# Parallel Programming with Aneka's Thread Programming Model

This tutorial will guide you through the process of writing parallel programs using Aneka's Thread Programming Model. It introduces fundamental concepts in Aneka's Thread Programming Model and illustrates the steps required to design and develop parallel applications with Aneka. Three simple and interesting examples of increasing complexity are presented in order to demonstrate the ease of using Aneka's object-oriented approach to writing parallel applications using distributed threads.

### Objectives

At the end of this tutorial you should be able to:

- understand Aneka's Thread Programming Model
- understand the tradeoffs between local and distributed threads
- design parallel applications by decomposing the problem
- write parallel applications using Aneka's Thread Programming Model

## 1 Introduction

Traditional thread programming has always been carried out within the realm of a single process. That is a process, composed of multiple threads of execution, share memory and other resources but execute within the confines of the process's memory space. The operating system allocates a fraction of the *timeslice* assigned to the entire process, to each of the threads within. This process, known as *context switching*, creates the illusion of running multiple threads concurrently. In modern machines however, where multi-core processors are not uncommon, true parallelism is achieved by assigning each thread to a single core. A multi-threaded program brings numerous advantages such as improved responsiveness in interactive applications, increased throughput in I/O intensive applications, increased server responsiveness when handling multiple clients, and a simplified program structure. Threads within a single process however cannot communicate with threads in other processes by sharing memory, and must resort to using other forms of inter-process communication, such as named pipes and sockets.

As applications become increasingly complex there is greater demand for computational power than can be delivered by a single multi-core machine. Often this requires utilizing an entire cluster of, possibly multi-core, machines. Such problems typically require a large number of repetitive calculations on different data sets. As a result, the problem can be broken down into smaller manageable units of work and then distributed across each of the nodes in the cluster. As with traditional threads, concurrent execution is thus achieved by executing each of these units of work simultaneously, but on different machines. Once the partial results have been computed by the different nodes, the results can be gathered at the client machine and combined to produce the final result. A simple relationship can be established between the total time taken to complete the application, and the number of nodes available for execution. The larger the number of nodes available, the greater is the number of work units that can be distributed and executed

simultaneously and thus shorter is the time taken to complete the application.

Aneka takes traditional thread programming a step further. It lets you write multi-threaded applications in 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 the strict sense of the word, these “threads” are independent processes executing on different nodes, and do not share memory or other resources. But *AnekaThreads*, as they are called, lets you write applications using the same thread constructs for concurrency and synchronization as with traditional threads. This lets you easily port existing multi-threaded compute intensive applications to parallel versions that can run faster by utilizing multiple machines simultaneously. The rest of this tutorial will explore Aneka’s Thread Programming Model.

### 1.1 The Aneka Environment

Aneka is both a development and a runtime environment. As a development environment, Aneka provides a set of libraries that allow you to write parallel applications using one of the three programming models supported. These are the *Task*, *Thread* and *MapReduce* models. As a runtime environment, Aneka executes the units of work that constitute your application, in parallel. This tutorial assumes that you have a basic understanding of the Aneka runtime environment, and the role of the scheduler and execution nodes. You are encouraged to look at the online documentation for more details.

Note that running applications on Aneka requires access to a runtime environment pre-installed on a cluster. For development purposes however, you may run Aneka standalone on your personal computer.

## 2 Defining AnekaThreads

A running program consists of one or more *threads* executing within a *process*. A process is an instance of a program in execution. Each process has its own memory address space, the executable program and data. A program with a single thread is called a *single-threaded* program, while a program with multiple threads is called a *multithreaded* program. Each thread within a program has its own stack for maintaining its state. A process is therefore a grouping of resources, while a thread is an entity that can be scheduled for execution on the CPU. Threads are also known as *light-weight* processes. In .Net a thread is represented by the Thread class.

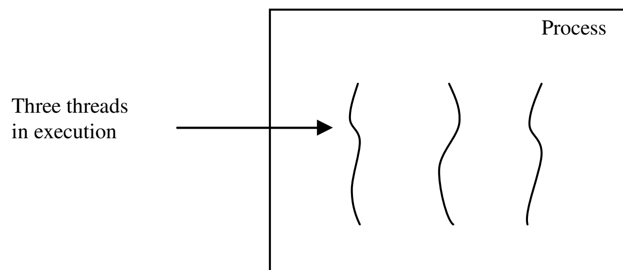


Figure 1: A process containing multiple threads

An *AnekaThread* is a unit of work that can be executed on a *remote* computer. Unlike standard threads, each AnekaThread is executed within a process of its own on the remote machine. An AnekaThread has a similar interface to the standard Thread class, and can be started and stopped (aborted) in much the same manner. AnekaThreads are however more simplistic and do not support all behaviors exposed by the .Net Thread class, such as managing thread priorities and operations such as *Suspend* and *Resume*. Figures 1 and 2 below show the essential differences between traditional threads and AnekaThreads.

