void ReadMatrices()
    {
        // code for reading the matrices a and b
    }
    /// <summary>
    /// Prints the given matrix.
    /// </summary>
    /// <param name="matrix">Matrix to print.</param>
    private static voidPrintMatrices(double[,] matrix)
    {
        // code for printing the matrix.
    }
    /// <summary>
    /// Creates an instance of the Aneka Application configured to use the
    /// Thread Programming Model.
    /// </summary>
    /// <returns>Application instance.</returns>
    private AnekaApplication<AnekaThread,ThreadManager> CreateApplication();
    {
```

LISTING 6.7

(Continued)

```
            Configuration conf =new Configuration();
            // this is the common address and port of a local installation
            // of the Aneka Cloud.
            conf.SchedulerUri = newUri("tcp://localhost:9090/Aneka");
            conf.Credentials =newUserCredentials("Administrator", string.Empty);
            // we will not need support for file transfer, hence we optimize the
            // application in order to not require any file transfer service.
            conf.UseFileTransfer = false;
            // we do not need any other configuration setting

            // we create the application instance and configure it.
            AnekaApplication<AnekaThread,ThreadManager> app =
                        new AnekaApplication<AnekaThread,ThreadManager>(conf);
            return app;
        }
    }
```

LISTING 6.7

(Continued)

AnekaThread.Target property and not the local variable referencing the object that was initially used to create the delegate.

6.4.3 Functional decomposition: *Sine*, *Cosine*, and *Tangent*

The modifications required to port this sample to Aneka threads are basically the same as those discussed in the previous example. There is only one significant difference in this case: Each of the threads has a reference to a delegate that is used to update the global sum at the end of the computation. Since we are operating in a distributed environment, the instance on which the object will operate is not shared among the threads, but each thread instance has its own local copy. This prevents the global sum from being updated in the master thread and requires a change in the update strategy utilized.

This example also illustrates how to modify the classes *Sine*, *Cosine*, and *Tangent* so that they can leverage the default serialization capabilities of the framework (see Listing 6.8).

This example demonstrated how to change the logic of the application in case the worker methods executed in the threads have a reference to a local object that is updated as a consequence of the execution. To allow the execution of such applications with Aneka threads, it is necessary to extrapolate the update logic from the worker method of the threads and perform it into the master thread.

SUMMARY

This chapter provided a brief overview of multithreaded programming and the technologies used for multiprocessing on a single machine. We introduced the basics of multicore technology, which is the latest technological advancement for achieving parallelism on a single computer, and discussed how such parallelism can be leveraged to speed up applications by using multithreaded programming. A thread defines a single control flow within a process, which is the logical unit for

```
using System;
// we do not anymore need the reference to the threading namespace.
// using System.Threading;
using System.Collections.Generic;

// reference to the Aneka namespaces of interest.
// common Aneka namespaces.
using Aneka;
using Aneka.Util;
using Aneka.Entity;
// Aneka Thread Programming Model user classes
using Aneka.Threading;


// this is not needed anymore.
// /// <summary>
// /// Delegate UpdateResult. Function pointer that is used to update the final result
// /// from the slave threads once the computation is completed.
// /// </summary>
// /// <param name="x">partial value to add.</param>
// public delegate void UpdateResult(double x);

/// <summary>
/// Class Sine. Computes the sine of a given value.
/// </summary>
[Serializable]
public class Sine
{
   /// <summary>
   /// Input value for which the sine function is computed.
   /// </summary>
   private double x;
   /// <summary>
   /// Gets or sets the input value of the sine function.
   /// </summary>
   public double X { get { return this.x; } set { this.x = value; } }
   /// <summary>
   /// Result value.
   /// </summary>
   private double y;
   /// <summary>
   /// Gets or sets the result value of the sine function.
   /// </summary>
   public double Y { get { return this.y; } set { this.y = value; } }
   // we can't use this anymore.
   // /// <summary>
   // /// Function pointer used to update the result.
   // /// </summary>
   // private UpdateResult updater;

   // we need a default constructor, which is automatically provided by the compiler
   // if we do not specify any constructor.
   // /// <summary>
   // /// Creates an instance of the Sine and sets the input to the given angle.
   // /// </summary>
   // /// <param name="x">Angle in radiants.</param>
   // /// <param name="updater">Function pointer used to update the result.</param>
```

LISTING 6.8

Mathematical Function (Modified Version).

```
    // public Sine(double x, UpdateResult updater)
    // {
    //    this.x = x;
    //     this.updater = updater;
    // }
    /// <summary>
    /// Executes the sine function.
    /// </summary>
    public void Apply()
    {
        this.y = Math.Sin(this.x);
        // we cannot use this anymore because there is no
        // shared memory space.
        // if (this.updater != null)
        // {
        //     this.updater(this.y);
        // }
    }
}

///<summary>
/// Class Cosine. Computes the cosine of a given value. The same changes have been
/// applied by removing the code not needed anymore rather than commenting it out.
///</summary>
[Serializable]
public class Cosine
{
    /// <summary>
    /// Input value for which the cosine function is computed.
    /// </summary>
    private double x;
    /// <summary>
    /// Gets or sets the input value of the cosine function.
    /// </summary>
    public double X { get { return this.x; } set { this.x = value; } }
    /// <summary>
    /// Result value.
    /// </summary>
    private double y;
    /// <summary>
    /// Gets or sets the result value of the cosine function.
    /// </summary>
    public double Y { get { return this.y; } set { this.y = value; } }
    /// <summary>
    /// Executes the cosine function.
    /// </summary>
    public voidApply()
    {
        this.y = Math.Cos(this.x);
    }
}

///<summary>
/// Class Cosine. Computes the cosine of a given value.  The same changes have been
/// applied by removing the code not needed anymore  rather than commenting it out.
///</summary>
[Serializable]
```

LISTING 6.8

(Continued)

```
public class Tangent
{
    /// <summary>
    /// Input value for which the tangent function is computed.
    /// </summary>
    private double x;
    /// <summary>
    /// Gets or sets the input value of the tangent function.
    /// </summary>
    public double X { get { return this.x; } set { this.x = value; } }
    /// <summary>
    /// Result value.
    /// </summary>
    private double y;
    /// <summary>
    /// Gets or sets the result value of the tangent function.
    /// </summary>
    public double Y { get { return this.y; } set { this.y = value; } }
    /// <summary>
    /// Executes the tangent function.
    /// </summary>
    public voidApply()
    {
        this.y = Math.Cos(this.x);
    }
}

///<summary>
/// Class Cosine. Computes the cosine of a given value. The same changes have been
/// applied by removing the code not needed anymore  rather than commenting it out.
///</summary>
[Serializable]
public class Tangent
{
    /// <summary>
    /// Input value for which the tangent function is computed.
    /// </summary>
    private double x;
    /// <summary>
    /// Gets or sets the input value of the tangent function.
    /// </summary>
    public double X { get { return this.x; } set { this.x = value; } }
    /// <summary>
    /// Result value.
    /// </summary>
    private double y;
    /// <summary>
    /// Gets or sets the result value of the tangent function.
    /// </summary>
    public double Y { get { return this.y; } set { this.y = value; } }
    /// <summary>
    /// Executes the tangent function.
    /// </summary>
    public voidApply()
    {
        this.y = Math.Tan(this.x);
    }
```

LISTING 6.8

(Continued)

```
}

/// <summary>
/// Class Program. Computes the function sin(x) + cos(x) + tan(x).
/// </summary>
public class Program
{
   /// <summary>
   /// Variable storing the coputed value for the function.
   /// </summary>
   private static double result;

   // we do not need synchronization anymore, because the update of the global
   // sum is done sequentially.
   // /// <summary>
   // /// Synchronization instance used to avoid keeping track of the threads.
   // /// </summary>
   // private static object synchRoot = new object();

   /// <summary>
   /// Reference to the distributed application the threads belong to .
   /// </summary>
   private static AnekaApplication<AnekaThread, ThreadManager> app;

   /// <summary>
   /// Read the command line parameters and perform the scalar product.
   /// </summary>
   /// <param name="args">Array strings containing the command line parameters. </param>
   public static voidMain(string[] args)
   {
      try
      {
         // activates the logging facility.
         Logger.Start();
         // creates the Aneka application instance.
            app = Program.CreateApplication();

         // gets a value for x
            double x = 3.4d;

            // creates the function pointer to the update method.
         UpdateResult updater = newUpdateResult(Program.Sum);

         // creates the sine thread.
            Sine sine = newSine(x, updater);
         AnekaThread tSine =newAnekaThread(new ThreadStart(sine.Apply), app);

         // creates the cosine thread.
         Cosine cosine = newCosine(x, updater);
         AnekaThread tCosine =newAnekaThread(new ThreadStart(cosine.Apply), app);

         // creates the tangent thread.
         Tangent tangent = newTangent(x, updater);
         AnekaThread tTangent =newAnekaThread(new ThreadStart(tangent.Apply), app);

         // shuffles the execution order.
            tTangent.Start();
```

LISTING 6.8

(Continued)

```
            tSine.Start();
            tCosine.Start();

            // waits for the completion of the  threads.
            tCosine.Join();
            tTangent.Join();
            tSine.Join();

            // once we have joined all the threads the values have been collected back
            // and we use the Target property in order to obtain the object with the
            // updated values.
            sine = (Sine) tSine.Target;
            cosine = (Cosine) tSine.Target;
            tangent = (Tangent) tSine.Target;

            Program.result = sine.Target.Y + cosine.Y + tangent.Y;

            // the result is available, dumps it to  console.
            Console.WriteLine("f({0}): {1}", x, Program.result);
        }
        catch(Exception ex)
        {
            IOUtil.DumpErrorReport(ex,  "Math Functions - Error executing " +
                                        "the application");
        }
        finally
        {
            try
            {
                // checks whether the application instance has been created
                // stops it.
                if (app != null)
                {
                    app.Stop();
                }
            }
            catch(Exception ex)
            {
                IOUtil.DumpErrorReport(ex,  "Math Functions - Error stopping " +
                                            "the application");
            }
            // stops the logging thread.
            Logger.Stop();
        }
    }
    // we do not need this anymore.
    // /// <summary>
    // /// Callback that is executed once the computation in the thread is completed
    // /// and adds the partial value passed as a parameter to the result.
    // /// </summary>
    // /// <param name="partial">Partial value to add.</param>
    // private static void Sum(double partial)
    // {
    //     lock(Program.synchRoot)
    //     {
    //         Program.result += partial;
    //     }
```

LISTING 6.8

(Continued)

```
        // }
        /// <summary>
        /// Creates an instance of the Aneka Application configured to use the
        /// Thread Programming Model.
        /// </summary>
        /// <returns>Application instance.</returns>
        private AnekaApplication<AnekaThread,ThreadManager> CreateApplication();
        {
            Configuration conf =new Configuration();
            // this is the common address and port of a local installation
            // of the Aneka Cloud.
            conf.SchedulerUri = newUri("tcp://localhost:9090/Aneka");
            conf.Credentials =newUserCredentials("Administrator", string.Empty);
            // we will not need support for file transfer, hence we optimize the
            // application in order to not require any file transfer service.
            conf.UseFileTransfer = false;
            // we do not need any other configuration setting

