

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>

MODULE – III**High-Throughput Computing – Task Programming (Chapter 7)**

Task computing is a wide area of distributed system programming encompassing several different models of architecting distributed applications,

A task represents a program, which 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.

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**7.1.1 Characterizing a task****7.1.2 Computing categories**

- 1 High-performance computing
- 2 High-throughput computing
- 3 Many-task computing

7.1.3 Frameworks for task computing

1. Condor
2. Globus Toolkit
3. Sun Grid Engine (SGE)
4. BOINC
5. Nimrod/G

7.2 Task-based application models**7.2.1 Embarrassingly parallel applications****7.2.2 Parameter sweep applications****7.2.3 MPI applications****7.2.4 Workflow applications with task dependencies**

- 1 What is a workflow?
- 2 Workflow technologies
 1. Kepler,
 2. DAGMan,
 3. Cloudbus Workflow Management System, and
 4. Offspring.

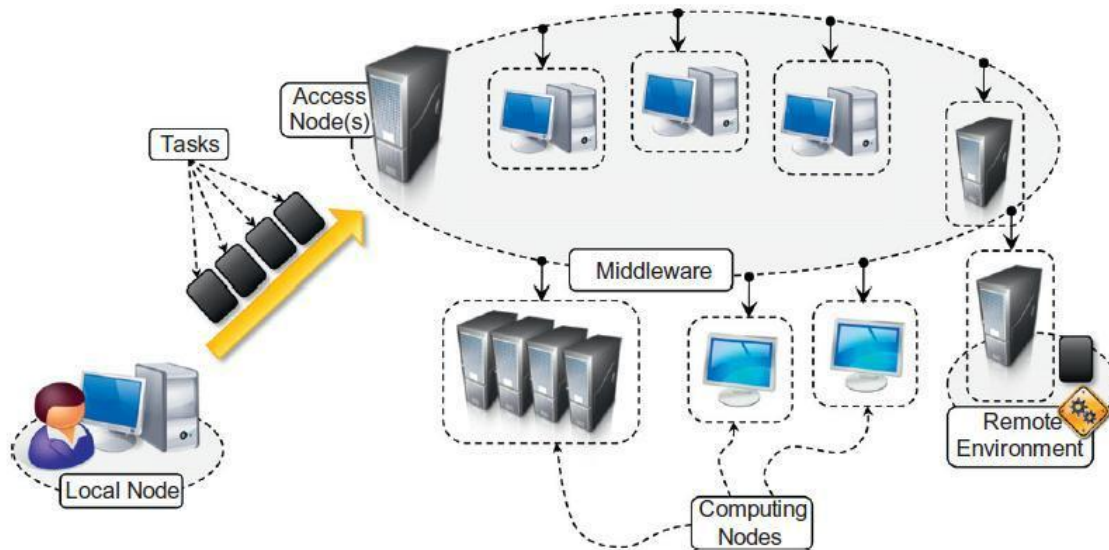
7.3 Aneka task-based programming**7.3.1 Task programming model****7.3.2 Developing applications with the task model****7.3.3 Developing a parameter sweep application****7.3.4 Managing workflows****7.1 Task computing**

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 run time environment.

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.

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**.

**FIGURE 7.1**

Task computing scenario.

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

A user 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.

Task submission is 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.

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.

7.1.1 Characterizing a task

A task represents a component of an application that can be logically isolated and executed separately.

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/C11/.NET class) that executes within the context of a specific runtime environment. A task is characterized by input files, executable code (programs, shell scripts, etc.), and output files.

The runtime environment in which tasks execute is the operating system or an equivalent sandboxed environment.

A task may also need specific software appliances on the remote execution nodes.

7.1.2 Computing categories

These categories 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:

1 High-performance computing

2 High-throughput computing

3 Many-task computing

1 High-performance computing

High-performance computing (HPC) is the use of distributed computing facilities for solving problems that need large computing power.

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.

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.

Ex: supercomputers and clusters are specifically designed to support HPC applications that are developed to solve “Grand Challenge” problems in science and engineering.

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.

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.

Ex: 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.

3 Many-task computing

MTC denotes high-performance computations comprising multiple distinct activities coupled via file system operations.

MTC is the heterogeneity of tasks that might be of different nature: Tasks may be small or large, single-processor or multiprocessor, compute-intensive or data-intensive, static or dynamic, homogeneous or heterogeneous.

MTC applications includes loosely coupled applications that are communication-intensive but not naturally expressed using the message-passing interface.

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.

7.1.3 Frameworks for task computing

Some popular software systems that support the task-computing framework are:

1. Condor
2. Globus Toolkit
3. Sun Grid Engine (SGE)
4. BOINC
5. Nimrod/G

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.

1. Condor

Condor is the most widely used and long-lived middleware for managing clusters, idle workstations, and a collection of clusters.

Condor supports 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.

Condor-G is a version of Condor that supports integration with grid computing resources, such as those managed by Globus.

2. Globus Toolkit

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.

The toolkit features software services, libraries, and tools for resource monitoring, discovery, and management as well as security and file management.

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.

3. Sun Grid Engine (SGE)

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.

4. BOINC

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 supports job checkpointing and duplication.

BOINC is composed of two main components: the BOINC server and the BOINC client.

The BOINC server is the central node that keeps track of all the available resources and scheduling jobs.

The BOINC client is the software component that is deployed on desktop machines and that creates the BOINC execution environment for job submission.

BOINC systems can be easily set up to provide more stable support for job execution by creating computing grids with dedicated machines.

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.

5. Nimrod/G

Tool for automated modeling and execution of parameter sweep applications over global computational grids.

It provides a simple declarative parametric modeling language for expressing parametric experiments.

It uses novel resource management and scheduling algorithms based on economic principles.

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.

It has been used for a very wide range of applications over the years, ranging from quantum chemistry to policy and environmental impact.

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.

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.

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.

Scheduling these applications is simplified and 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: image and video rendering, evolutionary optimization, and model forecasting.

In image and video rendering the task is represented by the rendering of a pixel or a frame, respectively.

For evolutionary optimization meta heuristics, 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.

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.

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 parameter set identifies the combination of variables whose assignments specialize the template task into a specific instance.

Any distributed computing framework that provides support for embarrassingly parallel applications can also support the execution of parameter sweep applications.

The only difference is that the tasks that will be executed are generated by iterating over all the possible and admissible combinations of parameters.

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.

A plethora of applications fall into this category. Scientific computing domain: evolutionary optimization algorithms, weather-forecasting models, computational fluid dynamics applications, Monte Carlo methods. 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.

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.

```

individuals 5 {100, 200, 300, 500, 1000}
generations 5 {50, 100, 200, 400}
foreach indiv in individuals do
  foreach generation in generations do
    task = generate_task(indiv, generation)
    submit_task(task)

```

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 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. Frameworks that natively support the execution of parameter sweep applications provide a set of useful commands for manipulating or operating on files. 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.

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.

```

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.

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.

```

<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.

The file is an XML document containing several sections, the most important of which are shared Files, 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 shared Files section contains the files that are required to execute the 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.

7.2.3 MPI applications

Message Passing Interface (MPI) is a specification for developing parallel programs that communicate by exchanging messages.

MPI has originated as an attempt to create common ground from the several distributed shared memory and message-passing infrastructures available for distributed computing. Now a days, MPI has become a de facto standard for developing portable and efficient message-passing HPC applications.

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.