## 3 Comparing Distributed Threads with Local Threads

An important difference between local and distributed threads lies in the sharing of resources. Local threads execute within the domain of a single process and communicate by sharing memory and other resources. For instance two local threads can read and write to the same data structure, and may coordinate their work using synchronization primitives such as locking and signaling. On the contrary, each distributed thread executes in isolation within a process of its own. AnekaThreads therefore cannot communicate with each other through shared data structures, even if they were all created by the same process. This restriction limits the use of distributed threads to applications where such communication and coordination is not required. Table 1 below highlights the commonalities and differences between local threads and AnekaThreads

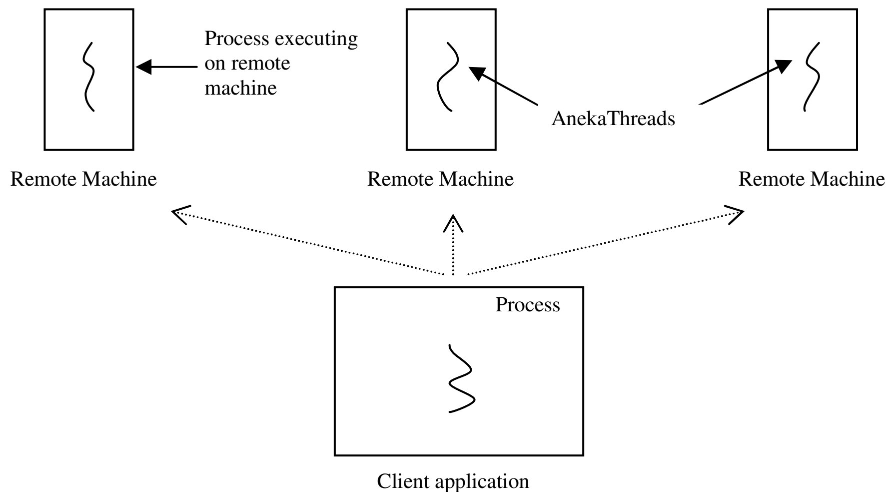


Figure 2: A process executing AnekaThreads on remote machines

| .Net Threading API | Aneka Threading API |
|---|---|
| System.Threading | Aneka.Threading |
| Thread | AnekaThread |

| | |
|---|---|
| Thread.ManagedThreadId (int) | AnekaThread.Id |
| Thread.Name | AnekaThread.Name |
| Thread.ThreadState (ThreadState) | AnekaThread.State |
| Thread.IsAlive | AnekaThread.IsAlive |
| Thread.IsRunning | AnekaThread.IsRunning |
| Thread.IsBackground | [Not provided] |
| Thread.Priority | [Not provided] |
| Thread.IsThreadPoolThread | [Not provided] |
| Thread.Start | AnekaThread.Start |
| Thread.Abort | AnekaThread.Abort |
| Thread.Sleep | [Not provided] |
| Thread.Interrupt | [Not provided] |
| Thread.Suspend | [Not provided] |
| Thread.Resume | [Not provided] |
| Thread.Join | AnekaThread.Join |

Table 1: Standard .Net Threads in comparison with AnekaThreads

### 3.1 Thread Synchronization

Learning to appreciate the differences between local and distributed threads will help you write more complex applications that utilize threads of both types. The .Net framework provides a number of synchronization primitives for controlling the interactions between local threads and avoiding race conditions, such as locking and signaling. The Aneka threading library on the other hand only supports a single synchronization mechanism using the *AnekaThread.Join* method as illustrated in figure 3 below.

Invoking AnekaThread's join method will cause the main application thread to block until the AnekaThread terminates by either completing successfully or failing. This basic level of synchronization can be useful in applications where the partial results of computations are required in order to proceed further. Since each of these threads execute in isolation completely independent of each other, and using with their own private data structures, no other forms of synchronization such as locking and signaling are necessary.

As the Aneka runtime environment is shared amongst a number of users, where multiple applications utilize the execution nodes, it is not possible to perform operations such as *Suspend*, *Resume*, *Interupt* and *Sleep* that may result in holding a resource indefinitely preventing from being used.

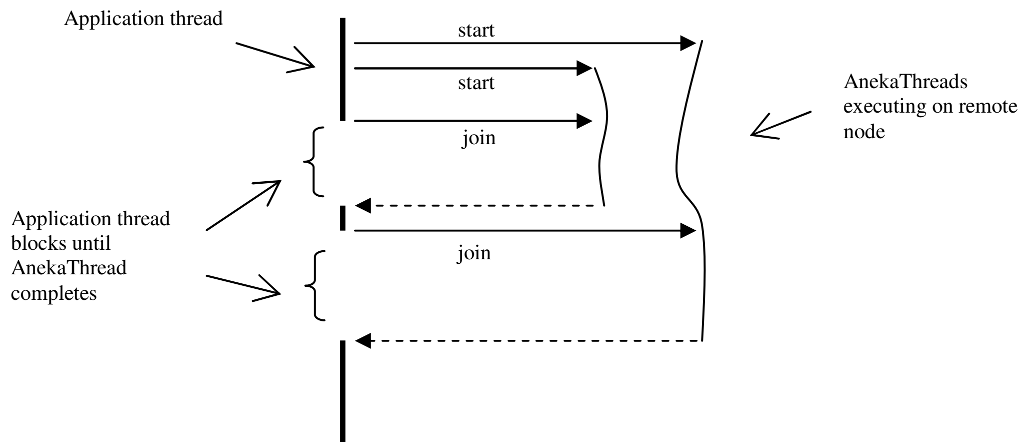


Figure 3: Basic synchronization provided by Aneka's threading library

### 3.2 Thread Priorities

.Net's Thread class supports thread priorities, where the scheduling priority can be one of Highest, AboveNormal, Normal, BelowNormal or Lowest. Operating systems are however not required to honor the priority of a thread. The current version of Aneka does not support thread priorities for AnekaThreads.

### 3.3 Thread Life-Cycle

The following diagram depicts the possible execution states for local threads supported by the .Net framework. A thread may, at any given time, be in one or more of these states. When a new thread is created its state is *Unstarted*, and transitions to *Running* when the Start method is invoked. There is also an additional state called *Background* which indicates whether a thread is running in the background or the foreground.