            // we create the application instance and configure it.
            AnekaApplication<AnekaThread,ThreadManager> app =
                                new AnekaApplication<AnekaThread,ThreadManager>(conf);
            return app;
         }
    }
```

LISTING 6.8

(Continued)

representing a running program in modern operating systems. Currently, all the most popular operating systems support multithreading, irrespective of whether the underlying hardware explicitly supports real parallelism or not. Real parallelism is supported by the use of multiple processors or cores at the same time, if they are available; otherwise, multithreading is obtained by interleaving the execution of multiple threads on the same processing unit.

To support multithreaded programming, programming languages define the abstraction of process and thread in their class libraries. A popular standard for operations on threads and thread synchronization is POSIX, which is supported by all the Linux/UNIX operating systems and is available as an additional library for the Windows operating systems family. A common implementation of POSIX is given in C/C++ as a library of functions. New-generation languages such as Java and C# (.NET) provide a set of abstractions for thread management and synchronization that is compliant and that most closely follows the object-oriented design that characterizes these languages. These implementations are portable over any operating system that provides an implementation for the runtime environment required by these languages.

Multithreaded programming is a practice that allows achieving parallelism within the boundaries of a single machine. Applications requiring a high degree of parallelism cannot be supported by normal multithreaded programming and must rely on distributed infrastructures such as clusters, grids, or, most recently, clouds. The use of these facilities imposes application redesign and the use of specific APIs, which might require significant changes to the existing applications. To address this issue, Aneka provides the Thread Programming Model, which extends the philosophy behind multithreaded programming beyond the boundaries of a single node and allows leveraging

heterogeneous distributed infrastructure for execution. To minimize application reconversion, the Thread Programming Model mimics the API of the *System.Threading* namespace, with some limitations that are imposed by the fact that threads are executed on a distributed infrastructure. High-throughput applications can be easily ported to Aneka threads with minimal or no changes at all to their logic. Examples of such features and the basic steps of converting a local multithreaded application to Aneka threads were given in the chapter by discussing simple applications demonstrating the methodology of domain and functional decomposition for parallel problems.

As a framework for distributed programming, Aneka provides many built-in features that are not generally of use while architecting an application in terms of concurrent threads. These are, for example, event notification and support for file transfer. These capabilities are available as core features of the Aneka application model but have not been demonstrated in the case of the Thread Programming Model, which is concerned with providing support for partitioning the execution of algorithms to speed up execution. However, they are indeed of great use in the case of "bag of tasks" applications, discussed in the next chapter.

Review questions

1. What is throughput computing and what does it aim to achieve?
2. What is multiprocessing? Describe the different techniques for implementing multiprocessing.
3. What is multicore technology and how does it relate to multiprocessing?
4. Briefly describe the architecture of a multicore system.
5. What is multitasking?
6. What is multithreading and how does it relate to multitasking?
7. Describe the relationship between a process and a thread.
8. Does parallelism of applications depend on parallel hardware architectures?
9. Describe the principal characteristics of a thread from a programming point of view and the uses of threads for parallelizing application execution.
10. What is POSIX?
11. Describe the support given for programming with threads in new-generation languages such as Java or C#.
12. What do the terms *logical thread* and *physical thread* refer to?
13. What are the common operations implemented for a thread?
14. Describe the two major techniques used to define a parallel implementation of computer algorithms.
15. What is an *embarrassingly parallel* problem?
16. Describe how to implement a parallel matrix scalar product by using domain decomposition.
17. How does communication impact design and the implementation of parallel or distributed algorithms?
18. Which kind of support does Aneka provide for multithreading?
19. Describe the major differences between Aneka threads and local threads.
20. What are the limitations of the Thread Programming Model?
21. Design a parallel implementation for the tabulation of the Gaussian function by using simple threads and then convert it to Aneka threads.

CHAPTER

High-Throughput Computing
Task Programming

7

Task computing is a wide area of distributed system programming encompassing several different models of architecting distributed applications, which, eventually, are based on the same fundamental abstraction: the *task*. A task generally represents a program, which might require input files and produce output files as a result of its execution. Applications are then constituted of a collection of tasks. These are submitted for execution and their output data are collected at the end of their execution. The way tasks are generated, the order in which they are executed, or whether they need to exchange data differentiate the application models that fall under the umbrella of task programming.

This chapter characterizes the abstraction of a task and provides a brief overview of the distributed application models that are based on the task abstraction. The Aneka Task Programming Model is taken as a reference implementation to illustrate the execution of *bag-of-tasks (BoT)* applications on a distributed infrastructure.

7.1 Task computing

Organizing an application in terms of tasks is the most intuitive and common practice for developing parallel and distributed computing applications. A task identifies one or more operations that produce a distinct output and that can be isolated as a single logical unit. In practice, a task is represented as a distinct unit of code, or a *program*, that can be separated and executed in a remote runtime environment. Programs are the most common option for representing tasks, especially in the field of scientific computing, which has leveraged distributed computing for its computational needs.

Multithreaded programming is mainly concerned with providing a support for parallelism within a single machine. Task computing provides distribution by harnessing the compute power of several computing nodes. Hence, the presence of a distributed infrastructure is explicit in this model. Historically, the infrastructures that have been leveraged to execute tasks are clusters, supercomputers, and computing grids. Now clouds have emerged as an attractive solution to obtain a huge computing power on demand for the execution of distributed applications. To achieve it, suitable middleware is needed. A reference scenario for task computing is depicted in Figure 7.1.

The *middleware* is a software layer that enables the coordinated use of multiple resources, which are drawn from a datacenter or geographically distributed networked computers. A user

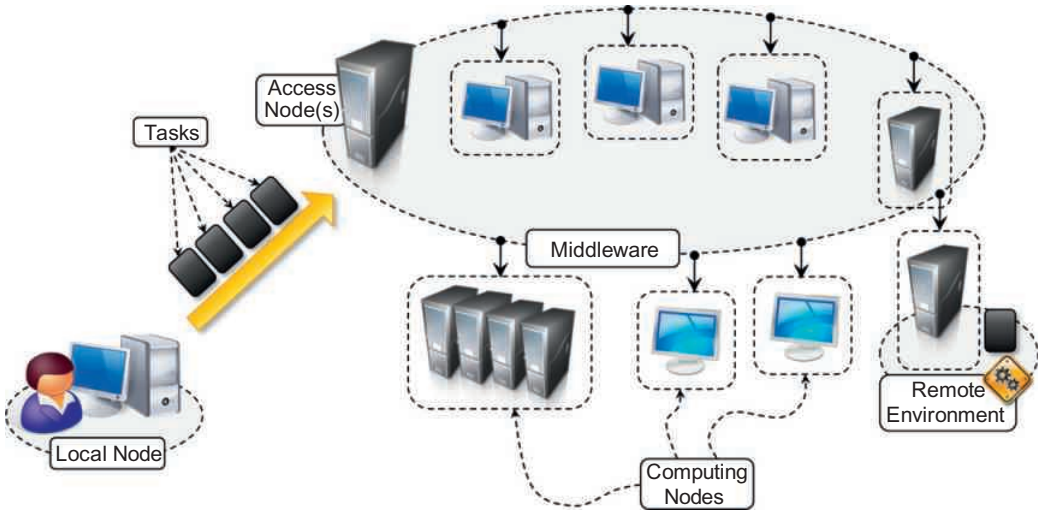


FIGURE 7.1

Task computing scenario.

submits the collection of tasks to the access point(s) of the middleware, which will take care of scheduling and monitoring the execution of tasks. Each computing resource provides an appropriate runtime environment, which may vary from implementation to implementation (a simple shell, a sandboxed environment, or a virtual machine). Task submission is normally done using the APIs provided by the middleware, whether a Web or programming language interface. Appropriate APIs are also provided to monitor task status and collect their results upon completion.

Because task abstraction is general, there exist different models of distributed applications falling under the umbrella of task computing. Despite this variety, it is possible to identify a set of common operations that the middleware needs to support the creation and execution of task-based applications. These operations are:

- Coordinating and scheduling tasks for execution on a set of remote nodes
- Moving programs to remote nodes and managing their dependencies
- Creating an environment for execution of tasks on the remote nodes
- Monitoring each task's execution and informing the user about its status
- Access to the output produced by the task

Models for task computing may differ in the way tasks are scheduled, which in turn depends on whether tasks are interrelated or they need to communicate among themselves.

7.1.1 Characterizing a task

A *task* is a general abstraction that identifies a program or a combination of programs that constitute a computing unit of a distributed application with a tangible output. A task represents a component of an application that can be logically isolated and executed separately. Distributed

applications are composed of tasks, the collective execution and interrelations of which define the nature of the applications. A task can be represented by different elements:

- A shell script composing together the execution of several applications
- A single program
- A unit of code (a Java/C++/.NET class) that executes within the context of a specific runtime environment

A task is generally characterized by input files, executable code (programs, shell scripts, etc.), and output files. In many cases the common runtime environment in which tasks execute is represented by the operating system or an equivalent sandboxed environment. A task may also need specific software appliances on the remote execution nodes in addition to the library dependencies that can be transferred to the node.

Some distributed applications may have additional constraints. For example, distributed computing frameworks that present the abstraction of tasks at programming level, by means of a class to inherit or an interface to implement, might require additional constraints (i.e., compliance to the inheritance rules) but also a richer set of features that can be exploited by developers. Based on the specific model of application, tasks might have dependencies.

7.1.2 Computing categories

According to the specific nature of the problem, a variety of categories for task computing have been proposed over time. These categories do not enforce any specific application model but provide an overall view of the characteristics of the problems. They implicitly impose requirements on the infrastructure and the middleware. Applications falling into this category are *high-performance computing (HPC)*, *high-throughput computing (HTC)*, and *many-task computing (MTC)*.

7.1.2.1 High-performance computing

High-performance computing (HPC) is the use of distributed computing facilities for solving problems that need large computing power. Historically, supercomputers and clusters are specifically designed to support HPC applications that are developed to solve "Grand Challenge" problems in science and engineering. The general profile of HPC applications is constituted by a large collection of compute-intensive tasks that need to be processed in a short period of time. It is common to have parallel and tightly coupled tasks, which require low-latency interconnection networks to minimize the data exchange time. The metrics to evaluate 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 system can perform.

7.1.2.2 High-throughput computing

High-throughput computing (HTC) is the use of distributed computing facilities for applications requiring large computing power over a long period of time. HTC systems need to be robust and to reliably operate over a long time scale. Traditionally, computing grids composed of heterogeneous resources (clusters, workstations, and volunteer desktop machines) have been used to support HTC. The general profile of HTC applications is that they are made up of a large number of tasks of which the execution can last for a considerable amount of time (i.e., weeks or months). Classical

examples of such applications are scientific simulations or statistical analyses. It is quite common to have independent tasks that can be scheduled in distributed resources because they do not need to communicate. HTC systems measure their performance in terms of jobs completed per month.

7.1.2.3 Many-task computing

The *many-task computing (MTC)* [61] model started receiving attention recently and covers a wide variety of applications. It aims to bridge the gap between HPC and HTC. MTC is similar to HTC, but it concentrates on the use of many computing resources over a short period of time to accomplish many computational tasks. In brief, MTC denotes high-performance computations comprising multiple distinct activities coupled via file system operations. What characterizes MTC is the heterogeneity of tasks that might be of considerably different nature: Tasks may be small or large, single-processor or multiprocessor, compute-intensive or data-intensive, static or dynamic, homogeneous or heterogeneous. The general profile of MTC applications includes loosely coupled applications that are generally communication-intensive but not naturally expressed using the message-passing interface commonly found in HPC, drawing attention to the many computations that are heterogeneous but not embarrassingly parallel. Given the large number of tasks commonly composing MTC applications, any distributed facility with a large availability of computing elements is able to support MTC. Such facilities include supercomputers, large clusters, and emerging cloud infrastructures.

7.1.3 Frameworks for task computing

There are several frameworks that can be used to support the execution of task-based applications on distributed computing resources, including clouds. Some popular software systems that support the task-computing framework are *Condor* [5], *Globus Toolkit* [12], *Sun Grid Engine (SGE)* [13], *BOINC* [14], *Nimrod/G* [164], and *Aneka*.

Architecture of all these systems is similar to the general reference architecture depicted in Figure 7.1. They consist of two main components: a scheduling node (one or more) and worker nodes. The organization of the system components may vary. For example, multiple scheduling nodes can be organized in hierarchical structures. This configuration is quite common in the middleware for computing grids, which harness a variety of distributed resources from one or more organizations or sites. Each of these sites may have their own scheduling engine, especially if the system contributes to the grid but also serves local users.

A classic example is the cluster setup where the system might feature an installation of Condor or SGE for batch job submission; these services are generally used locally to the site, but the cluster can be integrated into a larger grid where meta-schedulers such as *GRAM (Globus Resource Allocation Manager)*[1] can dispatch a collection of jobs to the cluster. Other options include the presence of gateway nodes that do not have any scheduling capabilities and simply constitute the access point to the system. These nodes have indexing services that allow users to identify the available resources in the system, its current status, and the available schedulers. For worker nodes, they generally provide a sandboxed environment where tasks are executed on behalf of a specific

[1]Globus Resource Allocation Manager, or GRAM, is a software component of the Globus Toolkit that is in charge of locating, submitting, monitoring, and canceling jobs in grid computing systems.

user or within a given security context, limiting the operations that can be performed by programs such as file system access. File staging is also a fundamental feature supported by these systems. Clusters are normally equipped with shared file systems and parallel I/O facilities. Grids provide users with various staging facilities, such as credential access to remote worker nodes or automated staging services that transparently move files from user local machine to remote nodes.

Condor is probably the most widely used and long-lived middleware for managing clusters, idle workstations, and a collection of clusters. Condor-G is a version of Condor that supports integration with grid computing resources, such as those managed by Globus. Condor supports common features of batch-queuing systems along with the capability to checkpoint jobs and manage overload nodes. It provides a powerful job resource-matching mechanism, which schedules jobs only on resources that have the appropriate runtime environment. Condor can handle both serial and parallel jobs on a wide variety of resources. It is used by hundreds of organizations in industry, government, and academia to manage infrastructures ranging from a handful to well over thousands of workstations.

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

The Globus Toolkit is a collection of technologies that enable grid computing. It provides a comprehensive set of tools for sharing computing power, databases, and other services across corporate, institutional, and geographic boundaries without sacrificing local autonomy. The toolkit features software services, libraries, and tools for resource monitoring, discovery, and management as well as security and file management. The Globus Toolkit addresses core issues of grid computing: the management of a distributed environment composed of heterogeneous resources spanning different organizations, with all this condition implies in terms of security and interoperation. To provide valid support for grid computing in such scenarios, the toolkit defines a collection of interfaces and protocol for interoperation that enable different systems to integrate with each other and expose resources outside their boundaries.

Nimrod/G [164] is a tool for automated modeling and execution of parameter sweep applications (parameter studies) over global computational grids. It provides a simple declarative parametric modeling language for expressing parametric experiments. A domain expert can easily create a plan for a parametric experiment and use the Nimrod/G system to deploy jobs on distributed resources for execution. It has been used for a very wide range of applications over the years, ranging from quantum chemistry to policy and environmental impact. Moreover, it uses novel resource management and scheduling algorithms based on economic principles. Specifically, it supports deadline- and budget-constrained scheduling of applications on distributed grid resources to minimize the execution cost and at the same deliver results in a timely manner.

Berkeley Open Infrastructure for Network Computing (BOINC) is framework for volunteer and grid computing. It allows us to turn desktop machines into volunteer computing nodes that are leveraged to run jobs when such machines become inactive. BOINC is composed of two main components: the *BOINC server* and the *BOINC client*. The former is the central node that keeps track of

all the available resources and scheduling jobs; the latter is the software component that is deployed on desktop machines and that creates the BOINC execution environment for job submission. Given the volatility of BOINC clients, BOINC supports job checkpointing and duplication. Even if mostly focused on volunteer computing, BOINC systems can be easily set up to provide more stable support for job execution by creating computing grids with dedicated machines. To leverage BOINC, it is necessary to create an application project. When installing BOINC clients, users can decide the application project to which they want to donate the CPU cycles of their computer. Currently several projects, ranging from medicine to astronomy and cryptography, are running on the BOINC infrastructure.

7.2 Task-based application models

There are several models based on the concept of the task as the fundamental unit for composing distributed applications. What makes these models different from one another is the way in which tasks are generated, the relationships they have with each other, and the presence of dependencies or other conditions—for example, a specific set of services in the runtime environment—that have to be met. In this section, we quickly review the most common and popular models based on the concept of the task.

7.2.1 Embarrassingly parallel applications

Embarrassingly parallel applications constitute the most simple and intuitive category of distributed applications. As we discussed in Chapter 6, embarrassingly parallel applications constitute a collection of tasks that are independent from each other and that can be executed in any order. The tasks might be of the same type or of different types, and they do not need to communicate among themselves.

This category of applications is supported by the majority of the frameworks for distributed computing. Since tasks do not need to communicate, there is a lot of freedom regarding the way they are scheduled. Tasks can be executed in any order, and there is no specific requirement for tasks to be executed at the same time. Therefore, scheduling these applications is simplified and mostly concerned with the optimal mapping of tasks to available resources. Frameworks and tools supporting embarrassingly parallel applications are the Globus Toolkit, BOINC, and Aneka.

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

7.2.2 Parameter sweep applications

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

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

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