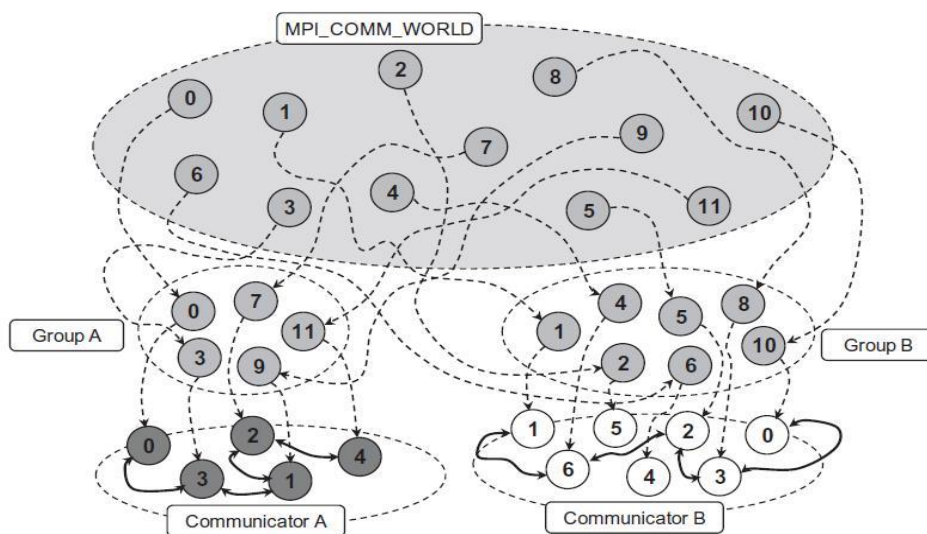


FIGURE 7.4

MPI reference scenario.

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.

Each MPI process is assigned a rank within the group.

The rank is a unique identifier that allows processes to communicate with each other within a 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, it is possible to use all the MPI functions to send or receive messages in either asynchronous or synchronous mode.

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.

A common model used in MPI is the master-worker model, where by one MPI process 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.

The output of the compilation process can be run as a distributed application by using a specific tool provided with the MPI implementation.

One of the most popular MPI software environments is developed by the Argonne National Laboratory in the United States.

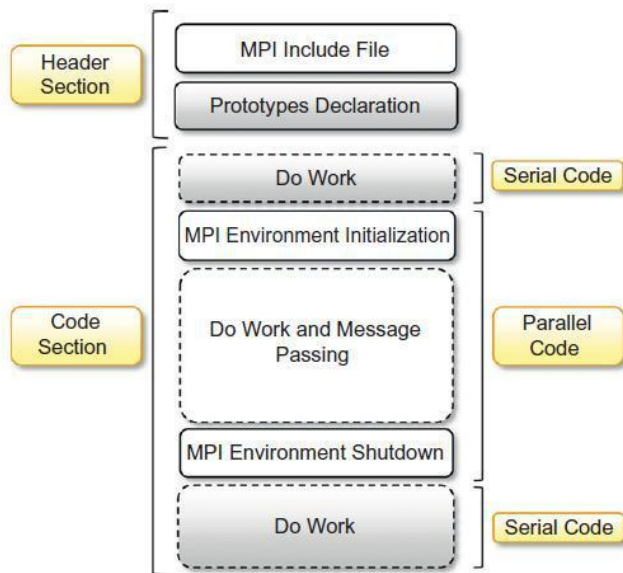


FIGURE 7.5
MPI program structure.

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 determine the way in which the applications are scheduled as well as where they are scheduled.

1 What is a workflow?

A workflow is the automation of a business process, in whole or part, during which documents, information, or tasks are passed from one participant (a resource; human or machine) to another for action, according to a set of procedural rules.

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.

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 constitute the input files of another task.

The DAG in **Figure 7.6** describes a sample Montage workflow. Montage is a toolkit for assembling images into mosaics; it has been specially designed to support astronomers in composing the images taken from different telescopes or points of view into a coherent image.

The 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.

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.

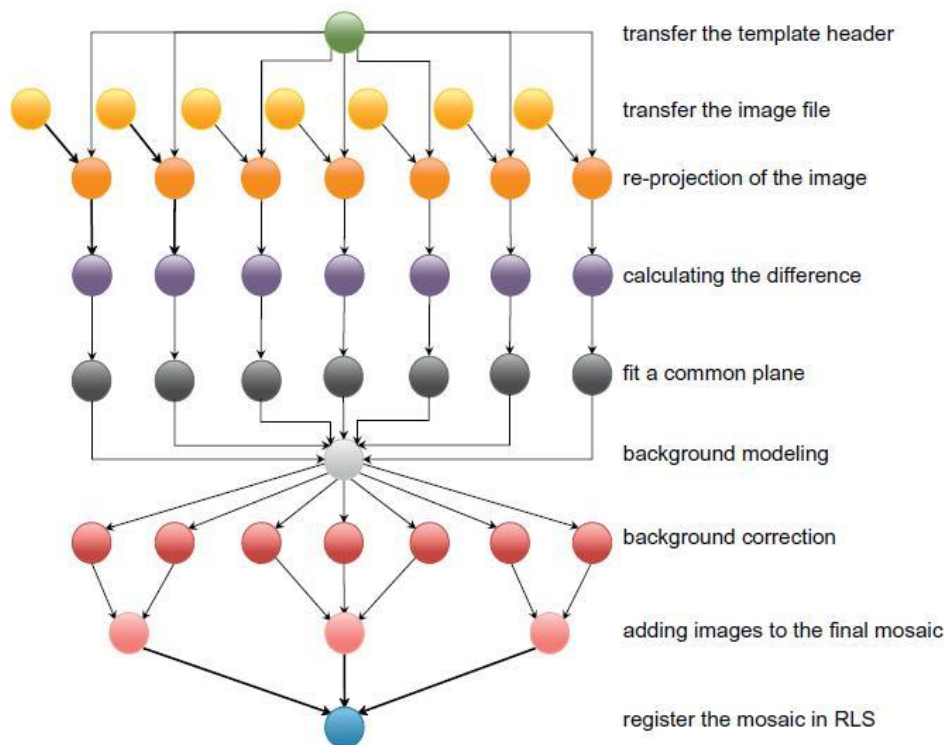


FIGURE 7.6

Sample Montage workflow.

2 Workflow technologies

Business-oriented computing workflows are defined as compositions of services.

There are specific languages and standards for the definition of workflows, such as Business Process Execution Language (BPEL).

An abstract reference model for a workflow management system, as depicted in **Figure 7.7**.

Design tools allow users to visually compose a workflow application.

This specification is 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.

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 of the most relevant technologies for designing and executing workflow-based applications are:

1. **Kepler,**
2. **DAGMan,**
3. **Cloudbus Workflow Management System, and**
4. **Offspring.**

1. Kepler

Kepler is an open-source scientific workflow engine.