Figure 5 illustrates the life-cycle of an AnekaThread. As both thread types are fundamentally different, one being local and the other distributed, the possible states they take differ from instantiation to termination. An instance of AnekaThread transitions from *Unstarted* to *Started* when its Start method is invoked. It then transitions to *Queued* when it is scheduled for execution at a remote computing node. When execution begins, it state is *Running*, and finally transitions to *Completed* when all work is done. During any one of these stages, an AnekaThread may fail resulting in the *Failed* state. Other states such as *StagingIn* and *StagingOut* are used when a thread requires files for execution, and produces files as output. Programming threads that require or produce files is beyond the scope of this tutorial and you are encouraged to refer the online documentation for more details. Lastly, from your perspective as a programmer you only get to initiate the first state change from *Unstarted* to *Started*. Thereafter, all stage changes are carried out by the Aneka runtime environment.

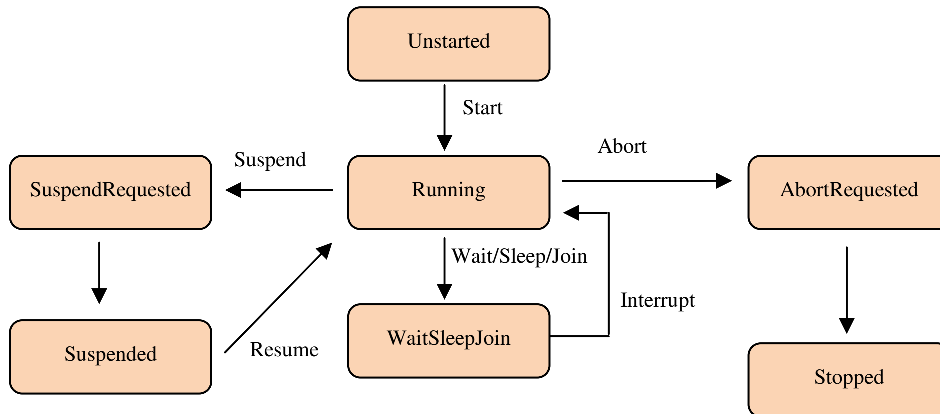


Figure 4: The life cycle of local threads in .Net

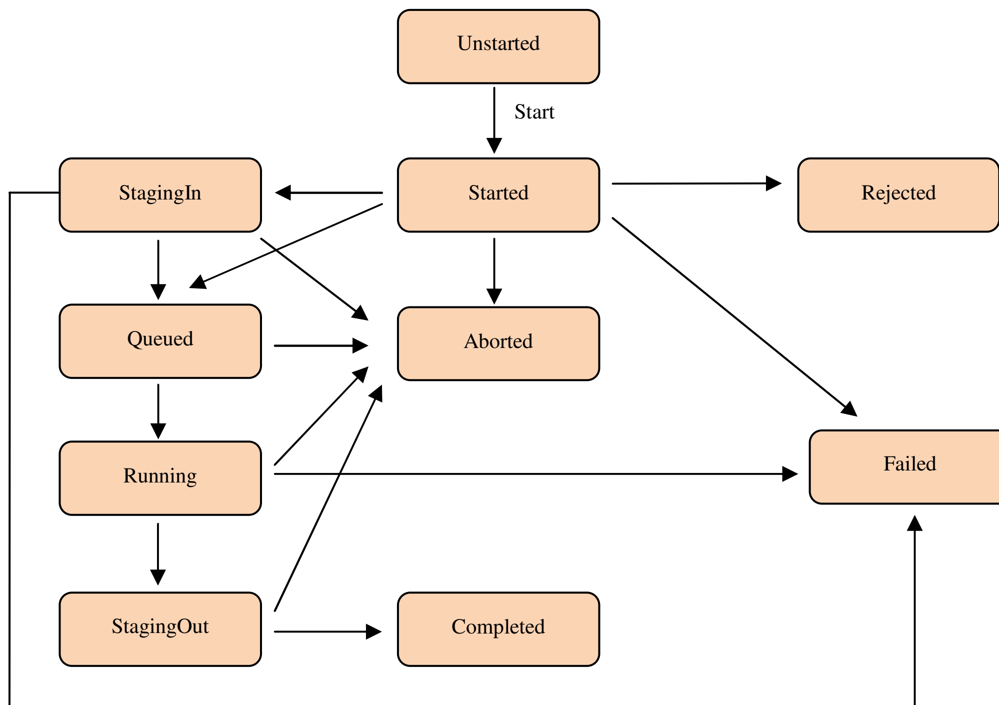


Figure 5: The life cycle of AnekaThreads

# 4 Developing Parallel Applications with AnekaThreads

Developing parallel applications requires an understanding the problem and its logical structure. Once you have figured this out, it is fairly easy to approach the solution. The following sections provide simple guidelines to follow when developing parallel applications with Aneka.