```
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 algorithm sketched in the example defines a bidimensional domain composed of discrete variables and then iterated over each combination of individuals and generations to generate all the tasks composing the application. In this case 20 tasks are generated. The function *generate_task* is specific to the application and creates the task instance by substituting the values of *indiv* and *generation* to the corresponding variables in the template definition. The function *submit_task* is specific to the middleware used and performs the actual task submission.

A template task is in general a composition of operations concerning the execution of legacy applications with the appropriate parameters and set of file system operations for moving data. Therefore, frameworks that natively support the execution of parameter sweep applications often

provide a set of useful commands for manipulating or operating on files. Also, the template task is often expressed as single file that composes together the commands provided. The commonly available commands are:

- *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.

All these commands can operate with parameters that are substituted with their actual values for each task instance.

Figures 7.2 and 7.3 provide examples of two possible task templates, the former as defined according to the notation used by Nimrod/G, and the latter as required by Aneka.

The template file has two sections: a header for the definition of the parameters, and a task definition section that includes shell commands mixed with Nimrod/G commands. The prefix *node:* identifies the remote location where the task is executed. Parameters are identified with the *${...}* notation. The example shown remotely executes the *echo* command and copies to the local user directory the output of the command by saving it into a file named according to the values of the parameters *x* and *y*.

The Aneka Parameter Sweep file defines the template task for executing the BLAST application. The file is an XML document containing several sections, the most important of which are *sharedFiles*, *parameters*, and *task*. *parameters* contains the definition of the parameters that will customize the template task. Two different types of parameters are defined: a single value and a range parameter. The *sharedFiles* section contains the files that are required to execute the task; *task* specifies the operations that characterize the template task. The task has a collection of input and output files for which local and remote paths are defined, as well as a collection of commands. In the case presented, a simple *execute* command is shown. With respect to the previous example there is no need to explicitly move the files to the remote destination, but this operation is automatically performed by Aneka.

7.2.3 MPI applications

Message Passing Interface (MPI) is a specification for developing parallel programs that communicate by exchanging messages. Compared to earlier models, MPI introduces the constraint of communication that involves MPI tasks that need to run at the same time. MPI has originated as an attempt to create common ground from the several distributed shared memory and message-passing

```
parameter x float range from 1 to 10 step 1;
parameter y float range from -4 to 5 step 1;

task main
  node:execute /bin/echo X:${x} Y:${y} > output
  copy node:output output.`expr ${y}\*10+${x}`
endtask
```

FIGURE 7.2

Nimrod/G task template definition.

```
<psm>
  <name>Aneka Blast</name>
  <description>BLAST simulation</description>
  <workspace>C:\Projects\Explorer\blast</workspace>
  <parameters>
    <single name="p" type="String" comment="The name of the program" value="blastn"/>
    <single name="d" type="String" comment="The database file" value="ecoli.nt"/>
    <range name="s" type="String" comment="The sequence file" from="0" to="2" interval="1"/>
  </parameters>
  <sharedFiles>
    <file path="blastall.exe" vpath="blastall.exe"/>
    <file path="ecoli.nt.nhr" vpath="ecoli.nt.nhr"/>
    <file path="ecoli.nt.nin" vpath="ecoli.nt.nin"/>
    <file path="ecoli.nt.nnd" vpath="ecoli.nt.nnd"/>
    <file path="ecoli.nt.nni" vpath="ecoli.nt.nni"/>
    <file path="ecoli.nt.nsd" vpath="ecoli.nt.nsd"/>
    <file path="ecoli.nt.nsi" vpath="ecoli.nt.nsi"/>
    <file path="ecoli.nt.nsq" vpath="ecoli.nt.nsq"/>
  </sharedFiles>
  <task>
    <inputs>
      <file path="seq($s).txt" vpath="seq($s).txt"/>
    </inputs>
    <outputs>
      <file path="output($s).txt" vpath="output($s).txt"/>
    </outputs>
    <commands>
      <execute cmd="blastall.exe" args="-p ($p) -d ($d) -i seq($s).txt -o output($s).txt"/>
    </commands>
  </task>
</psm>
```

FIGURE 7.3

Aneka parameter sweep file.

infrastructures available for distributed computing. Nowadays, MPI has become a *de facto* standard for developing portable and efficient message-passing HPC applications. Interface specifications have been defined and implemented for C/C++ and Fortran.

MPI provides developers with a set of routines that:

- Manage the distributed environment where MPI programs are executed

- Provide facilities for point-to-point communication
- Provide facilities for group communication
- Provide support for data structure definition and memory allocation
- Provide basic support for synchronization with blocking calls

The general reference architecture is depicted in Figure 7.4. A distributed application in MPI is composed of a collection of MPI processes that are executed in parallel in a distributed infrastructure that supports MPI (most likely a cluster or nodes leased from clouds).

MPI applications that share the same MPI runtime are by default as part of a global group called *MPI_COMM_WORLD*. Within this group, all the distributed processes have a unique identifier that allows the MPI runtime to localize and address them. It is possible to create specific groups as subsets of this global group—for example, for isolating all the MPI processes that belong to the same application. Each MPI process is assigned a rank within the group to which it belongs. The rank is a unique identifier that allows processes to communicate with each other within a group. Communication is made possible by means of a communicator component that can be defined for each group.

To create an MPI application it is necessary to define the code for the MPI process that will be executed in parallel. This program has, in general, the structure described in Figure 7.5. The section of code that is executed in parallel is clearly identified by two operations that set up the MPI environment and shut it down, respectively. In the code section defined within these two operations, it is possible to use all the MPI functions to send or receive messages in either asynchronous or synchronous mode.

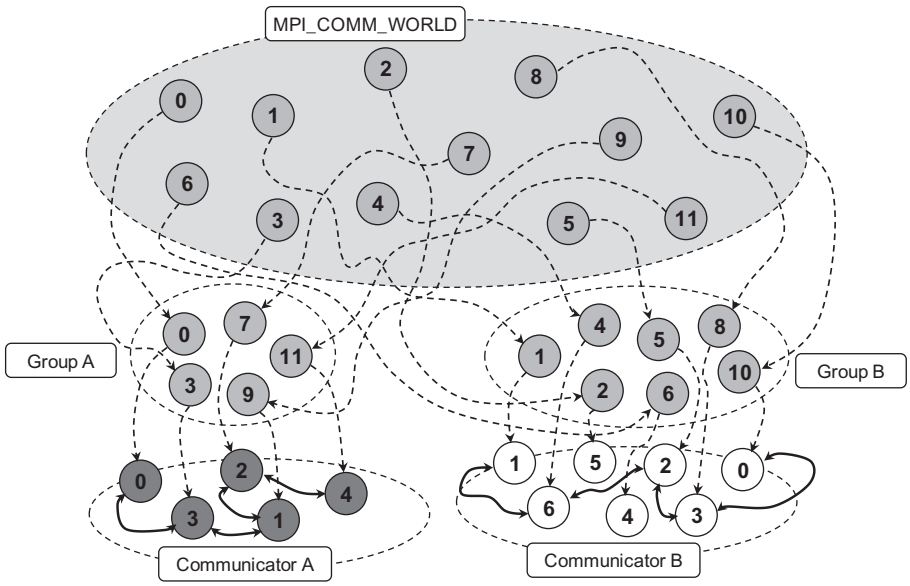


FIGURE 7.4

MPI reference scenario.

The diagram in Figure 7.5 might suggest that the MPI might allow the definition of completely symmetrical applications, since the portion of code executed in each node is the same. In reality, it is possible to implement distributed applications based on complex communication patterns by differentiating the operations performed by each node according to the rank of the program, which is known at runtime. A common model used in MPI is the *master-worker model*, whereby one MPI process (usually the one with rank 0) coordinates the execution of others that perform the same task.

Once the program has been defined in one of the available MPI implementations, it is compiled with a modified version of the compiler for the language. This compiler introduces additional code in order to properly manage the MPI runtime. The output of the compilation process can be run as a distributed application by using a specific tool provided with the MPI implementation.

A general installation that supports the execution of the MPI application is composed of a cluster. In this scenario MPI is normally installed in the shared file system and an MPI daemon is started on each node of the cluster in order to coordinate the parallel execution of MPI applications. Once the environment is set up, it is possible to run parallel applications by using the tools provided with the MPI implementation and to specify several options, such as the number of nodes to use to run the application.

At present there are several MPI implementations that can be leveraged to develop distributed applications, and the MPI specifications have currently reached version 2. One of the most popular MPI software environments (www.mcs.anl.gov/mpi/) is developed by the Argonne National Laboratory in the United States. MPI has gained a good deal of success as a parallel and distributed

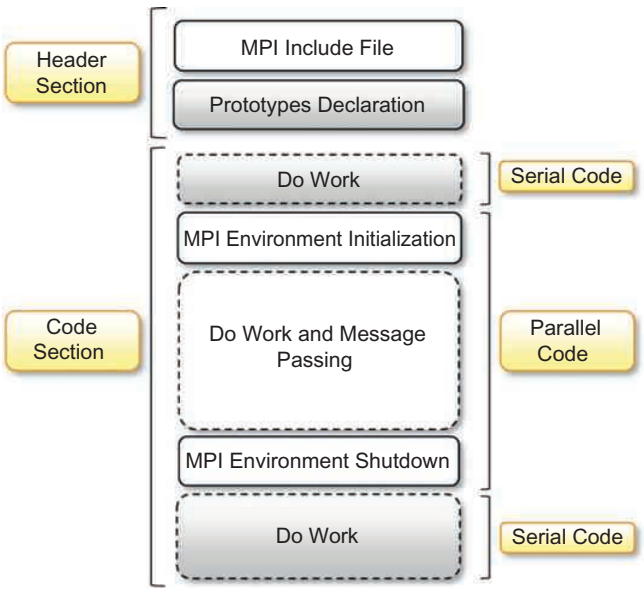


FIGURE 7.5

MPI program structure.

programming model for CPU-intensive mathematical computations such as linear systems solvers, matrix computations, finite element computations, linear algebra, and numerical simulations.

7.2.4 Workflow applications with task dependencies

Workflow applications are characterized by a collection of tasks that exhibit dependencies among them. Such dependencies, which are mostly data dependencies (i.e., the output of one task is a prerequisite of another task), determine the way in which the applications are scheduled as well as where they are scheduled. Concerns in this case are related to providing a feasible sequencing of tasks and to optimizing the placement of tasks so that the movement of data is minimized.