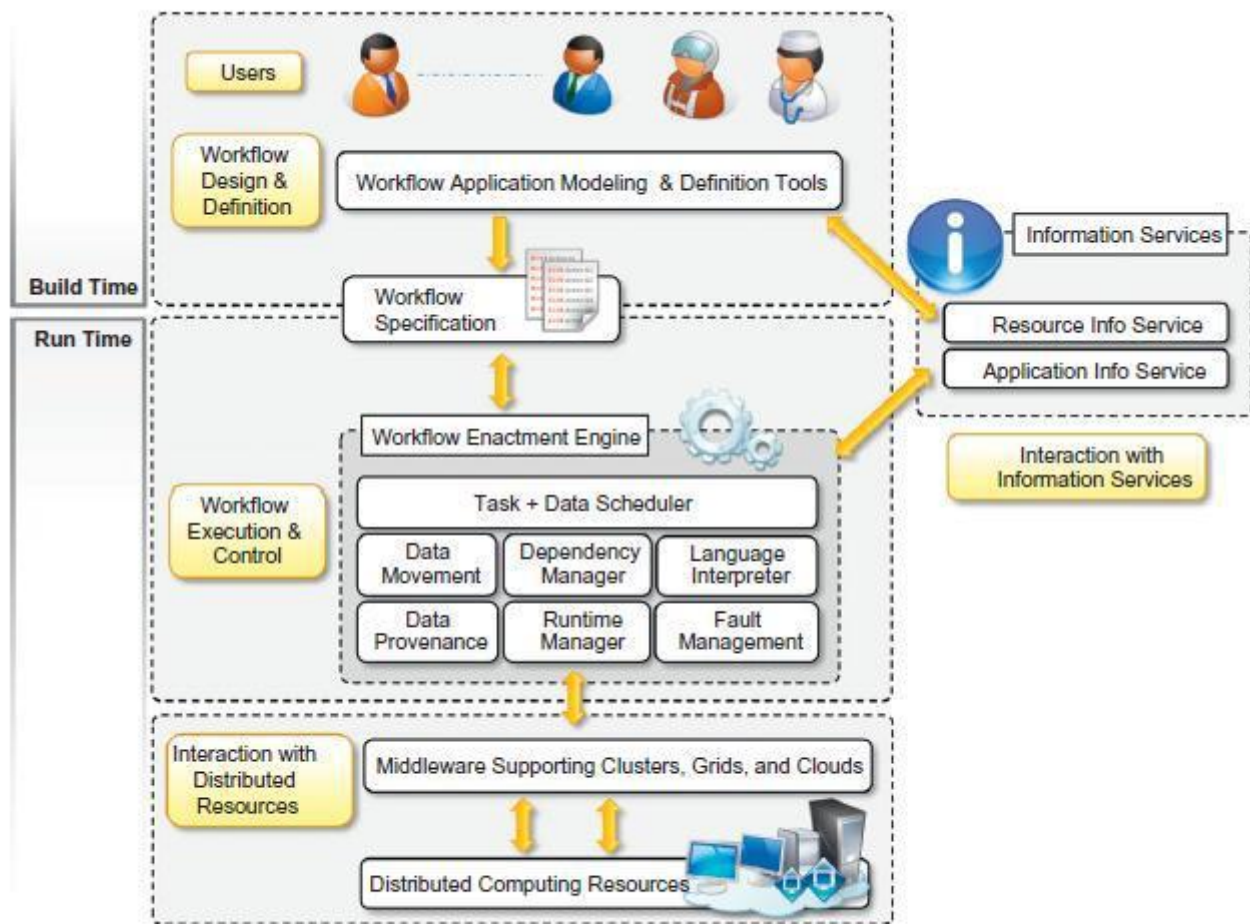
The system is based on the Ptolemy II system, which provides a solid platform for developing dataflow-oriented workflows.

Kepler provides a design environment based on the concept of actors, which are reusable and independent blocks of computation such as Web services, data- base calls. The connection between actors is made with ports.

An actor consumes data from the input ports and writes data/results to the output ports.

Kepler supports different models, such as synchronous and asynchronous models.

The workflow specification is expressed using a proprietary XML language.

**FIGURE 7.7**

Abstract model of a workflow system.

2. DAGMan

DAGMan (Directed Acyclic Graph Manager) constitutes an extension to the Condor scheduler to handle job interdependencies.

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.

3. Cloudbus Workflow Management System

Cloudbus Workflow Management System (WfMS) is a middleware platform built for managing large application workflows on distributed computing platforms such as grids and clouds.

It comprises software tools that help end users compose, schedule, execute, and monitor workflow applications through a Web-based portal.

The portal provides the capability of uploading workflows or defining new ones with a graphical editor.

To execute workflows, WfMS relies on the Gridbus Broker, a grid/cloud resource broker that supports the execution of applications with quality-of-service (QoS) attributes.

4. Offspring

It 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 is the ability to define dynamic workflows.

This strategy represents a semi structured workflow that can change its behaviour at runtime according to the execution of specific tasks.

This allows developers to dynamically control the dependencies of tasks at runtime.

Offspring supports integration with any distributed computing middleware that can manage a simple bag-of-tasks application.

Offspring allows the definition of workflows in the form of plug-ins.

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.

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.

Figure 7.8 provides an overall view of the components of the Task Programming Model and their roles during application execution.

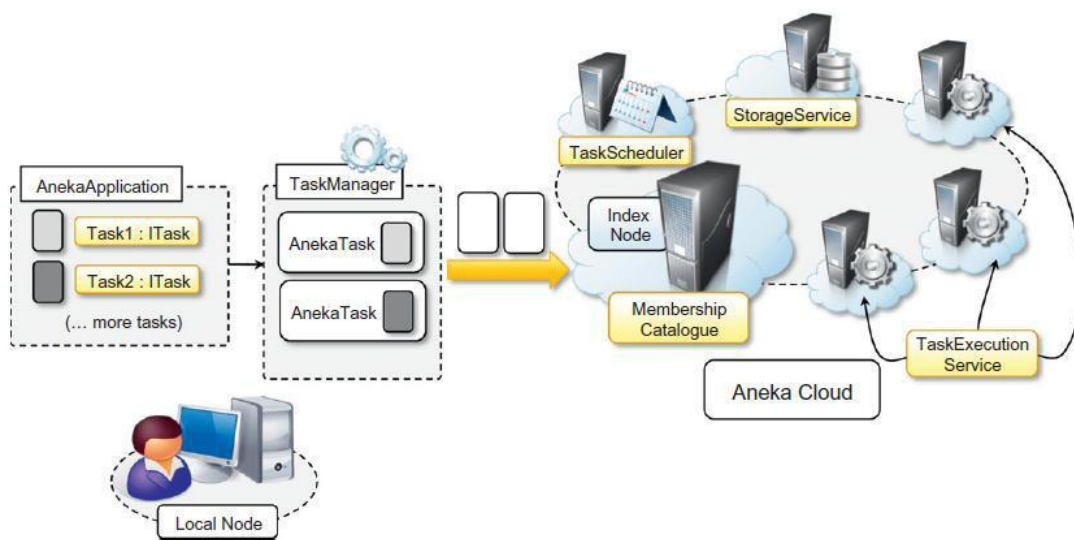


FIGURE 7.8

Task programming model scenario.

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 Aneka Application 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.

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 Execution Service for task execution and for monitoring task state.

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.

1. ITask and AnekaTask
2. Controlling task execution
3. File management
4. Task libraries
5. Web services integration

1. ITask and AnekaTask

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.

```
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.

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; they need to be serializable, since task instances are created and moved over the network.

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.

Listing 7.2 describes a simple implementation of a task class that computes the Gaussian distribution for a given point x.


```
// 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.

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.

WorkUnit has two collections of files, Input Files and Output Files; 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.

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

LISTING 7.3

Wrapping an ITask into an AnekaTask Instance.

2. Controlling task execution

Task classes and AnekaTask define the computation logic of a task-based application.

Aneka Application class provides the basic feature for implementing the coordination logic of the

application.

In task programming, it assumes the form of `Aneka Application<AnekaTask, TaskManager>`.

The operations provided for the task model are:

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

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.

Dynamic submission of tasks is a more efficient technique and involves the submission of tasks as a result of the event-based notification mechanism implemented in the `Aneka Application` class.

Listing 7.4 shows how to create and submit 400 Gauss tasks as a bag by using the static submission approach.

Each task can be referenced using its unique identifier (`Work Unit.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.

```
// 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.

Listing 7.5 extends the previous example and implements a dynamic task submission strategy for refining the computation of Gaussian distribution.

To capture the failure and the completion of tasks, it is necessary to listen to the events `WorkUnitFailed` and `WorkUnitFinished`.

This class exposes a `WorkUnit` property that, if not null, gives access to the 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(AnekaTasktask)` 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.

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 behaviour 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 once all tasks are terminated.

```

///<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 = new AnekaTask(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.

Listing 7.6 provides a complete implementation of the task submission program, implementing dynamic submission and the appropriate synchronization logic.

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 unblocks them a in thread by signalling the semaphore.

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.