## 4.1 Problem Decomposition

One of the key challenges in developing parallel applications lies in breaking down a large problem into smaller units of work, such that they can be executed concurrently on different machines. Decomposing a problem might not seem very evident at first, but it is often a good idea to start with a piece of paper. Two common approaches used for problem decomposition are:

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

The first approach is the most common and involves identifying repetitive calculations in the problem. Often these take the form of *for* or *while* loops in a sequential program. Every iteration of the loop is thus potentially a unit of work that can be computed independently from other iterations. The two examples, calculating Pi and matrix multiplication, shown later in this tutorial uses this approach. If an iteration is dependent on the values produced in the previous iteration, then the units of work can no longer be computed independently and some form of communication is required. The following diagram illustrates this:

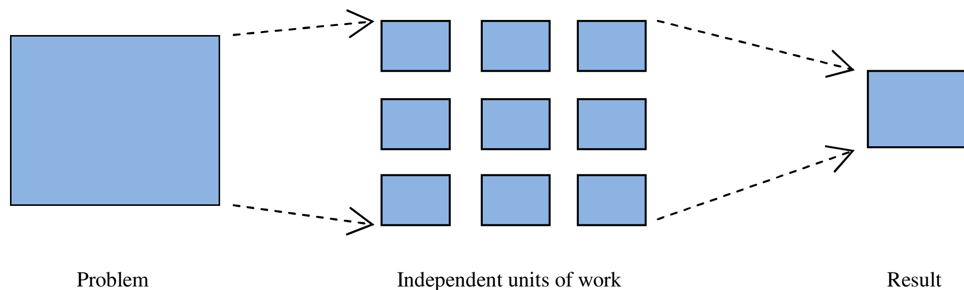


Figure 6: Exploiting repetitive calculations to break down a large problem in smaller units of work

The second approach involves identifying sufficiently large but isolated computations in the problem. Each of these distinct computations would then form a unit of work for concurrent execution. Unlike the first approach where each unit of work does the same amount of computation and would thus take more or less the same time to complete, the second approach involves distinct units of work, each of which make take significantly different amounts of time to complete. The first example program shown later in this tutorial, where the results of three trigonometric functions are

computed, uses this approach.

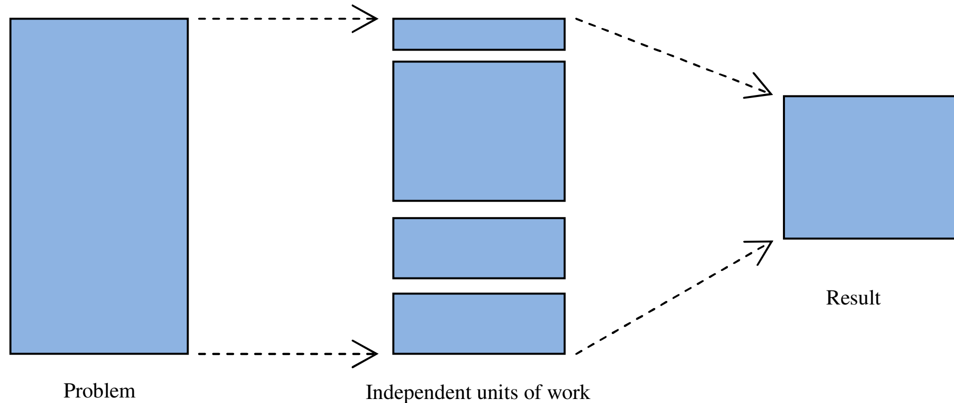


Figure 7: Each isolation computation can form a unit of work

### 4.2 Class Design and Serialization

Once you have decomposed the problem into smaller units of work, your next step is to design your classes. Ideally one your class should encapsulate the data and methods required to perform the computation on the remote node. This class must be annotated with the *Serializable* attribute to enable instances to be serialized and shipped to the Aneka runtime for execution.

```
[Serializable]
public class Work
{
   private int data1;
   private string data2;
   private int result;

   public int Result
   {
      get { return this.result; }
   }

   public Work(int data1, string data2)
   {
      this.data1 = data1;
      this.data2 = data2;
   }

   public void Compute()
   {
      // perform computation
      // and store result in
      // variable 'result'
   }
}
```

Figure 8: A sample class that encapsulates the work done by an AnekaThread

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

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

Figure 9: The constructor for AnekaThread takes a ThreadStart delegate as one of its parameters

Figure 9 above presents the constructor for AnekaThread, which takes a ThreadStart delegate and an instance of AnekaApplication as parameters. The delegate is a pointer to an instance method that will be executed on the remote node, and the AnekaApplication instance contains the required configuration to forward AnekaThreads to the Aneka runtime environment. Figure 11 illustrates the process of creating and starting an AnekaThread.

```
AnekaApplication<AnekaThread, ThreadManager> application = new
               AnekaApplication<AnekaThread, ThreadManager>(configuration);

Configuration configuration = new Configuration();
configuration.SchedulerUri = schedulerUri;
```

