Module 3

Concurrent Computing

Throughput computing focuses on delivering high volumes of computation in the form of transactions. Advances in hardware technologies led to the creation of multi core systems, which have made possible the delivery of high-throughput computations

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.

6.1 Introducing parallelism for single-machine computation

Parallelism has been a technique for improving the performance of computers since the early 1960s.

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.

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

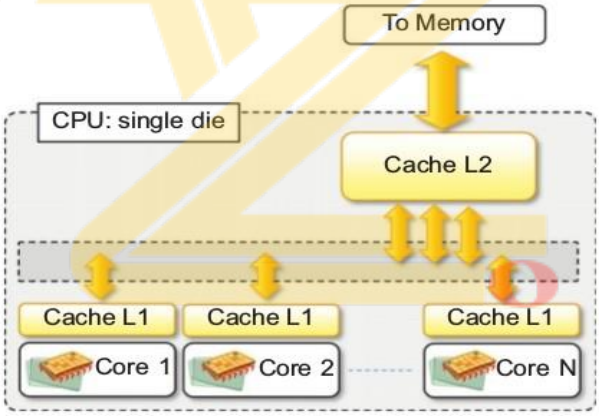

FIGURE 6.1

Multicore processor.

processing units to share the computation load.

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.

Multiprocessing is just one technique that can be used to achieve parallelism, and it does that by leveraging parallel hardware architectures.

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.

6.2 Programming applications with threads

Modern applications perform multiple operations at the same time.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.

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.

Operating systems that support multithreading identify threads as the minimal building blocks for expressing running code.

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.

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 replacing it with the information related to another process is known as a **context switch**.

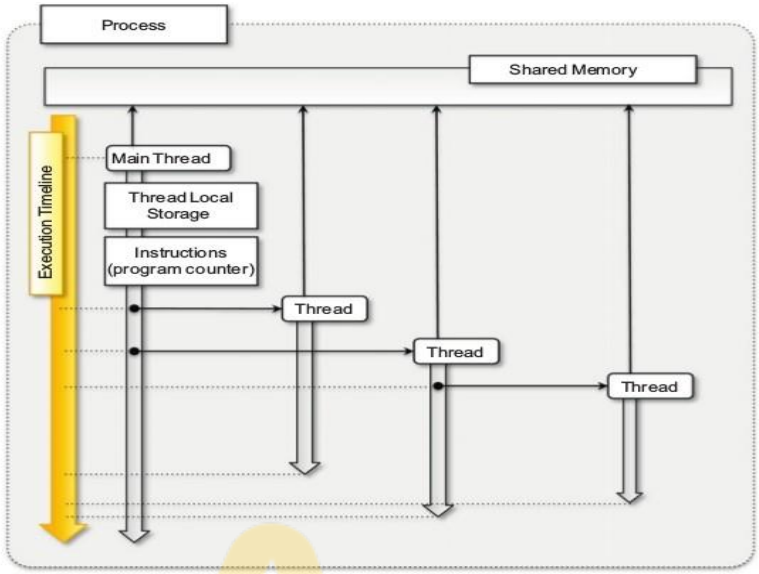

FIGURE 6.2
The relationship between processes and threads.

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.

Thread APIs

1 POSIX Threads

2 Threading support in java and .NET

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.

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.

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. 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.

- Both Java and .NET provide class Thread with the common operations 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.
- Suspend and resume are used to programmatically pause and then continue the execution of a thread. Sleep operation allows pausing the execution of a thread for a predefined period of time.
- Join operation that makes one thread wait until another thread is completed. Waiting states can be interrupted by using the interrupt operation.

6.2.2 Techniques for parallel computation with threads

Developing parallel applications requires an understanding of the problem and its logical structure. Decomposition is a useful technique that aids in understanding whether a problem is divided into components (or tasks) that can be executed concurrently. it allows the breaking down into independent units of work that can be executed concurrently with the support provided by threads.

1 Domain decomposition

2 Functional decomposition

3 Computation vs. Communication

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. 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.

Embarrassingly parallel problems constitute the easiest case for parallelization because there is no need to synchronize different threads that do not share any data. 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. If the values of all the iterations are dependent on some of the values obtained in the previous iterations, the problem is said to be **inherently sequential**. **Figure 6.3** provides a schematic representation of the decomposition of embarrassingly parallel and inherently sequential problems.