7.2.4.1 What is a workflow?

The term *workflow* has a long tradition in the business community, where the term is used to describe a composition of services that all together accomplish a business process. As defined by the Workflow Management Coalition, a workflow is *the automation of a business process, in whole or part, during which documents, information, or tasks are passed from one participant (a resource; human or machine) to another for action, according to a set of procedural rules* [64]. The concept of workflow as a structured execution of tasks that have dependencies on each other has demonstrated itself to be useful for expressing many scientific experiments and gave birth to the idea of *scientific workflow*. Many scientific experiments are a combination of problem-solving components, which, connected in a particular order, define the specific nature of the experiment. When such experiments exhibit a natural parallelism and need to execute a large number of operations or deal with huge quantities of data, it makes sense to execute them on a distributed infrastructure. In the case of scientific workflows, the process is identified by an application to run, the elements that are passed among participants are mostly tasks and data, and the participants are mostly computing or storage nodes. The set of procedural rules is defined by a workflow definition scheme that guides the scheduling of the application. A scientific workflow generally involves data management, analysis, simulation, and middleware supporting the execution of the workflow.

A scientific workflow is generally expressed by a *directed acyclic graph (DAG)*, which defines the dependencies among tasks or operations. The nodes on the DAG represent the tasks to be executed in a workflow application; the arcs connecting the nodes identify the dependencies among tasks and the data paths that connect the tasks. The most common dependency that is realized through a DAG is *data dependency*, which means that the output files of a task (or some of them) constitute the input files of another task. This dependency is represented as an arc originating from the node that identifies the first task and terminating in the node that identifies the second task.

The DAG in Figure 7.6 describes a sample Montage workflow.[2] *Montage* is a toolkit for assembling images into mosaics; it has been specially designed to support astronomers in composing the images taken from different telescopes or points of view into a coherent image. The toolkit provides several applications for manipulating images and composing them together; some of the applications perform background reprojection, perspective transformation, and brightness and color

[2]Montage workflow: http://vgrads.rice.edu/research/applications/images/montage-workflow/view.

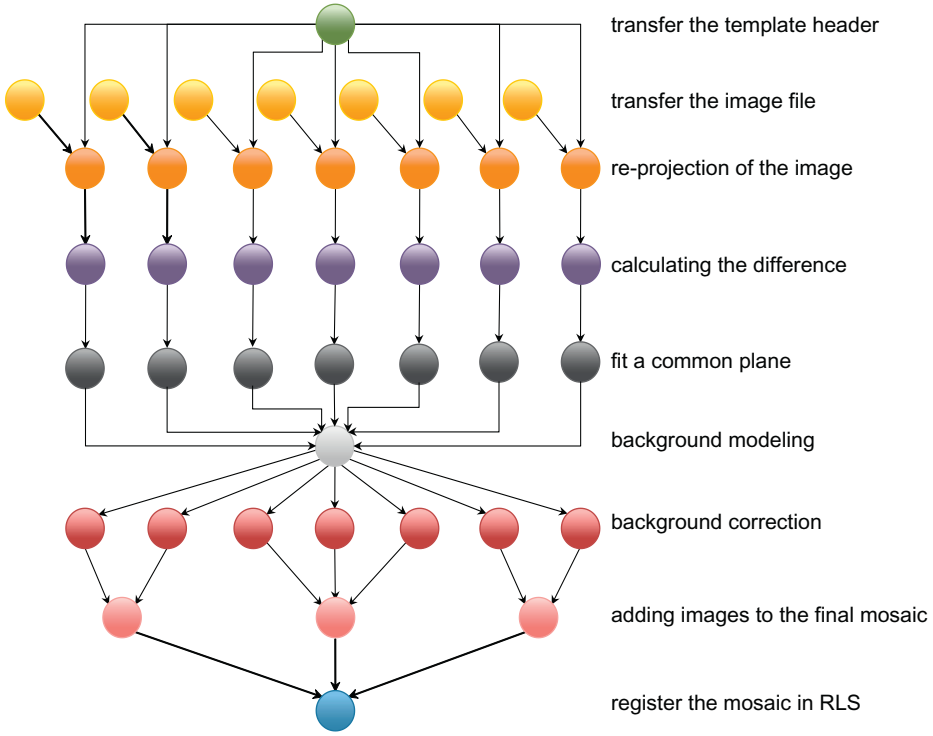


FIGURE 7.6

Sample Montage workflow.

correction. The workflow depicted in Figure 7.6 describes the general process for composing a mosaic; the labels on the right describe the different tasks that have to be performed to compose a mosaic. In the case presented in the diagram, a mosaic is composed of seven images. The entire process can take advantage of a distributed infrastructure for its execution, since there are several operations that can be performed in parallel. For each of the image files, the following process has to be performed: image file transfer, reprojection, calculation of the difference, and common plane placement. Therefore, each of the images can be processed in parallel for these tasks. Here is where a distributed infrastructure helps in executing workflows.

There might be another reason for executing workflows on a distributed infrastructure: It might be convenient to move the computation on a specific node because of data locality issues. For example, if an operation needs to access specific resources that are only available on a specific node, that operation cannot be performed elsewhere, whereas the rest of the operations might not have the same requirements. A scientific experiment might involve the use of several problem-solving components that might require the use of specific instrumentation; in this case all the tasks that have these constraints need to be executed where the instrumentation is available, thus creating a distributed execution of a process that is not parallel in principle.

7.2.4.2 Workflow technologies

Business-oriented computing workflows are defined as compositions of services, and there are specific languages and standards for the definition of workflows, such as *Business Process Execution Language (BPEL)* [65]. In the case of scientific computing there is no common ground for defining workflows, but several solutions and workflow languages coexist [66]. Despite such differences, it is possible to identify an abstract reference model for a workflow management system [67], as depicted in Figure 7.7. Design tools allow users to visually compose a workflow application. This specification is normally stored in the form of an XML document based on a specific workflow language and constitutes the input of the workflow engine, which controls the execution of the workflow by leveraging a distributed infrastructure. In most cases, the workflow engine is a client-side component that might interact directly with resources or with one or several middleware components for executing the workflow. Some frameworks can natively support the execution of workflow applications by providing a scheduler capable of directly processing the workflow specification.

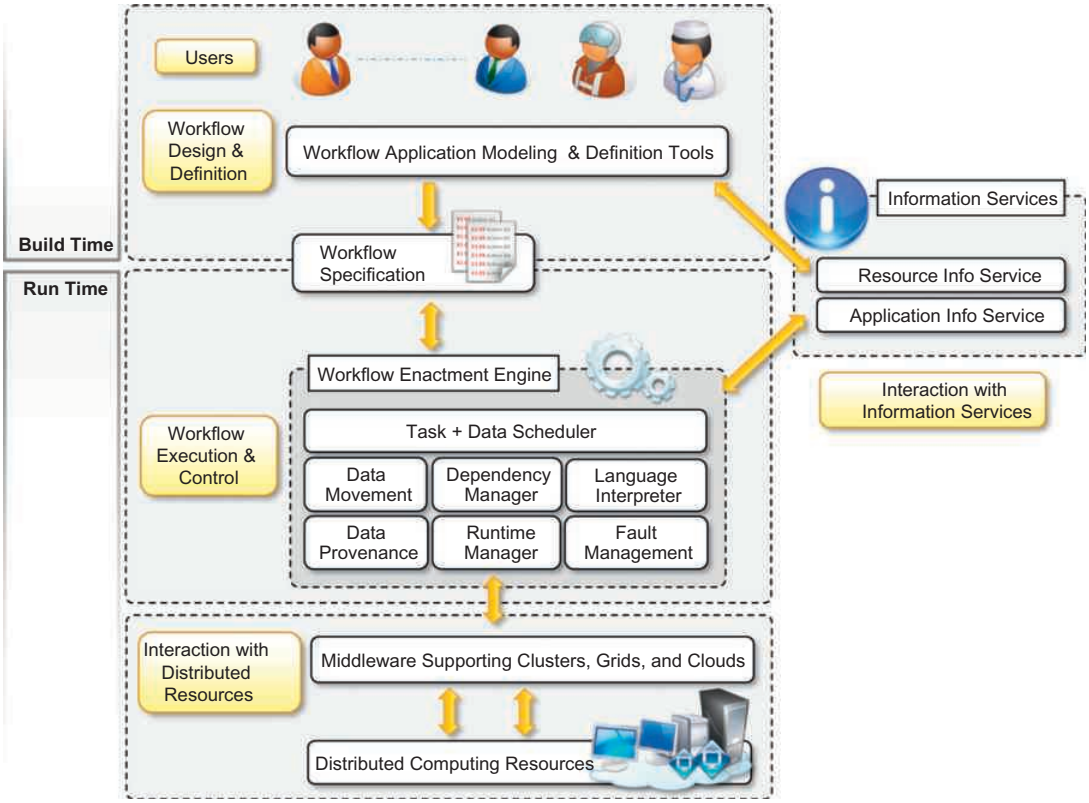


FIGURE 7.7

Abstract model of a workflow system.

Some of the most relevant technologies for designing and executing workflow-based applications are *Kepler*, *DAGMan*, *Cloudbus Workflow Management System*, and *Offspring*.

Kepler [68] is an open-source scientific workflow engine built from the collaboration of several research projects. The system is based on the *Ptolemy II* system [72], which provides a solid platform for developing dataflow-oriented workflows. Kepler provides a design environment based on the concept of actors, which are reusable and independent blocks of computation such as Web services, database calls, and the like. The connection between actors is made with ports. An actor consumes data from the input ports and writes data/results to the output ports. The novelty of Kepler is in its ability to separate the flow of data among components from the coordination logic that is used to execute workflow. Thus, for the same workflow, Kepler supports different models, such as synchronous and asynchronous models. The workflow specification is expressed using a proprietary XML language.

DAGMan (Directed Acyclic Graph Manager) [69], part of the Condor [5] project, constitutes an extension to the Condor scheduler to handle job interdependencies. Condor finds machines for the execution of programs but does not support the scheduling of jobs in a specific sequence. Therefore, DAGMan acts as a metascheduler for Condor by submitting the jobs to the scheduler in the appropriate order. The input of DAGMan is a simple text file that contains the information about the jobs, pointers to their job submission files, and the dependencies among jobs.

Cloudbus Workflow Management System (WfMS) [70] is a middleware platform built for managing large application workflows on distributed computing platforms such as grids and clouds. It comprises software tools that help end users compose, schedule, execute, and monitor workflow applications through a Web-based portal. The portal provides the capability of uploading workflows or defining new ones with a graphical editor. To execute workflows, WfMS relies on the Gridbus Broker, a grid/cloud resource broker that supports the execution of applications with quality-of-service (QoS) attributes over a heterogeneous distributed computing infrastructure, including Linux-based clusters, Globus, and Amazon EC2. WfMS uses a proprietary XML language for the specification of workflows.

A different perspective is taken by *Offspring* [71], which offers a programming-based approach to developing workflows. Users can develop strategies and plug them into the environment, which will execute them by leveraging a specific distribution engine. The advantage provided by Offspring over other solutions is the ability to define dynamic workflows. This strategy represents a semistructured workflow that can change its behavior at runtime according to the execution of specific tasks. This allows developers to dynamically control the dependencies of tasks at runtime rather than statically defining them. Offspring supports integration with any distributed computing middleware that can manage a simple bag-of-tasks application. It provides a native integration with Aneka and supports a simulated distribution engine for testing strategies during development. Because Offspring allows the definition of workflows in the form of plug-ins, it does not use any XML specification.

7.3 Aneka task-based programming

Aneka provides support for all the flavors of task-based programming by means of the *Task Programming Model*, which constitutes the basic support given by the framework for supporting

the execution of bag-of-tasks applications. Task programming is realized through the abstraction of the *Aneka.Tasks.ITask*. By using this abstraction as a basis support for execution of legacy applications, parameter sweep applications and workflows have been integrated into the framework. In this section, we introduce the fundamental concepts of the model and provide examples of how to develop applications for all the previously discussed application models.

7.3.1 Task programming model

The *Task Programming Model* provides a very intuitive abstraction for quickly developing distributed applications on top of Aneka. It provides a minimum set of APIs that are mostly centered on the *Aneka.Tasks.ITask* interface. This interface, together with the services supporting the execution of tasks in the middleware, constitutes the core feature of the model. Figure 7.8 provides an overall view of the components of the Task Programming Model and their roles during application execution.

Developers create distributed applications in terms of *ITask* instances, the collective execution of which describes a running application. These tasks, together with all the required dependencies (data files and libraries), are grouped and managed through the *AnekaApplication* class, which is specialized to support the execution of tasks. Two other components, *AnekaTask* and *TaskManager*, constitute the client-side view of a task-based application. The former constitutes the runtime wrapper Aneka uses to represent a task within the middleware; the latter is the underlying component that interacts with Aneka, submits the tasks, monitors their execution, and collects the results. In the middleware, four services coordinate their activities in order to execute task-based applications. These are *MembershipCatalogue*, *TaskScheduler*, *ExecutionService*, and *StorageService*.