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.

```

// 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 configFile = "conf.xml";
                if (args.Length > 0)
                {
                    configFile = args[0];
                }
                // get an instance of the Configuration class from file
                Configuration conf = Configuration.GetConfiguration(configFile);
                // create an instance of the GaussApp and starts its execution
                // with the given configuration instance
                GaussApp application = new GaussApp();
                application.SubmitApplication(conf);
            }

            catch(Exception ex)
            {
                IOUtil.DumpErrorReport(ex, "Fatal error while executing application.");
            }
            finally
            {
                // terminate the logging thread
                Logger.Stop();
            }
        }
    }
}

```



```

/// <summary>
/// Application submission method.
/// </summary>
/// <param name="conf">Application configuration.</param>
public void SubmitApplication(Configuration conf)
{
    // initialize the semaphore and the number of
    // task initially submitted
    this.semaphore = new ManualResetEvent(false);
    this.taskCount = 400;

    // specify that the submission of task is dynamic
    conf.SingleSubmission = false;
    this.app = new AnekaApplication<Task, TaskManager>(conf);
    // attach methods to the event handler that notify the client code
    // when tasks are completed or failed
    this.app.WorkUnitFailed +=
        new EventHandler<WorkUnitEventArgs<AnekaTask>>(this.OnWorkUnitFailed);
    this.app.WorkUnitFinished +=
        new EventHandler<WorkUnitEventArgs<AnekaTask>>(this.OnWorkUnitFinished);

    // attach the method OnAppFinished to the Finished event so we can capture
    // the application termination condition, this event will be fired in case of
    // both static application submission or dynamic application submission
    app.Finished += new EventHandler<ApplicationEventArgs>(this.OnAppFinished);

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

    // wait until signaled, once the thread is signaled the application is completed
    this.semaphore.Wait();
}

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

    this.taskCount--;
    if (this.taskCount == 0)
    {
        this.app.StopExecution();
    }
}
}

```



```

/// <summary>
/// Event handler for task completion.
/// </summary>
/// <param name="sender">Event source: the application instance.</param>
/// <param name="args">Event arguments.</param>
private void OnWorkUnitFinished(object sender, WorkUnitEventArgs<AnekaTask> args)
{
    // we do not have to synchronize this operation because
    // events handlers are run all in the same thread, and there
    // will not be other threads updating this variable
    this.taskCount--;

    // if the task is completed for sure we have a WorkUnit instance
    // and we do not need to check as we did before.
    GaussTask gauss = (GaussTask) args.WorkUnit.Task;
    // we check whether it is an initially submitted task or a task
    // that we submitted as a reaction to the completion of another task
    if (task.X - Math.Abs(task.X) == 0)
    {
        // ok it was an original task, then we increment of 0.5 the
        // value of X and submit another task
        GaussTask frag = GaussTask();
        frag.X = gauss.X + 0.5;

        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

GaussApplication.

3. File management

Task-based applications normally deal with files to perform their operations.

Files may constitute input data for tasks, may contain the result of a computation, or may represent executable code or library dependencies.

Any model based on the WorkUnit and ApplicationBase classes has built-in support for file management.

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 File Data instance provides information about a file:

- Its nature: whether it is a shared file, an input file, or an outputfile
- Its path both in the local and in the remote file system, including a different name

- A collection of attributes that provides other information.

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.

```
// 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"
FileData piTab = new FileData("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"
FileData piDat = new FileData("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 = new AnekaTask(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 reference task.Id

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

```

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

File dependencies management.

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.Storage Bucket Id 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 File Attributes. 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 File Attributes. 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 Storage BucketId property set will be staged out to the corresponding remote server.

Listing 7.8 shows a sample configuration file containing the settings required to access the remote files

<https://hemanthrajhemu.github.io>

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.

```
<?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.

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 Local Copy Task 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 Execute Task 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.
- Substitute operation. The Substitute Task performs a search-and-replace operation within a given file by saving the resulting file under a different name.
- File deletion. The Delete Task deletes a file that is accessible through the file system on the remote node.
- Timed delay. The Wait Task introduces a timed delay. This task can be used in several scenarios; it can be used for profiling or for simulation of the execution.
- Task composition. The Composite Task implements the composite pattern and allows expressing a task as a composition of multiple tasks that are executed in sequence.

5 Web services integration

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.

The reference scenario for Web-based submission is depicted in **Figure 7.9**.

Users create a distributed application instance on the cloud, 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 abort it if there is no need to complete job execution.

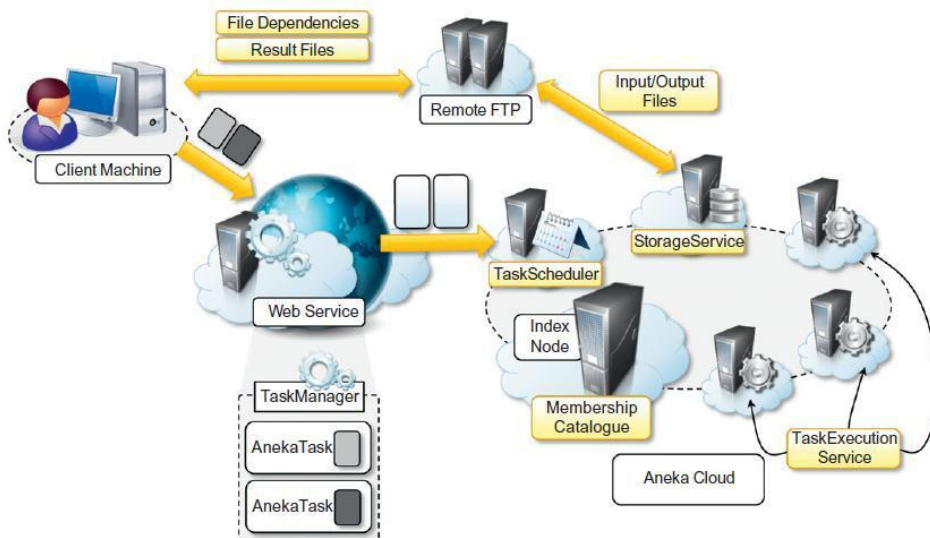


FIGURE 7.9

Web service submission scenario.

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.

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 PSM is organized into several name spaces 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.

1 Object model

2 Development and monitoring tools

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.PSM JobInfo).

Figure 7.11 shows the most relevant components of the object model.