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

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

for each of the resulting matrix elements is the following:

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

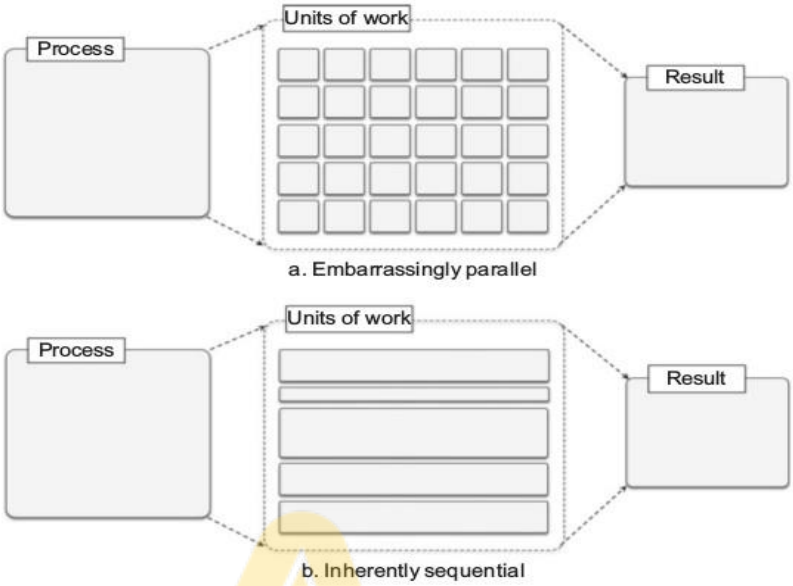

FIGURE 6.3
Domain decomposition techniques.

Figure 6.4 provides an overview of how a matrix product can be performed.

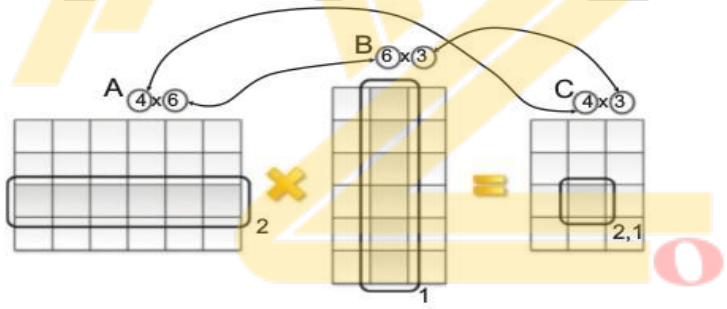

FIGURE 6.4
A matrix product.

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.

The .NET framework provides the System.Threading.Thread class that can be configured with a function pointer, also known as a delegate, to execute asynchronously. This class will also define the method for performing the actual computation.

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.

Figure 6.5 provides a pictorial view of how decomposition operates and allows parallelization.

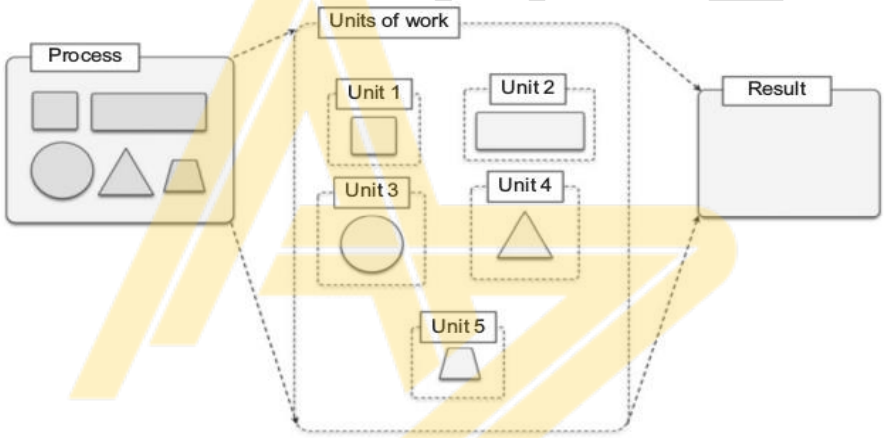

FIGURE 6.5

Functional decomposition.