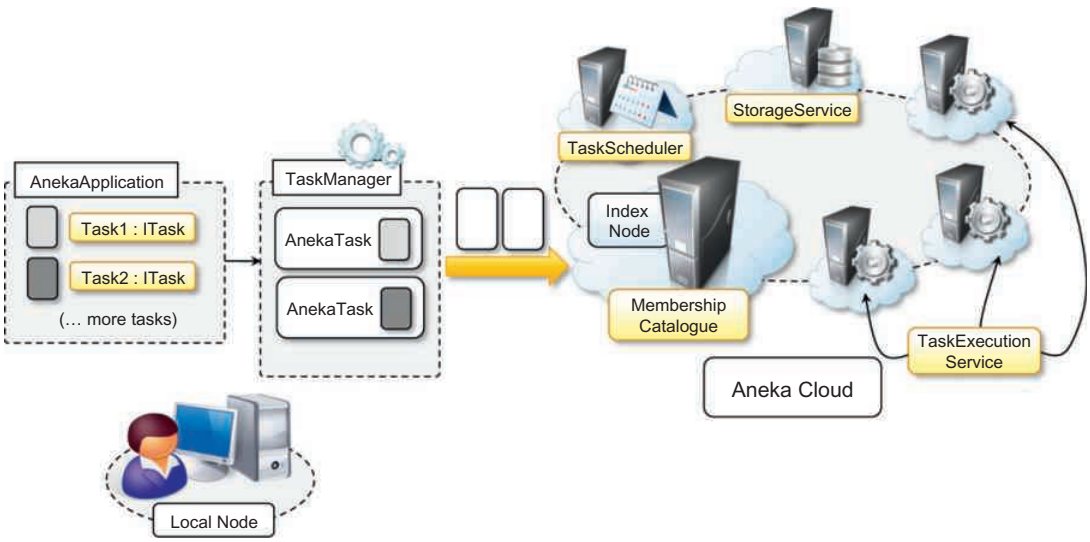


FIGURE 7.8

Task programming model scenario.

MembershipCatalogue constitutes the main access point of the cloud and acts as a service directory to locate the *TaskScheduler* service that is in charge of managing the execution of task-based applications. Its main responsibility is to allocate task instances to resources featuring the *ExecutionService* for task execution and for monitoring task state. If the application requires the data transfer support in the form of data files, input files, or output files, an available *StorageService* will be used as a staging facility for the application.

The features provided by the task model are completed by a Web service that allows any client to submit the execution of tasks to Aneka. The procedure for submitting tasks through the Web services is the same as that done by using the framework APIs. The user creates an application on Aneka and submits tasks within the context of this application. The Web service limits the type of tasks that can be submitted. Only a limited collection of tasks is available for submission; despite that, these tasks cover the functionality commonly found in other distributed computing systems.

7.3.2 Developing applications with the task model

Execution of task-based applications involves several components. The development of such applications is limited to the following operations:

- 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

Moreover, from a design point of view, the process of defining a task application ultimately reduces to the definition of the classes that implement *ITask*, which will be those that contribute to form the workload generated by the application.

7.3.2.1 ITask and AnekaTask

Almost all the client-side features for developing task-based applications with Aneka are contained in the *Aneka.Tasks* namespace (*Aneka.Tasks.dll*). The most important component for designing tasks is the *ITask* interface, which is defined in Listing 7.1. This interface exposes only one method: *Execute*. The method is invoked in order to execute the task on the remote node.

The *ITask* interface provides a programming approach for developing native tasks, which means tasks implemented in any of the supported programming languages of the .NET framework. The restrictions on implementing task classes are minimal; other than implementing the *ITask* interface, they need to be serializable, since task instances are created and moved over the network. Listing 7.2 describes a simple implementation of a task class that computes the Gaussian distribution for a given point *x*.

ITask provides minimum restrictions on how to implement a task class and decouples the specific operation of the task from the runtime wrapper classes. It is required for managing tasks within Aneka. This role is performed by the *AnekaTask* class that represents the task instance in accordance with the Aneka application model APIs. This class extends the *Aneka.Entity.WorkUnit* class and provides the feature for embedding *ITask* instances. *AnekaTask* is mostly used internally, and for end users it provides facilities for specifying input and output files for the task.

```
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();
    }
}
```

LISTING 7.1

ITask interface.

Listing 7.3 describes how to wrap an *ITask* instance into an *AnekaTask*. It also shows how to add input and output files specific to a given task. The Task Programming Model leverages the basic capabilities for file management that belong to the *WorkUnit* class, from which the *AnekaTask* class inherits. As discussed when we presented the Aneka Application Model (see Chapter 5). *WorkUnit* has two collections of files, *InputFiles* and *OutputFiles*; developers can add files to these collections and the runtime environment will automatically move these files where it is necessary. Input files will be staged into the Aneka Cloud and moved to the remote node where the task is executed. Output files will be collected from the execution node and moved to the local machine or a remote FTP server.

7.3.2.2 Controlling task execution

Task classes and *AnekaTask* define the computation logic of a task-based application, whereas the *AnekaApplication* class provides the basic feature for implementing the coordination logic of the application.

AnekaApplication is a generic class that can be specialized to support different programming models. In task programming, it assumes the form of *AnekaApplication* $<$ *AnekaTask, TaskManager* $>$. The operations provided for the task model as well as for other programming models are:

- Static and dynamic task submission
- Application state and task state monitoring
- Event-based notification of task completion or failure

By composing these features all together, it is possible to define the logic that is required to implement a specific task application. Static submission is a very common pattern in the case of task-based applications, and it involves the creation of all the tasks that need to be executed in one loop and their submission as a single bag. More complex task submission strategies are then required for implementing workflow-based applications, where the execution of tasks is determined by dependencies among them. In this case a dynamic submission of tasks is a more efficient

```
// File: GaussTask.cs
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);
      }
   }
}
```

LISTING 7.2

ITask interface implementation.

technique and involves the submission of tasks as a result of the event-based notification mechanism implemented in the *AnekaApplication* class.

Listing 7.4 shows how to create and submit 400 Gauss tasks as a bag by using the static submission approach. The *AnekaApplication* class has a collection of tasks containing all the tasks that have been submitted for execution. Each task can be referenced using its unique identifier (*WorkUnit.Id*) by the indexer operator [] applied to the application class. In the case of static submission, the tasks are added to the application, and the method *SubmitExecution()* is called.

```
// create a Gauss task and wraps it into an AnekaTaskinstance
GaussTaskgauss = new GaussTask();
AnekaTask task = newAnekaTask(gauss);
// add one input and one output files
task.AddFile("input.txt", FileDataType.Input, FileAttributes.Local);
task.AddFile("result.txt", FileDataType.Output, FileAttributes.Local);
```

LISTING 7.3

Wrapping an *ITask* into an *AnekaTask* Instance.

```
// 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();
```

LISTING 7.4

Static task submission.

A different scenario is constituted by dynamic submission, where tasks are submitted as a result of other events that occur during the execution—for example, the completion or the failure of previously submitted tasks or other conditions that are not related to the interaction of Aneka. In this case, developers have more freedom in selecting the most appropriate task submission strategy. For example, it is possible that an initial bag of tasks is submitted as described in Listing 7.4, and subsequently, as a result of the completion of some tasks, other tasks are generated and submitted. To implement this scenario it is necessary to rely on the event-based notification system provided by the *AnekaApplication* class and trigger the submission of tasks according to the firing of specific events. In particular, we are interested in the *WorkUnitFailed* and *WorkUnitCompleted* events.

Listing 7.5 extends the previous example and implements a dynamic task submission strategy for refining the computation of Gaussian distribution. Both static and dynamic task submissions are used: An initial bag of 400 Gauss tasks is submitted to Aneka, and as soon as these tasks are completed, a new task for the computation of an intermediate value of the distribution is submitted. To capture the failure and the completion of tasks, it is necessary to listen to the events

```
///<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);
   }
}
```

LISTING 7.5

Dynamic task submission.

WorkUnitFailed and *WorkUnitFinished*. The event signature requires the methods to have an object parameter (as for all the event handlers), which will contain the application instance, and a *WorkUnitEventArgs*$<$*AnekaTask*$>$ argument containing the information about the *WorkUnit* that triggered the event. This class exposes a *WorkUnit* property that, if not null, gives access to the

```
/// <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

      GaussTask fraq = 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);
}
```

LISTING 7.5

Dynamic task submission.

task instance. The event handler for the task failure simply dumps the information that the task is failed to the console, with, if possible, additional information about the error that occurred. The event handler for task completion checks whether the task completed was submitted within the original bag, and in this case submits another task by using the *ExecuteWorkUnit(AnekaTask task)* method. To discriminate tasks submitted within the initial bag and other tasks, the value of *GaussTask.X* is used. If *X* contains a value with no fractional digits, it is an initial task; otherwise, it is not.

Static and dynamic submission influence the way the application termination condition is determined. In static submission the determination of this condition is automatic: Once all the initially submitted tasks are failed or completed, the application is terminated. It is important in this case to activate, in the configuration of the application, the *SingleSubmission* flag by setting it to *true*. This will tell the runtime to automatically determine the completion of the application. In dynamic submission, it is impossible for the runtime to determine the termination of the application, since it is always to possible to submit new tasks. In this case it is the responsibility of the developer to signal to the application class the termination of the application by invoking the *StopExecution* method when appropriate.

In designing the coordination logic of the application, it is important to note that the task submission identifies an asynchronous execution pattern, which means that the method *SubmitExecution*, as well as the method *ExecuteWorkUnit*, returns when the submission of tasks is completed, but not the actual completion of tasks. This requires the developer to put in place the proper synchronization logic to let the main thread of the application wait until all the tasks are terminated and the application is completed. This behavior can be implemented using the synchronization APIs provided by the *System.Threading* namespace: *System.Threading.AutoResetEvent* or *System.Threading.ManualResetEvent.* These two APIs, together with a minimal logic, count all the tasks to be collected and signal the main thread (put in waiting state by calling the method *WaitHandle.Wait()*) once all tasks are terminated. It can also provide the required infrastructure for properly managing the execution flow of the application. Listing 7.6 provides a complete implementation of the task submission program, implementing dynamic submission and the appropriate synchronization logic.

The listing provides a complete definition of the *GaussApp* class, which also contains the main entry point of the application. A very simple logic for controlling the execution of the application has been implemented. The *GaussApp* application keeps track of the number of currently running tasks by using the *taskCount* field. When this value reaches zero, there are no more tasks to wait for and the application is stopped by calling *StopExecution*. This method fires the *ApplicationFinished* event whose event handler (*OnApplicationFinished*) unblocks the main thread by signalling the semaphore. The value of *taskCount* is initially set to 400, which is the size of the initial bag of tasks. Every time a task fails or completes, this field is decremented by one unit; if there is a new task submission, the field is incremented by one unit. At the end of the two event handlers (*OnWorkUnitFailed* and *OnWorkUnitFinished*), the value of *taskCount* is checked to see whether it is equal to zero and it is necessary to stop the application. We can observe that, besides the use of the *ManualResetEvent*, there is no need for other synchronization structures. Because the code that manipulates the value of *taskCount* is executed in one single thread, there will not be any races while incrementing or decrementing the value.

A final aspect that can be considered for controlling the execution of the task application is the resubmission strategy that is used. By default the configuration of the application sets the resubmission strategy as manual; this means that if a task fails because of an exception that occurred during its execution, the task instance is sent back to the client application, and it is the developer's responsibility to resubmit the task if necessary. In automatic resubmission, Aneka will keep resubmitting the task until a maximum number of attempts is reached. If the task keeps failing, the task failure event will eventually be fired. This property can be controlled by setting the configuration value *Configuration.ResubmitMode* to *ResubmitMode.Manual* or *ResubmitMode.Auto*. As previously stated, this property is set to *ResubmitMode.Manual* by default.

7.3.2.3 File management

Task-based applications normally deal with files to perform their operations. As we've discussed, files may constitute input data for tasks, may contain the result of a computation, or may represent executable code or library dependencies. Therefore, providing support for file transfers for task-based applications is essential. Aneka provides built-in capabilities for file management in a distributed infrastructure, and the Task Programming Model transparently leverages these capabilities. Any model based on the *WorkUnit* and *ApplicationBase* classes has built-in support for file

```
// File: GaussApp.cs
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 ManualResetEvent semaphore;
      /// <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);
         }
```

LISTING 7.6

GaussApplication.

```
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();
}
```

LISTING 7.6

(Continued)

```
/// <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)
   {
      // ok it was an original task, then we increment of 0.5 the
      // value of X and submit another task
      GaussTask fraq = GaussTask();
      fraq.X = gauss.X + 0.5;
      AnekaTask task =new AnekaTask(fraq);
```

LISTING 7.6

(Continued)

```
                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, ApplicationEventArgs args)
        {
            // unblock the main thread, because we have identified the termination
            // of the application
            this.semaphore.Set();
        }
    }
}
```

LISTING 7.6

(Continued)

management. It is possible to provide input files that are common to all the *WorkUnit* instances, through the *ApplicationBase.SharedFiles* collection, and instance-specific input and output files by leveraging the *WorkUnit.InputFiles* and *WorkUnit.OutputFiles* collections.

A fundamental component for the management of files is the *FileData* class, which constitutes the logic representation of physical files, as defined in the *Aneka.Data.Entity* namespace (*Aneka.Data.dll*). A *FileData* instance provides information about a file:

- Its nature: whether it is a shared file, an input file, or an output file
- Its path both in the local and in the remote file system, including a different name
- A collection of attributes that provides other information (such as the final destination of the file or whether the file is transient or not, etc.)