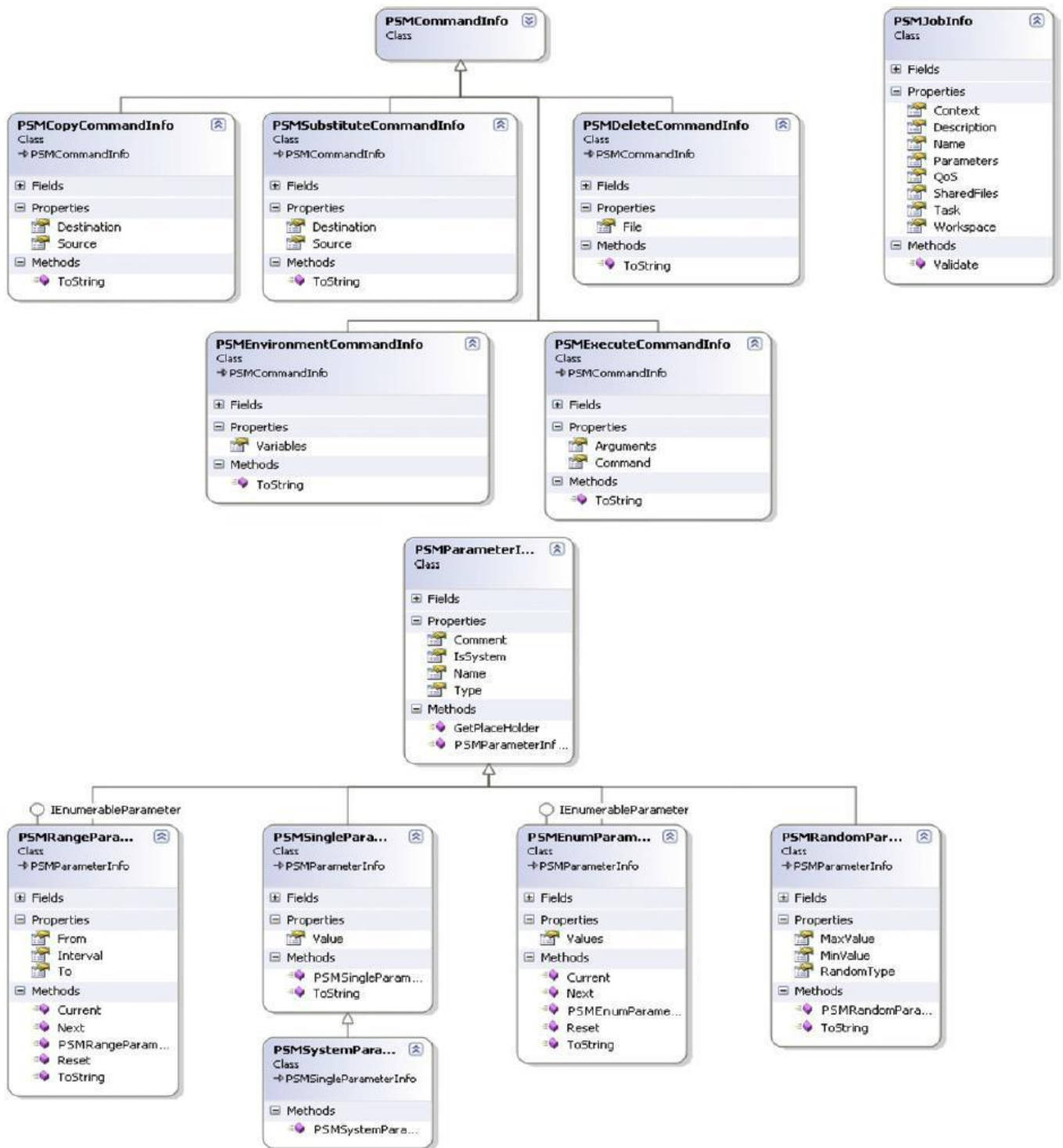


FIGURE 7.11

PSM object model (relevant classes).

Currently, it is possible to specify five different types of parameters:

- Constant parameter (PSM Single Parameter Info). This parameter identifies a specific value that is set at design time and will not change during the execution of the application.
- Range parameter (PSM Range Parameter Info). 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 (PSM Random Parameter Info). This parameter allows the generation of a random value in between a given range defined by a lower and an upper bound.
- Enumeration parameter (PSM Enum Parameter Info). 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 (PSM System Parameter Info). This parameter allows for mapping specific value that

will be substituted at runtime while the task instance is executed on the remote node.

The available commands for composing the task template perform the following operations:

- Local file copy on the remote node (PSM Copy Command Info)
- Remote file deletion (PSM Delete Command Info)
- Execution of programs through the shell (PSM Execute Command Info)
- Environment variable setting on the remote node (PSM Environment Command Info)
- String pattern replacement within files (PSM Substitute Command Info)

A parameter sweep application is executed by means of a job manager (IJob Manager), 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.

The implementation of IJob Manager 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.

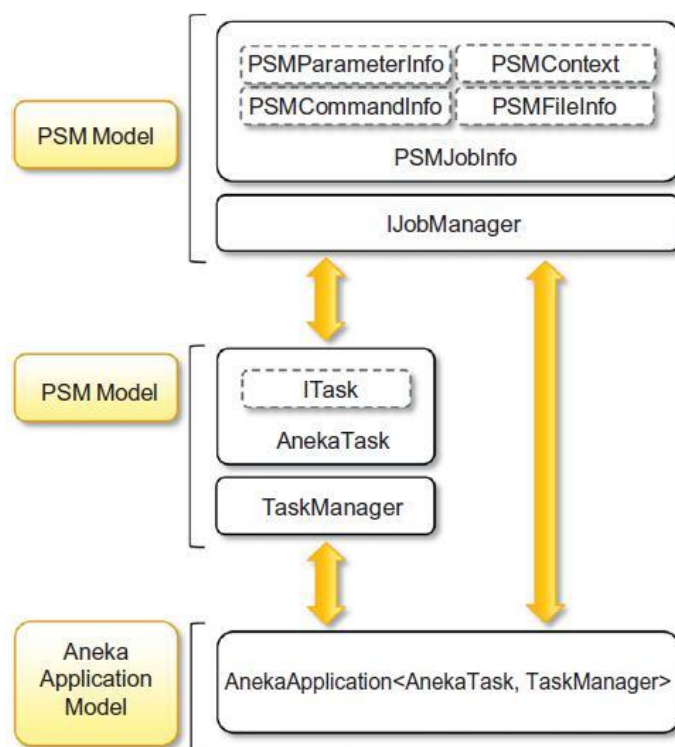


FIGURE 7.12

Parameter sweep model APIs.

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.

These tools are the:

Aneka Design Explorer and

The **Aneka PSM Console**.

Aneka Design Explorer

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

The Aneka PSM Console

The console offers a simplified interface for running applications with essential features for monitoring their execution.

Two different workflow managers can leverage Aneka for task execution:

- ## 1. The Workflow Engine

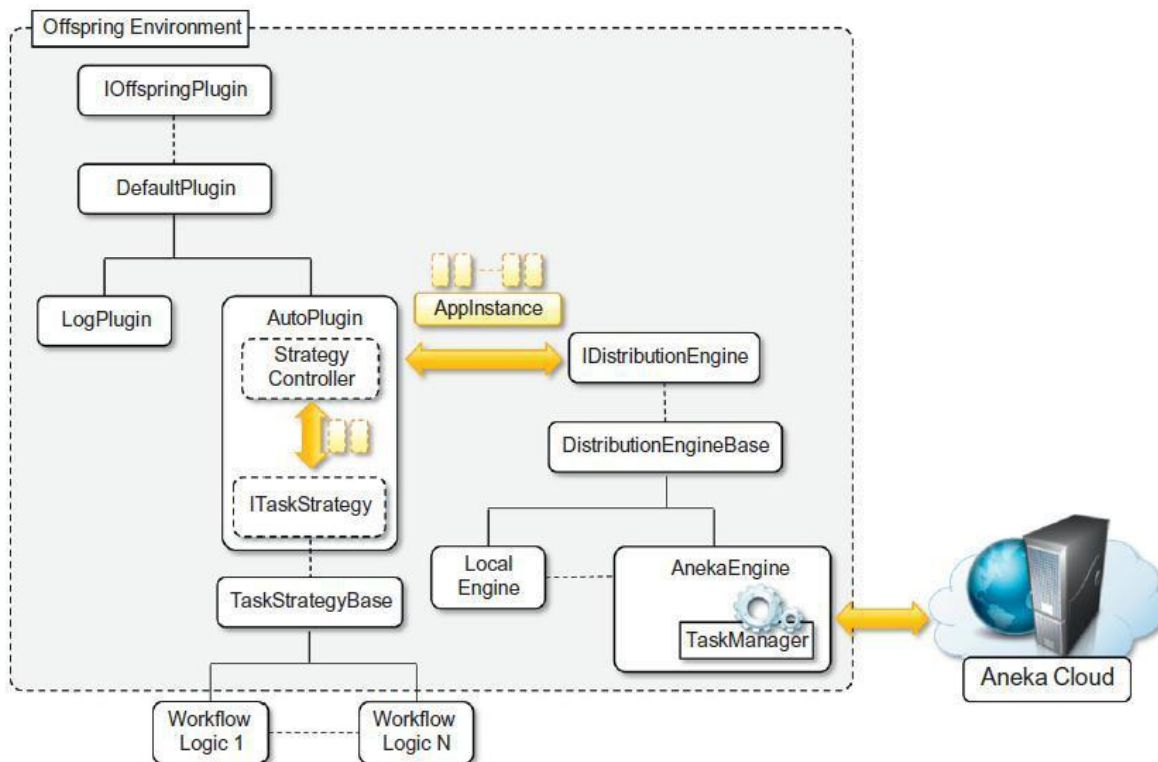
The Workflow Engine plug-in for Aneka which allows client applications developed with any technology and language to leverage Aneka for task execution.

Figure 7.13 describes the Offspring architecture. The system is composed of two types of components: **plug-ins** and a **distribution engine**.

Auto Plugin 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. The Strategy Controller, decouples the strategies from the distribution engine.