Figure 10: Creating and configuring an AnekaApplication instance

```
Work work = new Work(10, “hello world”);

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

Figure 11: Creating and starting an AnekaThread

## 5 A Parallel Math Program using AnekaThreads

The following program demonstrates the use of AnekaThreads to perform a simple mathematical computation. While the result might not be of any practical use, this program serves as a useful example to introduce programming with AnekaThreads. Consider the following mathematical equation:

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

As these trigonometric functions are independent operations, they can be executed in isolation. The only requirement is that after computation, the results of these operations much be combined to produce the final result, as illustrated in the figure below.

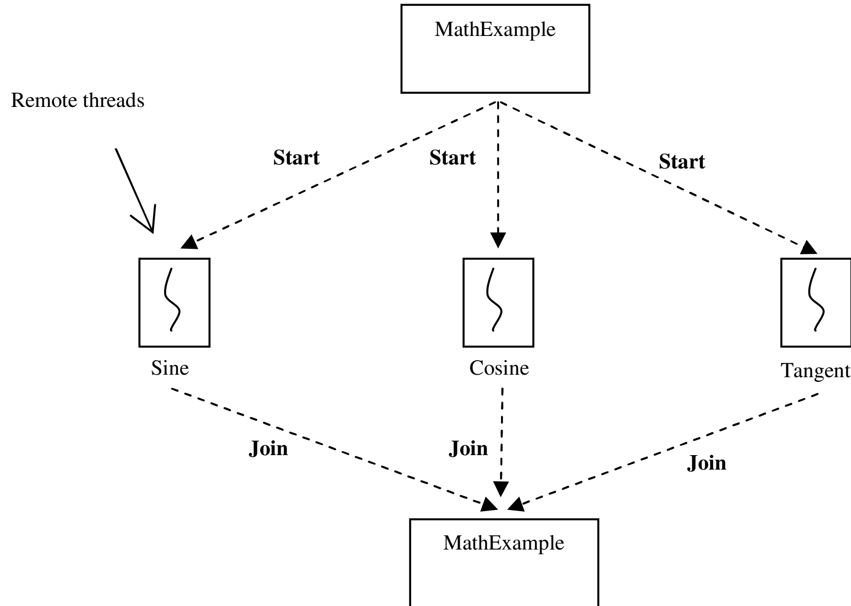


Figure 12: The flow control in an application using multiple distributed threads

The class MathExample is a single threaded client application that runs on the local machine. It spawns three AnekaThreads, each to compute the sin, cos and tan of a given angle. Each of these threads is then dispatched to remote compute nodes for concurrent execution, by invoking the *start* operation on the threads. Although MathExample can now continue execution, it needs to wait until all three threads have completed before it can combine their results. To do this, it invokes the *join* operation on each of the threads which causes MathExample to block until the threads have completed their operation on the remote node. The code listing below presents the complete solution.

**Program 1**

```
/// <summary>
/// Class Sine. Computes the sin value of an angle.
/// </summary>
[Serializable]
public class Sine
{
    /// <summary>
    /// The angle in degrees
    /// </summary>
    private double angle;

    /// <summary>
    /// The sin value of the angle
    /// </summary>
    private double result;

    /// <summary>
    /// Gets or sets the sin value of the angle
    /// </summary>
    public double Result
    {
        get { return result; }
        set { result = value; }
    }

    /// <summary>
    /// Creates and instance of the Sine class
    /// </summary>
    /// <param name="angle">The angle in degrees to convert</param>
    public Sine(double angle)
    {
        this.angle = angle;
    }

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



/// <summary>
/// Class Cosine. Computes the cos value of an angle.
/// </summary>
[Serializable]
public class Cosine
{
    /// <summary>
    /// The angle in degrees
    /// </summary>
    private double angle;
```

```
    /// <summary>
    /// The cos value of the angle
    /// </summary>
    private double result;

    /// <summary>
    /// Gets or sets the cos value of the angle
    /// </summary>
    public double Result
    {
        get { return result; }
        set { result = value; }
    }

    /// <summary>
    /// Creates and instance of Cosine class
    /// </summary>
    /// <param name="angle">The angle in degrees to convert</param>
    public Cosine(double angle)
    {
        this.angle = angle;
    }

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



/// <summary>
/// Class Tangent. Computes the tan of an angle.
/// </summary>
[Serializable]
public class Tangent
{
    /// <summary>
    /// The angle in degrees
    /// </summary>
    private double angle;

    /// <summary>
    /// The tan value of the angle
    /// </summary>
    private double result;

    /// <summary>
    /// Gets or sets the tan value of the angle
    /// </summary>
    public double Result
    {
        get { return result; }
        set { result = value; }
    }
```

```
    /// <summary>
    /// Creates and instance of the Tangent class
    /// </summary>
    /// <param name="angle">The angle in degrees to convert</param>
    public Tangent(double angle)
    {
        this.angle = angle;
    }

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

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

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

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

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

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

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

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

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

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

            // compute sum
            double sum = sine.Result + cosine.Result + tangent.Result;

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

Note that as with the standard .Net Thread class, the constructor of the AnekaThread class requires a ThreadStart delegate. That is, a pointer to a method that will be executed on the remote node. As discussed, the class to which this method belongs must be serializable. Instances of the class, containing all required data, are marshaled and shipped to the remote node for execution. The AnekaApplication instance acts as a client side gateway to the Aneka runtime environment, and forwards the AnekaThread instances for execution when their Start operation is invoked. The main thread of the MathExample waits until all remote threads are completed, by invoking the Join operation on the sinThread, cosThread and tanThread instances.

**Output**

```
Sum = 1.3347829113876
```

Note that each of the threads is executed independently, and the ordering of execution does not matter. Repeated runs of this program will always yield the same result. Any debug statements, such as printing to console, in the methods that compute sin, cos and tan will not appear on your console as they are executed on remote nodes.

## 6 Calculating Pi (π) Using a Dartboard

This example is a little more complex than the previous one, but produces a more useful result. It demonstrates the use of AnekaThreads for calculating the value of pi. This example is analogous to throwing darts at random points on a dartboard. Each thread independently approximates the value of pi using this approach, and the cumulative average computed from all threads is used as the overall approximation for pi. Increasing the number of AnekaThreads, improves the accuracy of pi.