Using the *FileData* class, the user specifies the file dependencies of tasks and the application, and the Aneka APIs will automatically transfer them to and from the Aneka Cloud when needed. Aneka allows specifying of both local and remote files stored on FTP servers or Amazon S3. A *FileData* instance is identified by three elements: an owner, a name, and a type. By means of the corresponding ID, the owner identifies which is the computing element that needs the file: application instance or work unit. The type specifies whether the file is shared or an input or output file. The name represents the name of the corresponding physical file.

Listing 7.7 demonstrates how to add file dependencies to the application and to tasks. It is possible to add both *FileData* instances, thus having more control of the information attached to the file, or to use more intuitive approaches that simply require the name and the type of the file.

The general interaction flow for file management is as follows:

- Once the application is submitted, the shared files are staged into the Aneka Cloud.
- If the file is local it will be searched into the directory location identified by the property *Configuration.Workspace*; if the file is remote, the specific configuration settings mapped by the *FileData.StorageBucketId* property will be used to access the remote server and stage in the file.
- If there is any failure in staging input files, the application will be terminated with an error.
- For each of the tasks belonging to the application, the corresponding input files are staged into the Aneka Cloud, as is done for shared files.
- Once the task is dispatched for execution to a remote node, the runtime will transfer all the shared files of the application and the input files of the task into the working directory of the task and eventually get renamed if the *FileData.VirtualPath* property is not null.
- After the execution of the task, the runtime will look for all the files that have been added into the *WorkUnit.OutputFiles* collection. If not null, the value of the *FileData.VirtualPath* property will be used to locate the files; otherwise, the *FileData.FileName* property will be the reference. All the files that do not contain the *FileAttributes.Optional* attribute need to be present; otherwise, the execution of the task is classified as a failure.
- Despite the successful execution or the failure of a task, the runtime tries to collect and move to their respective destinations all the files that are found. Files that contain the *FileAttributes.Local* attribute are moved to the local machine from where the application is saved and stored in the directory location identified by the property *Configuration.Workspace*. Files that have a *StorageBucketId* property set will be staged out to the corresponding remote server.

The infrastructure for file management provides a transparent and extensible service for the movement of data. The architecture for file management is based on the concept of factories and storage buckets. A *factory*, namely *IFileTransferFactory*, is a component that is used to abstract the creation of client and server components for file transfer so that the entire architecture can work with interfaces rather than specific implementations. *Storage buckets* are collections of string properties that are used to specialize these components through configuration files. Storage buckets are specified by the user, either by the configuration file or by programmatically adding this information to the configuration object, before submitting the application. Factories are used by the *StorageService* to pull in remote input and shared files and to pull out remote output files.

Listing 7.8 shows a sample configuration file containing the settings required to access the remote files through the FTP and S3 protocols. Within the *<Groups>* tag, there is a specific group named *StorageBuckets*; this group maintains the configuration settings for each storage bucket that needs to be used in for file transfer. Each *<Group>* tag represents a storage bucket and the *name* property contains the values referenced in the *FileData.StorageBucketId* property. The content of each of these groups is specific to the type of storage bucket used, which is identified by the *Scheme* property. These values are used by the specific implementations for the FTP and S3 protocols to access the remote servers and transfer the files.

```
// get an instance of the Configuration class from file
Configuration conf = Configuration.GetConfiguration("conf.xml");
AnekaApplication<Task,TaskManager>app =new AnekaApplication<Task,TaskManager>(conf);

// attach shared files with different methods by using the FileData class and directly
// using the API provided by the AnekaApplication class

// create a local shared file whose local and remote name is "pi.tab"
FileDatapiTab = newFileData("pi.tab",FileDataType.Shared);
app.AddSharedFile(piTab);
// once the file is added to the collection of shared files, its OwnerId property
// references app.Id

// create a remote shared file by specifying the attributes whose name is "pi.dat"
FileDatapiDat = newFileData("pi.dat",FileDataType.Shared, FileAttributes.None);
// the StorageBucketId property points a specific configuration section that is
// used to store the information for retrieving the file from the remote server
piDat.StorageBucketId = "FTPStore";
app.AddSharedFile(piDat);
// once the file is added to the collection of shared files, its OwnerId property
// references app.Id

// adds a local shared file
app.AddSharedFile("pi.xml");

for(int i=0; i<400; i++)
{
   GaussTask gauss = new GaussTask();
   gauss.X = i;
   AnekaTask task = newAnekaTask(gauss);

   // adds a local input file for the current task whose name is "<i>.txt"
   // where <i> is the value of the loop variable
   FileData input = new FileData(string.Format("{0}.txt", i);
   FileDataType.Input, FileAttributes.Local);
   // once transferred to the remote node, the file will have the name
   // "input.txt". Since tasks are executed in separate directories there
   // will be no name clashing
   input.VirtualPath ="input.txt";
   task.AddFile(input);
   // once the file is added to the task, it will be stored in the InputFiles
   // collection and its OwnerId property will referenced task.Id

   // adds anoutput file for the current task whose name is "out.txt" that will
   // be stored on S3
```

LISTING 7.7

File dependencies management.

```
    FileData output =new FileData("out.txt", FileDataType.Input, FileAttributes.None);
    // once transferred to the remote server, the file will have the name
    // "<i>.out"where <i> is the value of the loop variable. In this way we
    // easily avoid name clashing while storing output files into a single
    // directory
    output.VirtualPath =string.Format("{0}.out", i);
    output.StorageBucketId = "S3Store";
    task.AddFile(output);
    // once the file is added to the task, it will be stored in the InputFiles
    // collection and its OwnerId property will referenced task.Id

    // adds a localoutput file for the current task whose name is "trace.log".
    // The file is optional, this means that if after the execution of the task the file

    // is not present no exception or task failure will be risen.
    FileData trace =new FileData("trace.log", FileDataType.Input,
    FileAttributes.Local | FileAttributes.Optional);
    // once transferred to the local machine, the file will have the name
    // "<i>.log"where <i> is the value of the loop variable. In this way we
    // easily avoid name clashing while storing output files into a single
    // directory
    trace.VirtualPath =string.Format("{0}.log", i);
    task.AddFile(trace);
    // once the file is added to the task, it will be stored in the InputFiles
    // collection and its OwnerId property will referenced task.Id

    // add the task to the bag of work units to submit
    app.AddWorkunit(task);
}

// submit the entire bag, files will be moved automatically by the Aneka APIs
app.SubmitExecution();
```

LISTING 7.7

(Continued)

7.3.2.4 Task libraries

Aneka provides a set of ready-to-use tasks for performing the most basic operations for remote file management. These tasks are part of the *Aneka.Tasks.BaseTasks* namespace, which is part of the *Aneka.Tasks.dll* library. The following operations are implemented:

- *File copy.* The *LocalCopyTask* performs the copy of a file on the remote node; it takes a file as input and produces a copy of it under a different name or path.
- *Legacy application execution.* The *ExecuteTask* allows executing external and legacy applications by using the *System.Diagnostics.Process* class. It requires the location of the executable file to run, and it is also possible to specify command-line parameters. *ExecuteTask* also collects the standard error and standard output produced by the execution of the application.

```
<?xmlversion="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>
```

LISTING 7.8

Aneka application configuration file.

- *Substitute operation.* The *SubstituteTask* performs a search-and-replace operation within a given file by saving the resulting file under a different name. It is possible to specify a collection of string-based name-value pairs representing the strings to search together with their corresponding replacements.
- *File deletion.* The *DeleteTask* deletes a file that is accessible through the file system on the remote node.
- *Timed delay.* The *WaitTask* introduces a timed delay. This task can be used in several scenarios; for example, it can be used for profiling or simply for simulation of the execution. In addition, it can also be used to introduce a pause between the execution of two applications if needed.

- *Task composition.* The *CompositeTask* implements the composite pattern[3] and allows expressing a task as a composition of multiple tasks that are executed in sequence. This task is very useful to perform complex tasks involving the combination of operations implemented in other tasks.

The base task library does not provide any support for data transfer since this operation is automatically performed by the infrastructure when needed. Besides these simple tasks, the Aneka API allows for the creation of any user-defined task by simply implementing the *ITask* interface and supporting object serialization.

7.3.2.5 Web services integration

Aneka provides integration with other technologies and applications by means of Web services, which allow some of the services hosted in the Aneka Cloud to be accessible in platform-independent fashion. Among these, the task submission Web service allows third-party applications to submit tasks as they happen in traditional computing grids.

The task submission Web service is an additional component that can be deployed in any ASP.NET Web server and that exposes a simple interface for job submission, which is compliant with the Aneka Application Model. The task Web service provides an interface that is more compliant with the traditional way fostered by grid computing. Therefore, the new concept of the term *job*, which is a collection of predefined tasks, is introduced. The reference scenario for Web-based submission is depicted in Figure 7.9. Users create a distributed application instance on the cloud and, within the context of this application, they can submit jobs querying the status of the application or a single job. It is up to the users to then terminate the application when all the jobs are completed or to abort it if there is no need to complete job execution.

Jobs can be created by putting together the tasks defined in the basic task library. Operations supported through the Web service interface are the following:

- Local file copy on the remote node
- File deletion
- Legacy application execution through the common shell services
- Parameter substitution

It is also possible specify input and output files for each job. The only restriction in this case is that both input and output files need to reside in remote FTP servers. This enables Aneka to automatically stage the files from these servers without user intervention. Figure 7.10 gives detailed information about the object model exposed through the Web service for job submission.

Traditional grid technologies such as the Gridbus Broker [15] and the Workflow Engine [70] can make use of a task Web service to submit their tasks for execution on Cloud nodes managed by Aneka.

[3]In software engineering, the composite pattern is a software design pattern that allows expressing a combination of components as a single component. The advantage of using such a pattern resides in creating a software infrastructure that allows forwarding the execution of an operation to a group of objects by treating it as single unit and in a completely transparent manner. Reference: E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Software Design*, Addison-Wesley, 1995, ISBN: 0201633612.

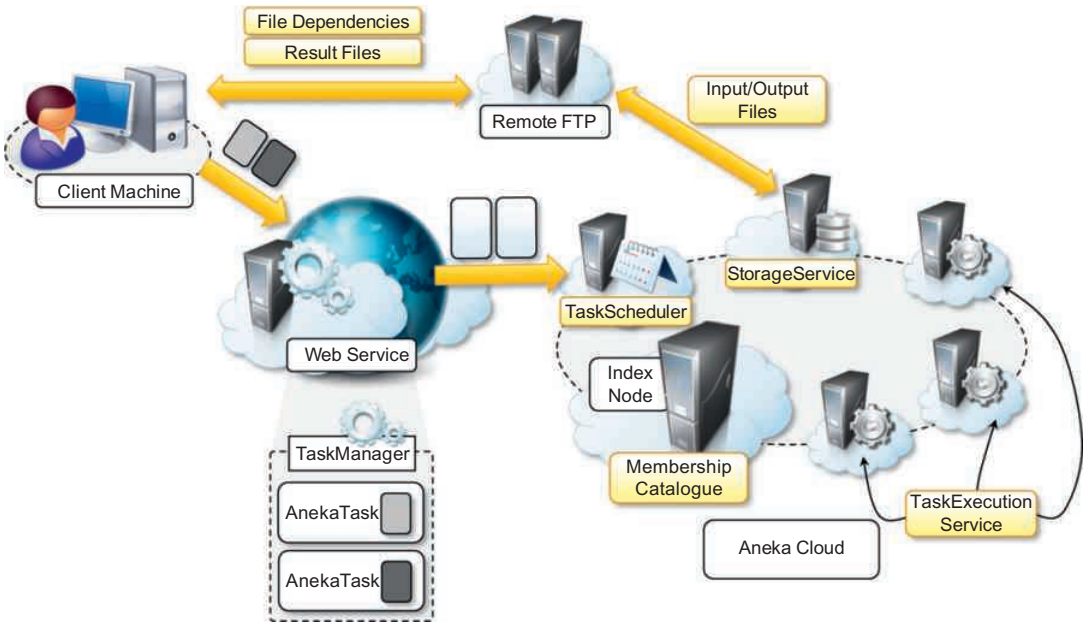


FIGURE 7.9

Web service submission scenario.

7.3.3 Developing a parameter sweep application

Aneka integrates support for parameter-sweeping applications on top of the task model by means of a collection of client components that allow developers to quickly prototype applications through either programming APIs or graphical user interfaces (GUIs). The set of abstractions and tools supporting the development of parameter sweep applications constitutes the *Parameter Sweep Model (PSM)*.

The PSM is organized into several namespaces under the common root *Aneka.PSM*. More precisely:

- *Aneka.PSM.Core (Aneka.PSM.Core.dll)* contains the base classes for defining a template task and the client components managing the generation of tasks, given the set of parameters.
- *Aneka.PSM.Workbench (Aneka.PSM.Workbench.exe)* and *Aneka.PSM.Wizard (Aneka.PSM.Wizard.dll)* contain the user interface support for designing and monitoring parameter sweep applications. Mostly they contain the classes and components required by the *Design Explorer*, which is the main GUI for developing parameter sweep applications.
- *Aneka.PSM.Console (Aneka.PSM.Console.exe)* contains the components and classes supporting the execution of parameter sweep applications in console mode.