The connection with Aneka is realized through the Aneka Engine, which implements the operations of IDistribution Engine for the Aneka middleware and relies on the services exposed by the task model programming APIs.



Offspring architecture.

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.

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, where as 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.

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.

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.

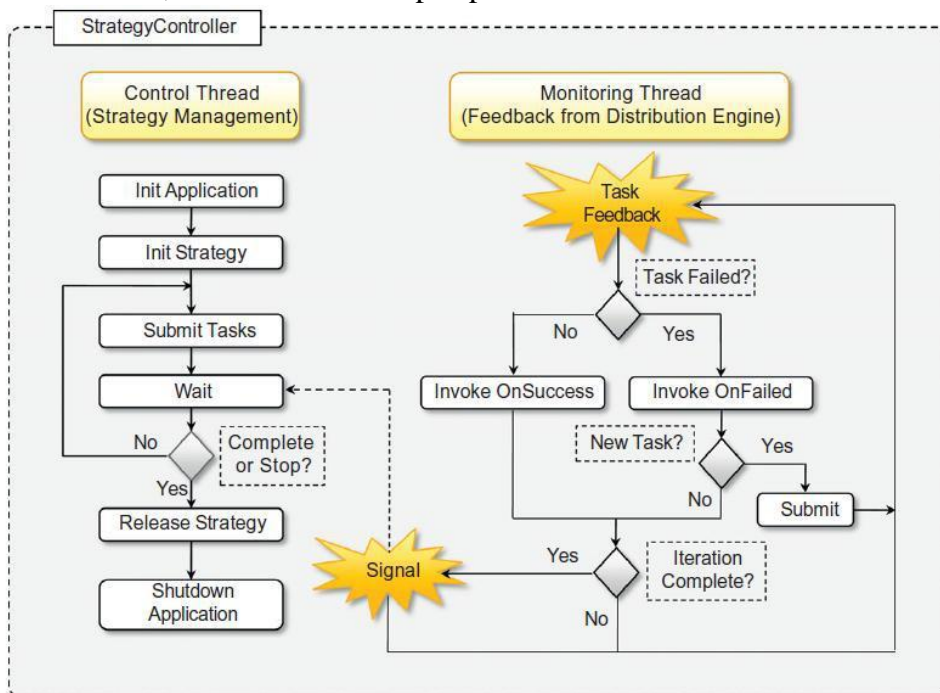


FIGURE 7.14

Workflow coordination.

MODULE – III

Concurrent Computing (Chapter 6)

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 processing units to share the computation load.

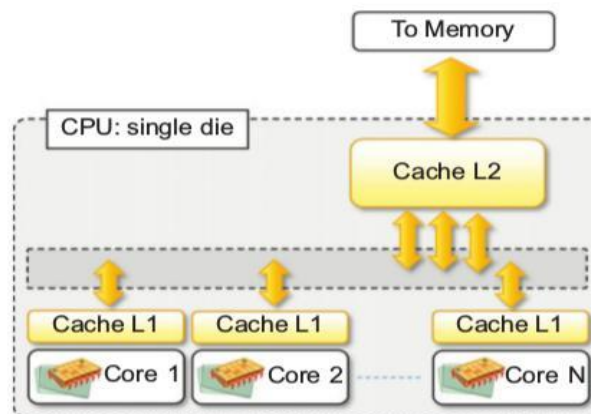


FIGURE 6.1

Multicore processor.

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**.

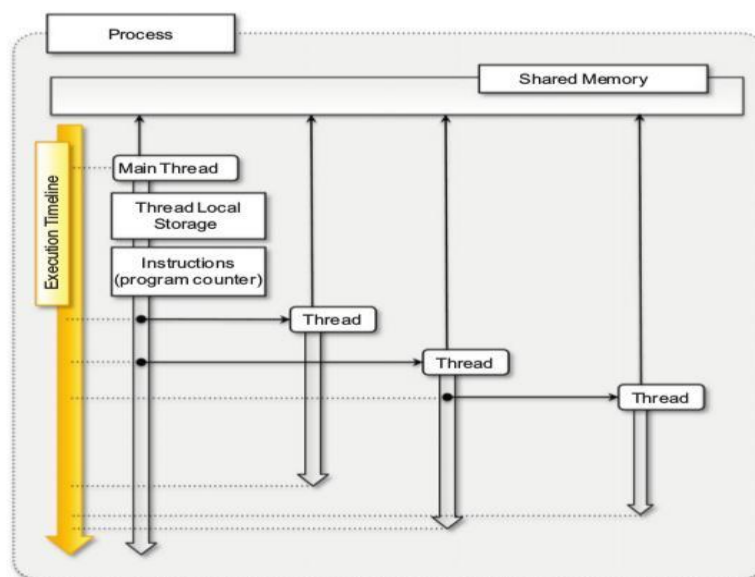


FIGURE 6.2

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.

6.2.2 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.

6.2.3 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 for each of the resulting matrix elements is the following:

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

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.

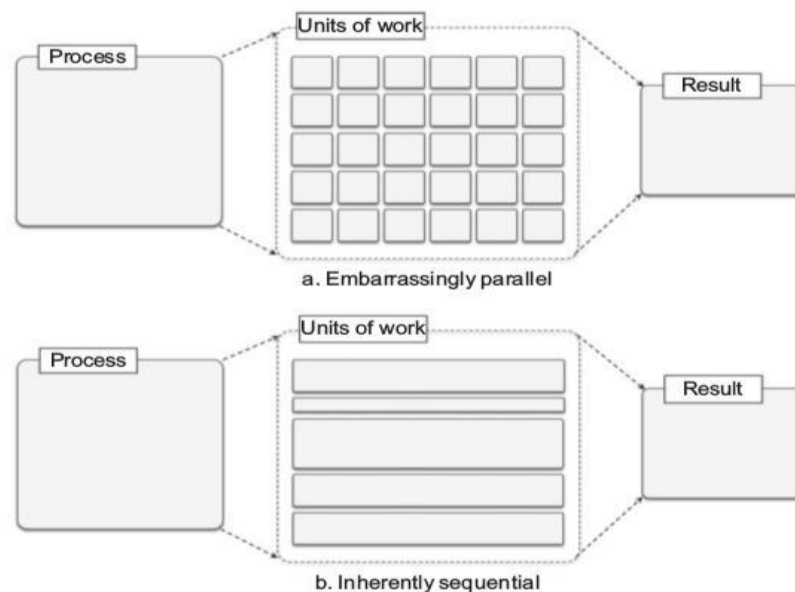
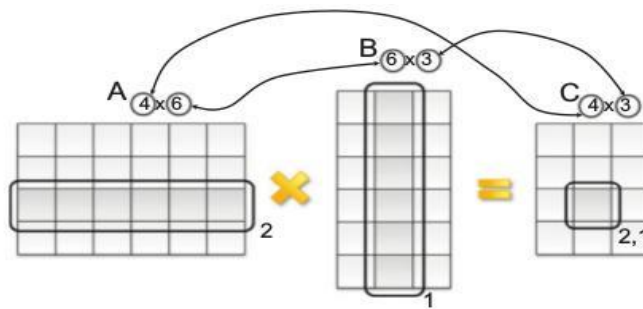


FIGURE 6.3

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. **Listing 6.1** shows the class **ScalarProduct**.