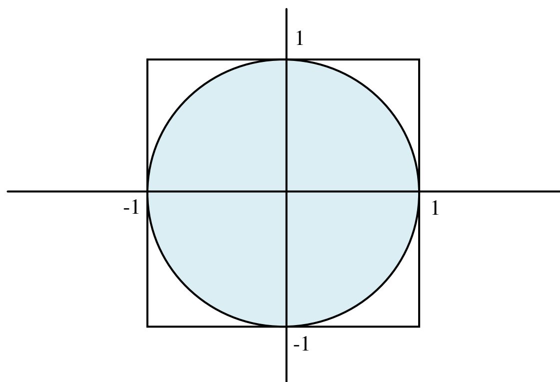


Figure 13: A circle inscribed in a square bounded by the coordinates (1,1), (1,-1), (-1,-1) and (-1,1)

The process of calculating pi involves throwing darts at random points within the square. A dart may thus land inside the circle or outside of it, but within the square. A dart that lands inside the circle is called a *hit*. The ratio of the area of the circle, $\pi r^2$, to the area of the square, $4 r^2$, is $\pi/4$. The x-y coordinates of the darts are random numbers in the range [-1, 1]. The value of pi is given by the following equation:

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

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

Each thread computes the value of pi by throwing darts a predetermined number of times.

Increasing the number of threads, will keep the work done by each thread the same, but will increase the overall accuracy of the cumulative average of pi. The following is the complete solution.

**Program 2**

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

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

    /// <summary>
    /// Gets or sets the approximation for
    /// the value of Pi.
    /// </summary>
    public double Result
    {
        get { return result; }
        set { result = value; }
    }

    /// <summary>
    /// Creates and instance of <see cref="T:Aneka.Examples.PiCalculator.Dart"/>
    /// </summary>
    /// <param name="iterations">The number of iterations to throw
    /// the dart at random points</param>
    public Dart(int iterations)
    {
        this.iterations = iterations;
    }

    /// <summary>
    /// Throws the dart at random points for the
    /// specified number of iterations.
    /// </summary>
    public void Fire()
    {
        int hit = 0;
        Random rand = new Random();

        for (int i = 0; i < this.iterations; i++)
        {
            double x = rand.NextDouble() * 2 - 1;
            double y = rand.NextDouble() * 2 - 1;
```

```
            if ((x * x) + (y * y) <= 1.0)
            {
                hit++;
            }
        }
        this.result = 4.0 * ((double)hit / (double)iterations);
    }
}

/// <summary>
/// Class Dartboard. Represents a dartboard to which a collection of darts
/// can be thrown. Each of the darts thrown in encapsulated in an AnekaThread
/// instance and executed on the remote runtime environment.
/// </summary>
public class Dartboard
{
    /// <summary>
    /// The application configuration
    /// </summary>
    private Configuration configuration;

    /// <summary>
    /// Creates an instance of Dartboard.
    /// </summary>
    /// <param name="schedulerUri">The uri to Aneka's scheduler
    /// </param>
    public Dartboard(Uri schedulerUri)
    {
        configuration = new Configuration();
        configuration.SchedulerUri = schedulerUri;
    }

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

        IList<AnekaThread> threads = this.CreateComputeThreads(application,
                                                  noOfDarts, iterations);

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


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

```
    // stop application
    application.StopExecution();

    return pi;
}

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

    for (int x = 0; x < noOfDarts; x++)
    {
        Dart dart = new Dart(iterations);
        AnekaThread thread = new AnekaThread(dart.Fire, application);
        threads.Add(thread);
    }

    return threads;
}

/// <summary>
/// Executes the list of AnekaThread instances
/// on the Aneka runtime environment.
/// </summary>
/// <param name="threads">The list of AnekaThread
/// instances to execute</param>
private void ExecuteThreads(IList<AnekaThread> threads)
{
    foreach (AnekaThread thread in threads)
    {
        thread.Start();
    }
}

/// <summary>
/// Composes the resulting value of pi by calculating the cumulative average
/// computed by each of the AnekaThread instances.
/// This method pauses until all threads have completed execution.
/// </summary>
/// <param name="threads">The list of instances that were submitted for
/// execution</param>
/// <returns>The average value of pi</returns>
private double ComposeResult(IList<AnekaThread> threads)
{
    // wait till all threads complete..
    foreach (AnekaThread thread in threads)
```

```
        {
            thread.Join();
        }

        double total = 0;

        foreach (AnekaThread thread in threads)
        {
            Dart dart = (Dart)thread.Target;
            total += dart.Result;
        }

        return total / threads.Count;
    }

    /// <summary>
    /// The main entry point to the application.
    /// </summary>
    /// <param name="args">Currently requires no command
    /// line arguments.</param>
    static void Main(string[] args)
    {
        Uri uri = new Uri("tcp://400w-ICT0217-09:9090/Aneka");

        Dartboard dboard = new Dartboard(uri);

        double pi = dboard.ThrowDarts(25, 3000);

        Console.WriteLine("Value of pi = " + pi);

        Console.ReadKey();
    }
}
```

The class Dartboard controls the execution of AnekaThread instances on remote nodes. The method ThrowDarts takes the number of darts and the repetitions per dart as parameters. An AnekaThread is created for each dart and executed on a remote node for the given number of repetitions. Note how a List is used to maintain the collection of AnekaThreads in the program. The method ExecuteThreads iterates through this list to start all threads, and the method ComposeResult iterates through the list to join all threads with the main application thread before calculating the cumulative average for pi.

**Output**

***Run 1:***

```
Value of pi = 3.13797333333333
```

***Run 2:***

```
Value of pi = 3.15202666666667
```

***Run 3:***

```
Value of pi = 3.13845333333333
```

Note that multiple runs of the program results in slightly different but nevertheless fairly accurate values for pi. You may increase this accuracy by either using more AnekaThreads, or increasing the work done by each AnekaThread.