These namespaces define the support for developing and controlling parameter sweep applications on top of Aneka.

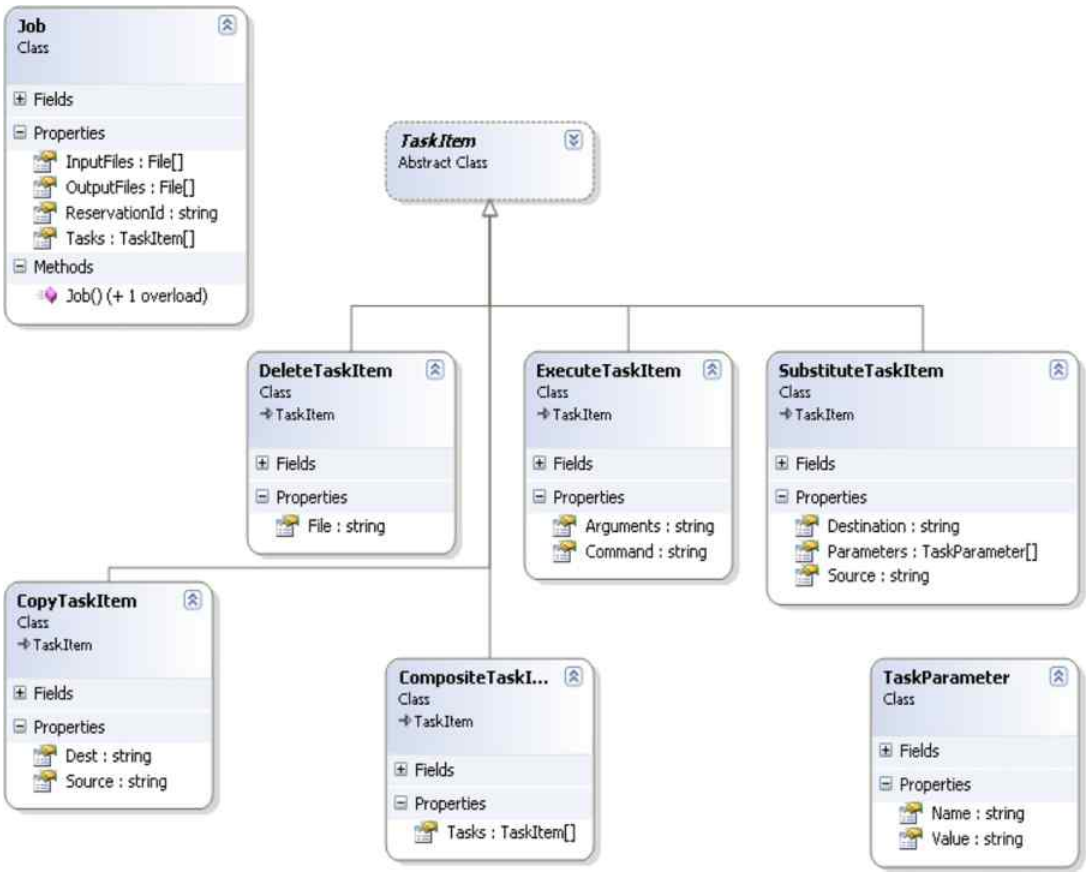


FIGURE 7.10

Job object model.

7.3.3.1 Object model

The fundamental elements of the Parameter Sweep Model are defined in the *Aneka.PSM.Core* namespace. This model introduces the concept of *job (Aneka.PSM.Core.PSMJobInfo)*, which identifies a parameter sweep application. A job comprises file dependencies and parameter definitions, together with their admissible domains, and the definition of the template task. Figure 7.11 shows the most relevant components of the object model.

The root component for application design is the *PSMJobInfo* class, which contains information about shared files and input and output files (*PSMFileInfo*). In accordance with the Aneka Application Model, shared files are common to all the instances of the template task, whereas input

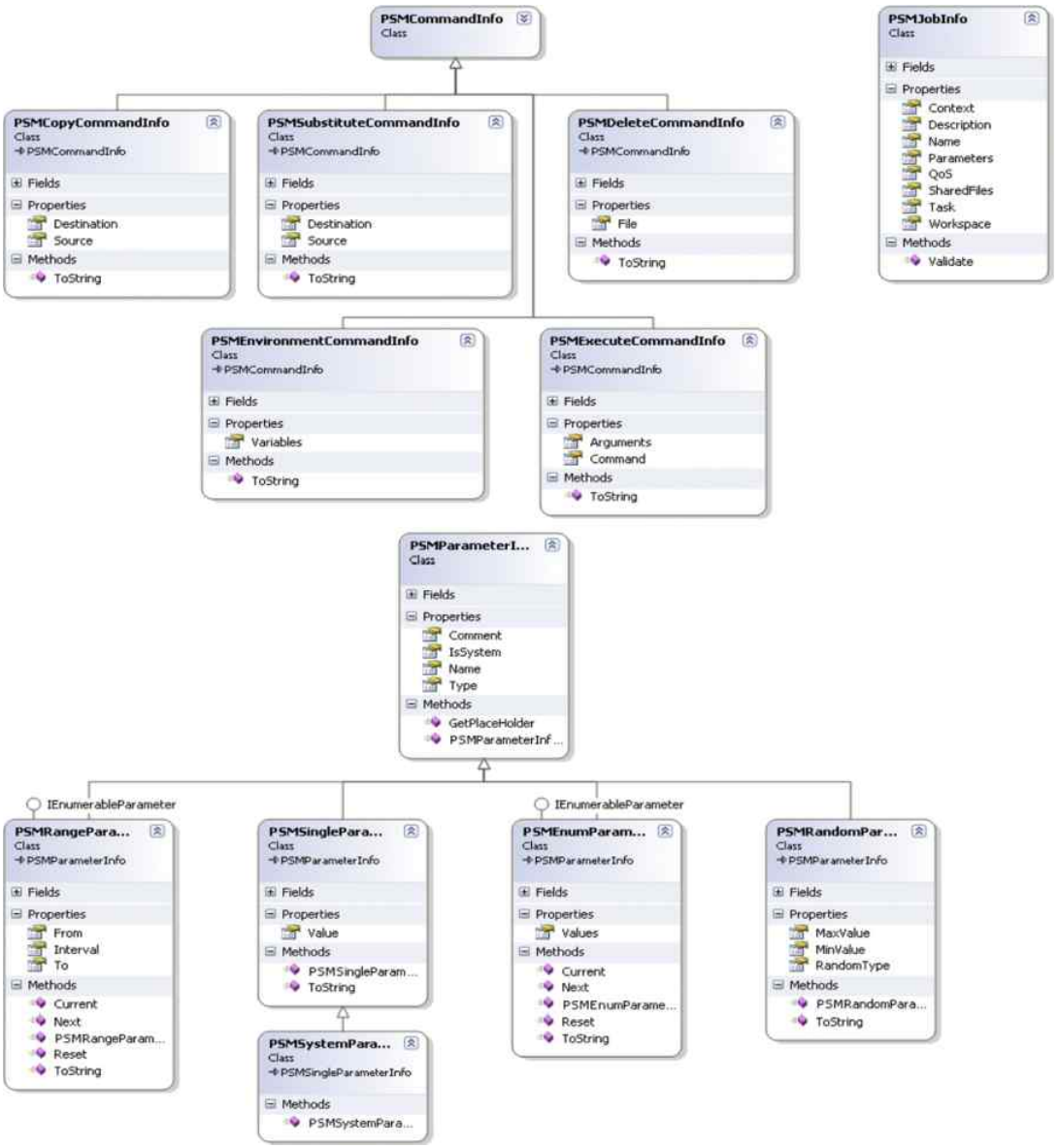


FIGURE 7.11

PSM object model (relevant classes).

and output files are specific to a given task instance. Therefore, these files can be expressed as a function of the parameters. Currently, it is possible to specify five different types of parameters:

- *Constant parameter* (*PSMSingleParameterInfo*). This parameter identifies a specific value that is set at design time and will not change during the execution of the application.
- *Range parameter* (*PSMRangeParameterInfo*). This parameter allows defining a range of allowed values, which might be integer or real. The parameter identifies a domain composed of discrete values and requires the specification of a lower bound, an upper bound, and a step for the generation of all the admissible values.
- *Random parameter* (*PSMRandomParameterInfo*). This parameter allows the generation of a random value in between a given range defined by a lower and an upper bound. The value generated is real.
- *Enumeration parameter* (*PSMEnumParameterInfo*). This parameter allows for specifying a discrete set of values of any type. It is useful to specify discrete sets that are not based on numeric values.
- *System parameter* (*PSMSystemParameterInfo*). This parameter allows for mapping a specific value that will be substituted at runtime while the task instance is executed on the remote node.

Other than these parameters, the object model reserves special parameters that are used to identify specific values of the PSM object model, such as the task identifier and other data. Parameters have access to the execution environment by means of an execution context (*PSMContext*) that is responsible for providing default and runtime values. The task template is defined as a collection of commands (*PSMCommandInfo*), which replicate and extend the features available in the base task library. The available commands for composing the task template perform the following operations:

- Local file copy on the remote node (*PSMCopyCommandInfo*)
- Remote file deletion (*PSMDeleteCommandInfo*)
- Execution of programs through the shell (*PSMExecuteCommandInfo*)
- Environment variable setting on the remote node (*PSMEnvironmentCommandInfo*)
- String pattern replacement within files (*PSMSubstituteCommandInfo*)

By following the same approach described for the creation of tasks, it is possible to define the task template by composing these basic blocks. All the properties exposed by these commands can include the previously defined parameters, the values of which will be provided during the generation of the task instances.

A parameter sweep application is executed by means of a job manager (*IJobManager*), which interfaces the developer with the underlying APIs of the task model. Figure 7.12 shows the relationships among the PSM APIs, with a specific reference to the job manager, and the task model APIs.

Through the *IJobManager* interface it is possible to specify user credentials and configuration for interacting with the Aneka middleware. The implementation of *IJobManager* will then create a corresponding Aneka application instance and leverage the task model API to submit all the task instances generated from the template task. The interface also exposes facilities for controlling and monitoring the execution of the parameter sweep application as well as support for registering the statistics about the application.

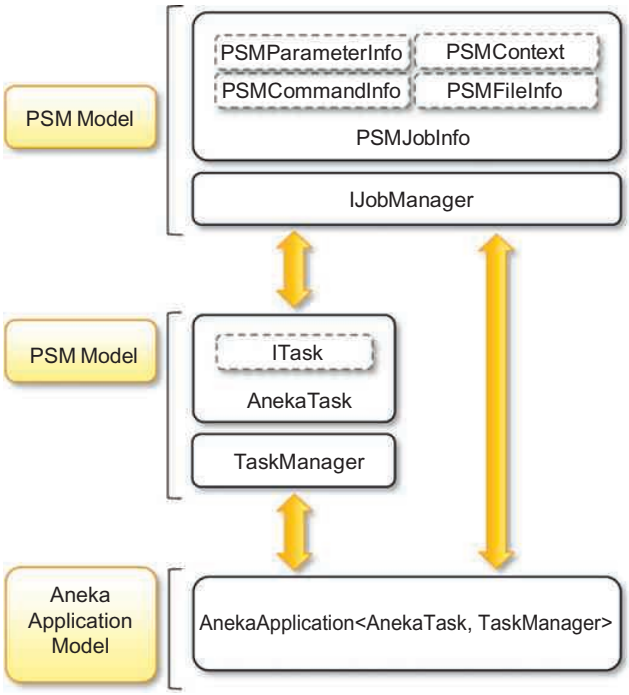


FIGURE 7.12

Parameter sweep model APIs.

7.3.3.2 Development and monitoring tools

The core libraries allow developers to directly program parameter sweep applications and embed them into other applications. Additional tools simplify design and development of parameter sweep applications by providing support for visual design of the applications and interactive and noninteractive application execution. These tools are the *Aneka Design Explorer* and the *Aneka PSM Console*.

The *Aneka Design Explorer* is an integrated visual environment for quickly prototyping parameter sweep applications, executing them, and monitoring their status. It provides a simple wizard that helps the user visually define any aspect of parameter sweep applications, such as file dependencies and result files, parameters, and template tasks. The environment also provides a collection of components that help users monitor application execution, aggregate statistics about application execution, gain detailed task transition monitoring, and gain extensive access to application logs.

The *Aneka PSM Console* is a command-line utility designed to run parameter sweep applications in noninteractive mode. The console offers a simplified interface for running applications with essential features for monitoring their execution. With respect to the Design Explorer, the console offers less support for keeping and visualizing aggregate statistics, but it exposes the same data in a more simplified textual form.