/// Class ScalarProduct. Computes the scalar product between the row and the column arrays. public class ScalarProduct

```
{
    /// Scalar product.
    private double result;
    /// Gets the resulting scalar product.
    public double Result{ get { returnthis.result; } }
    /// Arrays containing the elements of the row and the column to
    multiply. private double[] row, column;
    /// Creates instance of the ScalarProduct class and configures it with the given row & column arrays.
    /// <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.column = column;
    }
    /// Executes the scalar product between the row and the column.
    /// <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.

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 using System;

```

using System.Threading;
using System.Collections.Generic;
/// Class MatrixProduct. Performs the matrix product of two
matrices. public class MatrixProduct
{
    /// First and second matrix of the
    produt. private static double[,]a, b;
    /// Result matrix.
    private static double[,] c;
    /// Dictionary mapping thread instances to corresponding ScalarProduct instances that are
    run inside. private static IDictionary<Thread, ScalarProduct>workers.
    /// Read the command line parameters and perform the scalar product.
    ///<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();
    }
    /// Executes parallel matrix product by decomposing problem in independent scalar product between
    /// rows and columns.
    private static void ExecuteThreads()
    {
        MatrixProduct.workers = newList<Thread>();
        int rows = MatrixProduct.a.Length;
        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 get the columns
                // we need to traverse the array and copy 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 = new ScalarProduct(row, column);
        Thread worker =
        new Thread(new ThreadStart(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);
    }
}
/// Waits for the completion of all the threads and composes the final result
matrix. 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);
}
/// Reads the matrices.
private static void ReadMatrices()
{
    // code for reading the matrices a and b
}
/// Prints the given matrix.
//<param name="matrix">Matrix to print.</param>
private static void PrintMatrices(double[,] matrix)
{
    // code for printing the matrix.
}
}
}

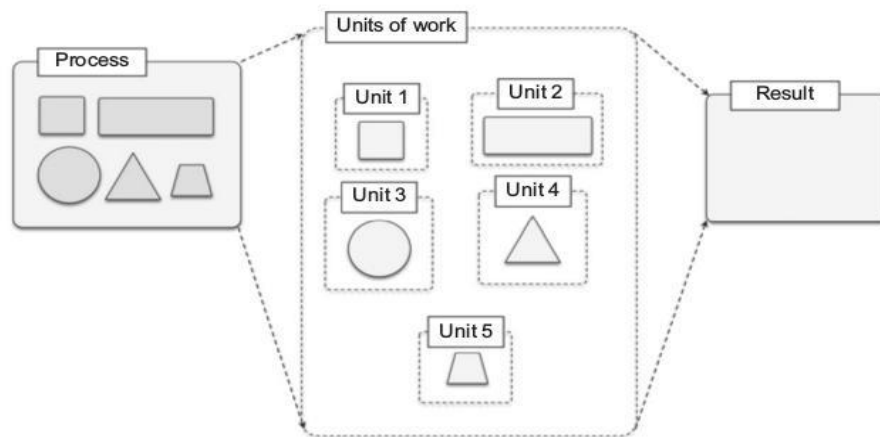
```

LISTING 6.2 MatrixProduct Class (Main Program).

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.

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. A possible implementation of a parallel version of computation is shown in **Listing 6.3**.

```
using System;
using System.Threading;
using System.Collections.Generic;
/// Delegate UpdateResult. Function pointer that is used to update the final result from the slave threads once
/// the computation is completed.
/// <param name="x">partial value to add.</param>
public delegate void UpdateResult(double x);
/// Class Sine. Computes the sine of a given
value. public class Sine
{
    /// Input value for which the sine function is
    computed. private double x;
    /// Gets the input value of the sine function.
    public double X { get { return this.x; } }
    /// Result value.
    private double y;
    /// Gets the result value of the sine function.
    public double Y { get { return this.y; } }
    /// Function pointer used to update the
    result. private UpdateResult updater;
    /// Creates an instance of the Sine and sets the input to the given angle.
    /// <param name="x">Angle in radians.</param>
    /// <param name="updater">Function pointer used to update the
    result.</param> public Sine(double x, UpdateResult updater)
    {
        this.x = x; this.updater
        = updater;
```

```

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

```



```

    /// Function pointer used to update the
    result. private Updater;
    /// Creates an instance of the Tangent and sets the input to the given angle.
    /// <param name="x">Angle in radians.</param>
    /// <param name="updater">Function pointer used to update the
    result.</param> public Tangent(double x, Updater updater)
    {
        this.x = x; this.updater
        = updater;
    }
    /// Executes the cosine function.
    public void Apply()
    {
        this.y = Math.Tan(this.x);
        if (this.updater != null)
        {
            this.updater(this.y);
        }
    }
}
/// Class Program. Computes the function  $\sin(x) + \cos(x) + \tan(x)$ . public class Program
{
    /// Variable storing the computed value for the
    function. private static double result;
    /// Synchronization instance used to avoid keeping track of the
    threads. private static object synchRoot = new object();
    /// Read the command line parameters and perform the scalar product.
    /// <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 = new UpdateResult(Program.Sum);
        // creates the sine thread.
        Sine sine = new Sine(x, updater);
        Thread tSine = new Thread(new
        ThreadStart(sine.Apply)); // creates the cosine thread.
        Cosine cosine = new Cosine(x, updater);
        Thread tCosine = new Thread(new
        ThreadStart(cosine.Apply)); // creates the tangent thread.
        Tangent tangent = new Tangent(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);
    }
    /// Callback that is executed once computation in thread is completed and adds partial value passed
    /// as a parameter to result.
    /// <param name="partial">Partial value to add.</param>
    private static void Sum(double partial)
    {
        lock(Program.synchRoot)
        {
            Program.result += partial;
        }
    }
}

```

LISTING 6.3 Mathematical Function.

3 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
<i>System.Threading</i>	<i>Aneka.Threading</i>
<i>Thread</i>	<i>AnekaThread</i>
<i>Thread.ManagedThreadId (int)</i>	<i>AnekaThread.Id (string)</i>
<i>Thread.Name</i>	<i>AnekaThread.Name</i>
<i>Thread.ThreadState (ThreadState)</i>	<i>AnekaThread.State</i>
<i>Thread.IsAlive</i>	<i>AnekaThread.IsAlive</i>
<i>Thread.IsRunning</i>	<i>AnekaThread.IsRunning</i>
<i>Thread.IsBackground</i>	<i>AnekaThread.IsBackground[false]</i>
<i>Thread.Priority</i>	<i>AnekaThread.Priority[ThreadPriority.Normal]</i>
<i>Thread.IsThreadPoolThread</i>	<i>AnekaThread.IsThreadPoolThread [false]</i>
<i>Thread.Start</i>	<i>AnekaThread.Start</i>
<i>Thread.Abort</i>	<i>AnekaThread.Abort</i>
<i>Thread.Sleep</i>	[Not provided]
<i>Thread.Interrupt</i>	[Not provided]
<i>Thread.Suspend</i>	[Not provided]
<i>Thread.Resume</i>	[Not provided]
<i>Thread.Join</i>	<i>AnekaThread.Join</i>

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. 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.

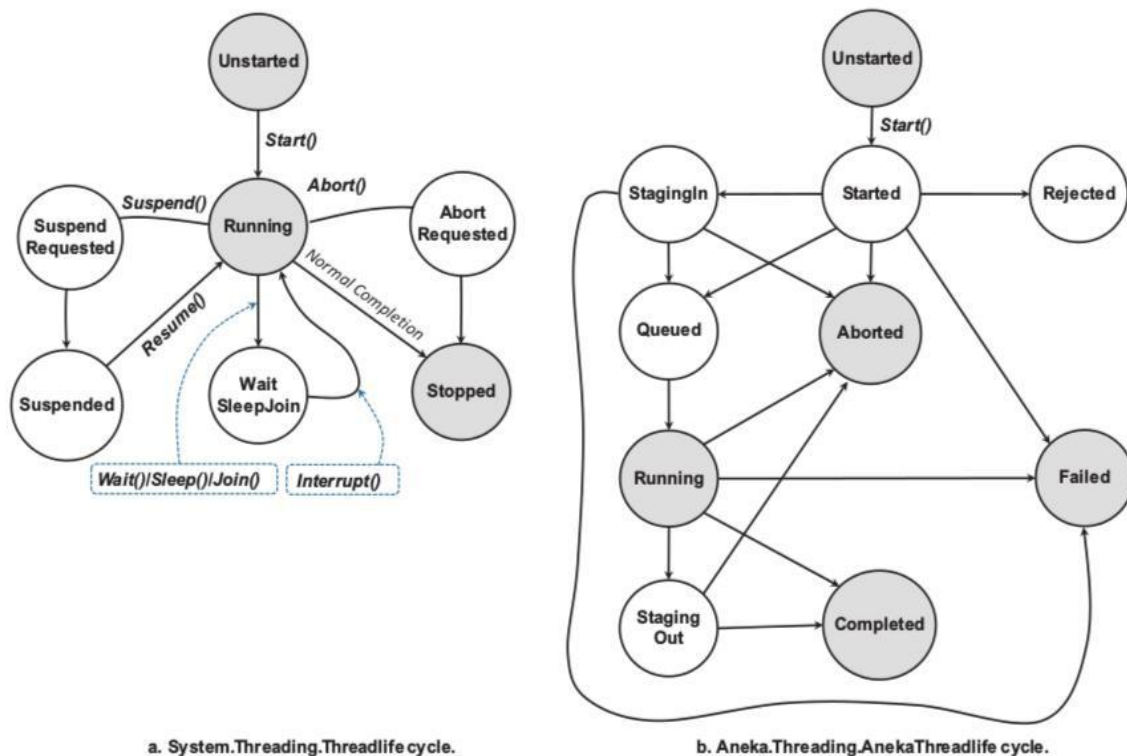


FIGURE 6.6

Thread life-cycle comparison.

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

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 `AnekaApplicationN<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.

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.