## 7 Matrix Multiplication

A classic problem for parallel computing, matrix multiplication has many practical applications in numerous fields. The following example demonstrates matrix multiplication using AnekaThreads. While there are many strategies for decomposing the work across different threads, the method described below uses a two-dimensional decomposition:

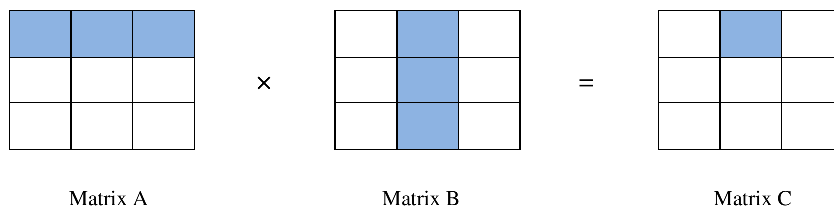


Figure 14: Multiplying two square matrices.

Each element of the resulting matrix, C, is computed by multiplying the corresponding row and column vectors of matrix A and matrix B. Each of these computations is carried out by a separate instance of AnekaThread on a remote node. Multiplying two square matrices of dimension $n$ will thus result in $n \times n$ AnekaThreads. The code listing below presents the solution.

**Program 3**

```
/// <summary>
/// Class Matrix. Represents a square matrix where
/// each element occupies a slot in a two-dimensional array.
/// </summary>
[Serializable]
public class Matrix
{
    /// <summary>
    /// Array of elements in the matrix.
    /// </summary>
    private double[,] data;

    /// <summary>
    /// Gets or sets the 2D array containting
    /// the elements in the matrix.
    /// </summary>
    public double[,] Data
    {
        get { return this.data; }
        set { this.data = value; }
    }
```

```
/// <summary>
/// The size of the square matrix.
/// </summary>
private int size;

/// <summary>
/// Gets or sets the size of the
/// square matrix.
/// </summary>
public int Size
{
    get { return this.size; }
    set { this.size = value; }
}

/// <summary>
/// Creates a new square matrix of dimension
/// <paramref name="size"/>
/// </summary>
/// <param name="size">
/// The dimension of square matrix.
/// </param>
public Matrix(int size)
{
    data = new double[size, size];
    this.size = size;
}

/// <summary>
/// Initializes the matrix with
/// random doubles between 0 to 10.
/// </summary>
public void InitRandom()
{
    Random rand = new Random();

    for (int x = 0; x < size; x++)
    {
        for (int y = 0; y < size; y++)
        {
            data[x, y] = rand.NextDouble() * 10;
        }
    }
}

/// <summary>
/// Prints the elements in the matrix
/// to the console.
/// </summary>
public void Print()
{
    for (int x = 0; x < size; x++)
    {
        for (int y = 0; y < size; y++)
        {
            Console.Write(data[x, y].ToString("0.00"));

            if (y < size - 1)
            {
                Console.Write(", ");
```

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


/// <summary>
/// Class RowColumnMultiplier. Multiplies a row matrix with a
/// column matrix to produce a matrix with one element.
/// </summary>
[Serializable]
public class RowColumnMultiplier
{
    /// <summary>
    /// A row matrix to multiply
    /// </summary>
    private double[] row;

    /// <summary>
    /// A column matrix to multiply
    /// </summary>
    private double[] column;

    /// <summary>
    /// The result of row-column multiplication
    /// </summary>
    private double result;

    /// <summary>
    /// Gets the result of the row-column
    /// multiplication
    /// </summary>
    public double Result
    {
        get { return this.result; }
    }

    /// <summary>
    /// Creates a new RowColumnMultiplier
    /// </summary>
    /// <param name="row">The row to multiply</param>
    /// <param name="column">The column to multiply</param>
    public RowColumnMultiplier(double[] row, double[] column)
    {
        this.row = row;
        this.column = column;
    }

    /// <summary>
    /// Multiplies the row and column matrices
    /// </summary>
    public void DoMultiply()
    {
        // row and column are of the same dimension
        for (int x = 0; x < row.Length; x++)
        {
            this.result += row[x] * column[x];
        }
    }
```

```
}



/// <summary>
/// Class MatrixMultiplier. Multiplies two square matrices, where each element
/// in the resulting matrix, C, is computed by multiplying the corressponding
/// row and column
/// </summary>
public class MatrixMultiplier
{
    /// <summary>
    /// The application configuration
    /// </summary>
    private Configuration configuration;

    /// <summary>
    /// Creates an instance of MatrixMultiplier
    /// </summary>
    /// <param name="schedulerUri">The uri to the Aneka scheduler</param>
    public MatrixMultiplier(Uri schedulerUri)
    {
        configuration = new Configuration();
        configuration.SchedulerUri = schedulerUri;
    }

    /// <summary>
    /// Multiplies two matrices A and B and returns the resulting matrix C. This
    /// method creates a list of AnekaThread instances to compute each of the
    /// elements in Matrix C. These threads are submitted to the Aneka runtime
    /// for execution and the results of each of these executions are used to
    /// compose the resulting matrix C.
    /// </summary>
    /// <param name="matrixA">Matrix A</param>
    /// <param name="matrixB">Matrix B</param>
    /// <returns>The result, Matric C</returns>
    public Matrix Multiply(Matrix matrixA, Matrix matrixB)
    {
        // Create application and computation threads
        AnekaApplication<AnekaThread, ThreadManager> application = new
                   AnekaApplication<AnekaThread, ThreadManager>(configuration);

        IList<AnekaThread> threads = this.CreateComputeThreads(application,
                                                                   matrixA, matrixB);

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

        // gather results
        Matrix matrixC = this.ComposeResult(threads, matrixA.Size);

        // stop application
        application.StopExecution();

        return matrixC;
    }

    /// <summary>
    /// Creates AnekaThread instances to compute each of the
    /// elements in the resulting matrix C. These threads are initalized to
```