The 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 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:

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

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.

Computation vs. Communication

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.

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, submitted for execution by leveraging clouds.

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.

This model provides the best advantage in the case of **embarrassingly parallel applications**.

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 by 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.

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.

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.

1 Interface compatibility

2 Thread life cycle

3 Thread synchronization

4 Thread priorities

5 Type serialization

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.

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*

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.

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.

2 Thread life cycle

Aneka Thread life cycle is different from the life cycle of local threads. It is not possible to directly map the state values of a local thread to 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.

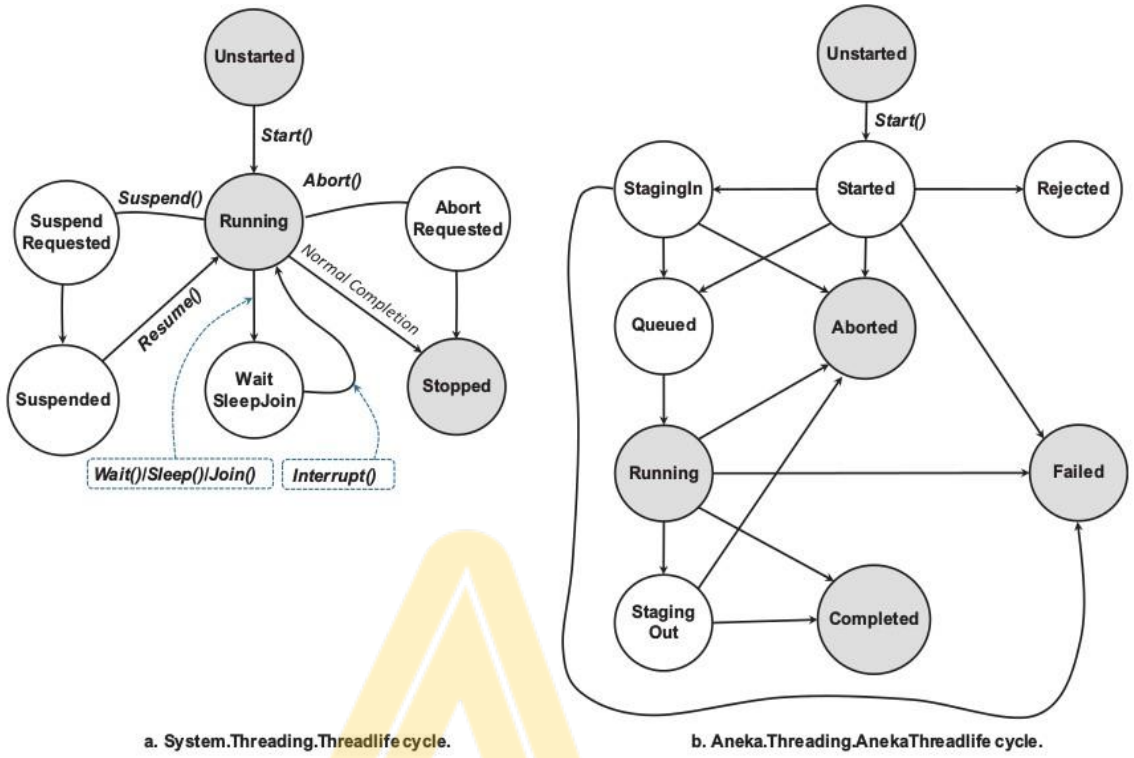

FIGURE 6.6

Thread life-cycle comparison.

In local threads most of the state transitions are controlled by the developer, who actually triggers the state transition by invoking methods. Whereas in Aneka threads, many of the state transitions are controlled by the middleware.

Aneka threads support file staging and they are scheduled by the middleware, which can queue them for a considerable amount of time.

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.

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.

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.

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.

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 except join operation.

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.

Aneka does not support thread priorities, 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.

5 Type serialization

Serialization is the process of converting an object into a stream of bytes to store the object or transmit it to memory, a database, or a file. Main purpose is to save the state of an object to recreate it when needed. The reverse process is called deserialization.

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.

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.

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.

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

TheThread 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.

The Aneka APIs support 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>

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.