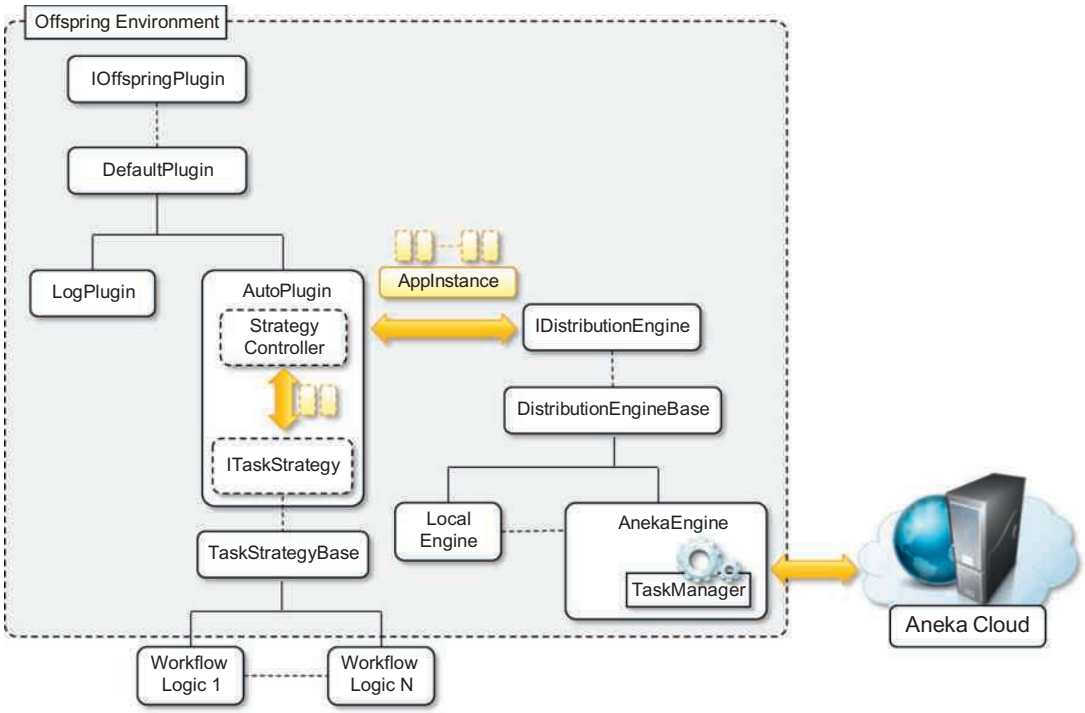


FIGURE 7.13

Offspring architecture.

7.3.4 Managing workflows

Support for workflow in Aneka is not native but is obtained with plug-ins that allow client-based workflow managers to submit tasks to Aneka. Currently, two different workflow managers can leverage Aneka for task execution: the *Workflow Engine* [70] and *Offspring* [71]. The former leverages the task submission Web service exposed by Aneka; the latter directly interacts with the Aneka programming APIs. The Workflow Engine plug-in for Aneka constitutes an example of the integration capabilities offered by the framework, which allows client applications developed with any technology and language to leverage Aneka for task execution. The integration developed for Offspring constitutes another example of how it is possible to construct another programming model on top of the existing APIs available in the framework. Therefore, we discuss this solution in more detail.

Figure 7.13 describes the Offspring architecture. The system is composed of two types of components: plug-ins and a distribution engine. Plug-ins are used to enrich the environment of features; the distribution engine represents access to the distributed computing infrastructure leveraged for task execution. Among the available plug-ins, *AutoPlugin* provides facilities for the definition of workflows in terms of strategies. A *strategy* generates the tasks that are submitted for execution and defines the logic, in terms of sequencing, coordination, and dependencies, used to submit the task through the engine. A specific component, the *StrategyController*, decouples the strategies

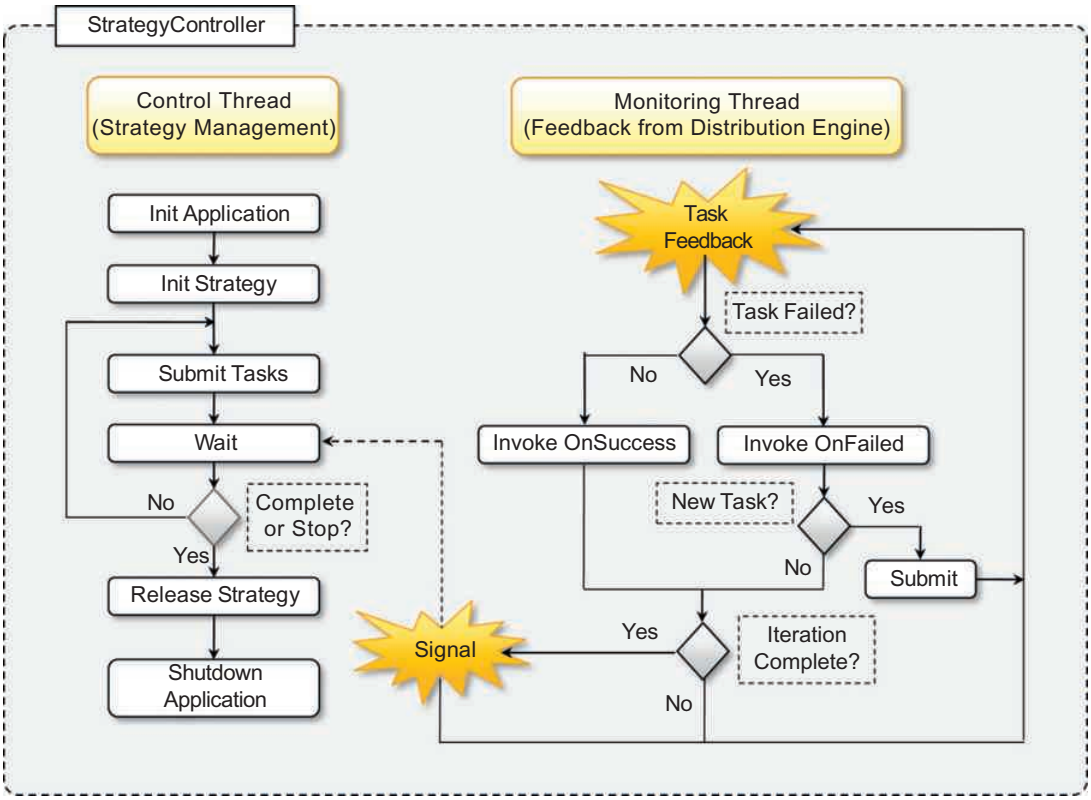


FIGURE 7.14

Workflow coordination.

from the distribution engine; therefore, strategies can be defined independently of the specific middleware used. The connection with Aneka is realized through the *AnekaEngine*, which implements the operations of *IDistributionEngine* for the Aneka middleware and relies on the services exposed by the task model programming APIs.

The system allows for the execution of a dynamic workflow, the structure of which is defined as the workflow executes. Two different types of tasks can be defined: native tasks and legacy tasks. *Native tasks* are completely implemented in managed code. *Legacy tasks* manage file dependencies and wrap all the data necessary for the execution of legacy programs on a remote node. Furthermore, a strategy may define shared file dependencies that are necessary to all the tasks generated by the workflow. The dependencies among tasks are implicitly defined by the execution of the strategy by the *StrategyController* and the events fired by the distributed engine.

Figure 7.14 describes the interactions among these components. Two main execution threads control the execution of a strategy. A *control thread* manages the execution of the strategy, whereas a *monitoring thread* collects the feedback from the distribution engine and allows for the dynamic

reaction of the strategy to the execution of previously submitted tasks. From the workflow developer's point of view, the logic is quite simple. The execution of a strategy is composed of three macro steps: setup, execution, and finalization. The first step involves the setup of the strategy and the application mapping it. Correspondingly, the finalization step is in charge of releasing all the internal resources allocated by the strategy and shutting down the application. The core of the workflow execution resides in the execution step, which is broken down into a set of iterations. During each of the iterations a collection of tasks is submitted; these tasks do not have dependencies from each other and can be executed in parallel. As soon as a task completes or fails, the strategy is queried to see whether a new set of tasks needs to be executed. In this way, dependencies among tasks are implemented. If there are more tasks to be executed, they are submitted and the controller waits for feedback from the engine; otherwise, an iteration of the strategy is completed. At the end of each iteration, the controller checks to see whether the strategy has completed the execution, and in this case, the finalization step is performed.

The *AnekaEngine* creates an instance of the *AnekaApplication* class for each execution of a strategy and configures the template class with a specific implementation of the *TaskManager*, which overrides the behavior implemented for file management and optimizes the staging of output files. To support the implementation of workflows without any dependency from the distribution engine, the application configuration settings are controlled by the distribution engine and shared among all the strategies executed through the engine.

SUMMARY

This chapter introduced the concept of task-based programming and provided an overview of the technologies supporting the development of distributed applications based on the concept of tasks. Task-based programming constitutes the most intuitive approach to distributing the computation of an application over a set of nodes. The main abstraction of task-based programming is the concept of a *task*, which represents a group of operations that can be isolated and executed as a single unit. A task can be a simple program that is executed through the shell or a more complex piece of code requiring a specific runtime environment to execute. Quite often, tasks require input files for their execution and produce output files as a result. According to this model, an application is expressed as a collection of tasks; the way these tasks are interrelated and their specific nature and characteristics differentiate the various models that are an expression of task-based programming.

Traditionally, the task-based programming model has been successfully used in the development of distributed applications in many areas. We identified three major computing categories in which the task model can be utilized. *High-performance computing (HPC)* refers to the use of distributed computing facilities for solving problems that need large computing power. Common HPC applications feature a large collection of compute-intensive tasks, the duration of which is relatively short. *High-throughput computing (HTC)* identifies scenarios in which distributed computing facilities are used to support the execution of applications that need large computing power for a long period of time. Tasks may not be numerous, but they have a long duration, and infrastructure reliability becomes fundamental. *Many-task computing (MTC)* is the latest emergent trend and identifies a heterogeneous set of applications and requirements for applications, which fills the gap between HPC and HTC.

We have briefly reviewed common models related to task programming. *Embarrassingly parallel* applications are composed of a collection of tasks that do not relate to each other, can be executed in any order, and do not require co-allocation. *Parameter sweep* applications are a special instance of the embarrassingly parallel model. They are characterized by a collection of independent tasks that are automatically generated from a template task by varying the combination of parameter values. In this case, the executed task is the same in terms of computation logic, but it operates on different data. Therefore, a parameter sweep application can also be considered an expression of the *single program, multiple data (SPMD)* model. *MPI* applications are characterized by a collection of tasks that need to be executed all together and that exchange data by message passing. Even though the program executed by an MPI application tasks might be the same, it is quite common to provide an implementation logic that differentiates the behavior of each task according to its rank. *Workflow* applications are characterized by a collection of tasks for which the dependencies can be expressed in terms of a directed acyclic graph. Dependencies are mostly represented by files, which are produced as output of a specific task, and are required for the computation of the dependent tasks. The nature of the tasks and the kind of computation performed by each task differ generally.

We introduced the task model and the services implemented in Aneka that support task-based programming as a practical example of a framework that enables the development and execution of distributed applications based on tasks. The task model comprises a set of services (directory, scheduling, execution, and storage) of which the coordination constitutes runtime support for the execution of embarrassingly parallel applications. The fundamental features of the task model in terms of task definition, submission, execution, and file dependencies management was demonstrated with a practical example. On top of this infrastructure, client-side components and integration with other technologies allow providers to support parameter sweep and workflow applications. Parameter sweep applications are realized through the Parameter Sweep Model (PSM), which is characterized by a collection of client-side components that provide different, and more suitable, interfaces for this kind of application. Workflow applications are not natively supported by Aneka, but integration with other technologies allows us to leverage Aneka for workflow execution. For example, a plug-in using the Aneka task submission Web service allows the Workflow Engine to use Aneka as a back-end for workflow execution. The Aneka distribution engine implemented in Offspring provides another example of how it is possible to quickly prototype another programming model (in this case, a workflow-based model) by leveraging the base APIs of the task model.

Review questions

1. What is a task? How does task computing relate to distributed computing?
2. List and explain the computing categories that relate to task computing.
3. What are the main functionalities of a framework that supports task computing?
4. List some of the most popular frameworks for task computing.
5. What does the term *bag of tasks* mean?
6. Give an example of a parameter sweep application.
7. What is MPI? What are its main characteristics?

8. What is a workflow? What are the additional properties of this application model with respect to an embarrassingly parallel application?
9. Describe the reference model of a workflow management system.
10. How does Aneka support task computing?
11. What are the main components of the Task Programming Model?
12. Discuss the differences between ITask and AnekaTask.
13. Discuss the differences between static and dynamic task submission.
14. Discuss the facilities and the general architecture provided by Aneka for movement of data for task-based applications.
15. How it is possible to run a legacy application using the Task Programming Model?
16. Does Aneka provide any feature for leveraging the Task Programming Model from other technologies and platforms?
17. Using the Task Programming Model, design and implement a simple application that performs the discrete computation of the integral according to the method proposed by Riemann[4] of a given function over a specified interval.
18. What are the features provided by Aneka for the execution of parameter sweep applications?
19. Does Aneka provide native support for the execution of workflows?
20. By taking as a reference the Montage workflow described in Figure 7.6, design a sketch of the control flow of an Offspring strategy that can be used to execute a workflow on Aneka.

[4] http://en.wikipedia.org/wiki/Riemann_inegral.