```
// 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;
///Creates an instance of the Aneka Application configured to use Thread Programming Model.
/// <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 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.

Listing 6.5 provides a very simple example of how to create Aneka threads.

```
// ..... continues from the previous listing
///Thread worker method
private void WorkerMethod()
{
    // .....
}
```

```

///Creates a collection of threads that are executed in the context of given application.
/// <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 = new ThreadStart(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.

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:

1. Including the System.Runtime.Serialization namespace.
2. 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.
3. Providing a constructor with the following signature: ScalarProduct(SerializationInfo, StreamingContext). This constructor is invoked when the instance is deserialized.

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.

```

using System.Runtime.Serialization;
/// 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 ISerializable
interface. [Serializable]
public class ScalarProduct : ISerializable
{
    /// Scalar product.
    private double result;
    /// Gets the resulting scalar product.
    public double Result{get { returnthis.result; }}
    /// Arrays containing the elements of the row and the column to
    multiply. private double[] row, column;
    /// Creates an instance of ScalarProduct class and configures it with given row and column arrays.
    /// <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 = column;
    }
    /// Deserialization constructor used by .NET runtime to recreate instances of types implementing

```

```

/// custom serialization.
/// <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[];
}
/// Executes the scalar product between the row and the column.
/// <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];
    }
}
/// Serialization method used by .NET runtime to serialize instances of types implementing custom
/// serialization.
/// <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[];
}
/// Executes the scalar product between the row and the column.
/// <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];
    }
}
/// Serialization method used by .NET runtime to serialize instances of types implementing custom
/// serialization.
/// <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 ScalarProduct Class (Modified Version).

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).

```
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;
/// Class MatrixProduct. Performs the matrix product of two
matrices. public class MatrixProduct
{
    /// First and second matrix of the produt.
    private static double[,],a, b;
    /// Result matrix.
    private static double[,] c;
    /// 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.
    private static IDictionary<AnekaThread, ScalarProduct> workers.
    /// Reference to the distributed application the threads belong to.
    private static AnekaApplication<AnekaThread, ThreadManager> app;
    /// Read the command line parameters and perform the scalar product.
    /// <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. 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();
}
/// Executes parallel matrix product by decomposing problem in independent scalar product between rows
/// and columns.

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 & number of columns is represented by length of
    2nd 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 = new ScalarProduct(row, column);
        // we change the System.Threading.Thread class with the corresponding
        // Aneka.Threading.AnekaThread class and reference the application instance.
        AnekaThread worker = new AnekaThread(new ThreadStart(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);
    }
}
/// Waits for the completion of all the threads and composes the final result
matrix. 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);
}
/// Reads the matrices.
private static void ReadMatrices()
{
    // code for reading the matrices a and b
}
/// Prints the given matrix.
/// <param name="matrix">Matrix to print.</param>
private static void PrintMatrices(double[,] matrix)
{
    // code for printing the matrix.
}
/// Creates an instance of the Aneka Application configured to use the
/// Thread Programming Model.
/// <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 = new Uri("tcp://localhost:9090/Aneka");
    conf.Credentials = new UserCredentials("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 MatrixProduct Class (Modified Version).**6.4.3 Functional decomposition: Sine, Cosine, and Tangent**

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 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**).

```

using System;
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;
/// <param name="x">partial value to add.</param>
public delegate void UpdateResult(double x);
/// Class Sine. Computes the sine of a given
value. [Serializable]
public class Sine
{
    /// Input value for which the sine function is
    computed. private double x;
    /// Gets or sets the input value of the sine function.
    public double X { get { return this.x; } set { this.x = value; } }
    /// Result value.
    private double y;
    /// Gets or sets the result value of the sine function.
    public double Y { get { return this.y; } set { this.y = value; } }
    //we need a default constructor, which is automatically provided by compiler if we do not
    specify any constructor.
    /// Creates an instance of the Sine and sets the input to the given angle.
    /// <param name="x">Angle in radiants.</param>
    /// <param name="updater">Function pointer used to update the
    result.</param> public void Apply()
    {
        this.y = Math.Sin(this.x);
        // we cannot use this anymore because there is no shared memory space.
    }
}
/// 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. [Serializable]
public class Cosine
{
    /// Input value for which the cosine function is computed.
    private double x;
    /// Gets or sets the input value of the cosine function.
    public double X { get { return this.x; } set { this.x = value; } }
    /// Result value.
    private double y;
    /// Gets or sets the result value of the cosine function.
    public double Y { get { return this.y; } set { this.y = value; } }
    /// Executes the cosine
    function. public void Apply()
    {
        this.y = Math.Cos(this.x);
    }
}
/// 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. [Serializable]

public class Tangent

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

/// 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. [Serializable]

public class Tangent

```
{
    /// Input value for which the tangent function is
    computed. private double x;
    /// Gets or sets the input value of the tangent function.
    public double X { get { return this.x; } set { this.x = value; } }
    /// Result value.
    private double y;
    /// Gets or sets the result value of the tangent function.
    public double Y { get { return this.y; } set { this.y = value; } }
    /// Executes the tangent
    function. public void Apply()
    {
        this.y = Math.Tan(this.x);
    }
}
```

/// Class Program. Computes the function $\sin(x) + \cos(x) + \tan(x)$. public class Program

```
{
    /// Variable storing the coputed value for the
    function. private static double result;
    ///we do not need synchronization anymore, because update of global sum is done sequentially.
    /// Synchronization instance used to avoid keeping track of the threads.
    private static object synchRoot = new object();
    /// Reference to the distributed application the threads belong to .
    private static AnekaApplication <AnekaThread, ThreadManager> app;
    /// Read the command line parameters and perform the scalar product.
    /// <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.
app = Program.CreateApplication();
// gets a value for x
double x = 3.4d;
// creates the function pointer to the update method.
UpdateResult updater = new UpdateResult(Program.Sum);
// creates the sine thread.
Sine sine = new Sine(x, updater);
AnekaThread tSine = new AnekaThread(new
ThreadStart(sine.Apply), app); // creates the cosine thread.
Cosine cosine = new Cosine(x, updater);
AnekaThread tCosine = new AnekaThread(new
ThreadStart(cosine.Apply), app); // creates the tangent thread.
Tangent tangent = new Tangent(x, updater);
AnekaThread tTangent = new AnekaThread(new ThreadStart(tangent.Apply), app);
// shuffles the execution
order. tTangent.Start();
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 //Target property in order to obtain the object with 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 " + "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 "+ application" );
    }
}
// stops the logging thread .

```

```

        Logger.Stop();
    }
}
private static void Sum(double partial)
{
    lock(Program.synchRoot)
    {
        Program.result += partial;
    }
}
/// Creates an instance of the Aneka Application configured to use the
/// Thread Programming Model.
/// <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 Mathematical Function (Modified Version).