```
/// execute RowColumnMultiplier’s DoMultiply method on the remote node.
/// </summary>
/// <param name="application">The AnekaApplication instance containing the
/// application configuration</param>
/// <param name="matrixA">Matrix A</param>
/// <param name="matrixB">Matric B</param>
/// <returns>The result, Matrix C</returns>
private IList<AnekaThread> CreateComputeThreads(
              AnekaApplication<AnekaThread, ThreadManager> application,
               Matrix matrixA, Matrix matrixB)
{
    IList<AnekaThread> threads = new List<AnekaThread>();

    int dimension = matrixA.Size;

    for (int row = 0; row < dimension; row++)
    {
        double[] rowData = this.ExtractRow(matrixA.Data, row, matrixA.Size);

        for (int column = 0; column < dimension; column++)
        {
            double[] columnData = this.ExtractColumn(matrixB.Data, column,
                                                     matrixB.Size);

            RowColumnMultiplier rcMultiplier = new RowColumnMultiplier
                                                      (rowData, columnData);
            AnekaThread anekaThread = new AnekaThread
                                     (rcMultiplier.DoMultiply, application);
            threads.Add(anekaThread);
        }
    }

    return threads;
}

/// <summary>
/// Executes the list of AnekaThread instances on the Aneka runtime
/// environment.
/// </summary>
/// <param name="threads">The list of AnekaThread instances to
/// execute</param>
private void ExecuteThreads(IList<AnekaThread> threads)
{
    foreach (AnekaThread thread in threads)
    {
        thread.Start();
    }
}

/// <summary>
/// Composes the resulting matrix C. This method pauses until all elements
/// of matrix C have been computed. The results of each of these executions
/// are then used to compose matric C.
/// </summary>
/// <param name="threads">The list of AnekaThread instances that were
/// submitted for execution</param>
/// <param name="size">The size of the resulting matrix C</param>
/// <returns>The result, Matrix C</returns>
private Matrix ComposeResult(IList<AnekaThread> threads, int size)
{
    // wait till all threads complete..
```

```
    foreach (AnekaThread thread in threads)
    {
        thread.Join();
    }

    // compose resultant matrix
    Matrix matrixC = new Matrix(size);
    for (int row = 0; row < size; row++)
    {
        for(int column = 0; column < size; column++)
        {
            AnekaThread thread = threads[(row * size) + column];
            RowColumnMultiplier rcMultiplier =
                                        (RowColumnMultiplier)thread.Target;
            matrixC.Data[row, column] = rcMultiplier.Result;
        }
    }

    return matrixC;
}

/// <summary>
/// Extracts a row from a two-dimensional array.
/// </summary>
/// <param name="array">The two-dimensional array</param>
/// <param name="rowIndex">The index of the row to extract</param>
/// <param name="length">The length of the row to extract</param>
/// <returns>A one-dimensional array</returns>
private double[] ExtractRow(double[,] array, int rowIndex, int length)
{
    double[] row = new double[length];
    for (int x = 0; x < length; x++)
    {
        row[x] = array[rowIndex, x];
    }
    return row;
}

/// <summary>
/// Extracts a column from a two-dimensional array.
/// </summary>
/// <param name="array">The two-dimensional array<</param>
/// <param name="columnIndex">The index of the column to extract</param>
/// <param name="length">The length of the column to extract</param>
/// <returns>A one-dimensional array</returns>
private double[] ExtractColumn(double[,] array, int columnIndex, int length)
{
    double[] column = new double[length];
    for (int x = 0; x < length; x++)
    {
        column[x] = array[x, columnIndex];
    }
    return column;
}

/// <summary>
/// The main entry point to the application
/// </summary>
/// <param name="args"></param>
static void Main(string[] args)
{
```

```
            Matrix matrixA = new Matrix(10);
            Matrix matrixB = new Matrix(10);

            matrixA.InitRandom();
            matrixB.InitRandom();

            Uri schedulerUri = new Uri("tcp://localhost:9090/Aneka");

            MatrixMultiplier multiplier = new MatrixMultiplier(schedulerUri);
            Matrix matrixC = multiplier.Multiply(matrixA, matrixB);
            matrixC.Print();

            Console.ReadKey();
        }
    }
```

Instances of the class Matrix are used to represent the square matrices A, B and C. The elements are stored in a two-dimensional array and can be initialized with random values between 0 and 10. The method DoMultiply in class RowColumnMultiplier carries out the actual computation of an element in the resulting matrix C, by multiplying a single row and column in matrices A and B. Note that this class implements the serializable interface allowing instances to be marshaled and shipped to the execution nodes in the Aneka runtime environment, where they will be executed as an AneakThread. Finally, the class MatrixMultiplier is responsible for decomposing the problem into smaller units work by creating AnekaThreads to compute each of the elements in the resulting matrix C. The following is the output generated as a result of multiple two matrices of dimension 10x10, each initialized with random numbers.

**Output**

```
190.47, 194.68, 119.38, 64.72,  158.61, 90.76,  145.67, 150.65, 99.46,  178.52
285.89, 264.89, 266.50, 133.85, 247.26, 133.39, 203.49, 188.56, 192.71, 283.68
303.57, 324.57, 353.44, 170.24, 308.25, 223.62, 304.64, 185.23, 226.89, 423.50
154.51, 124.76, 125.75, 77.30,  144.43, 91.86,  110.69, 79.06,  86.55,  157.74
207.68, 156.19, 156.86, 114.02, 213.76, 116.95, 123.23, 132.47, 131.18, 223.53
251.74, 258.54, 231.65, 134.63, 246.85, 170.58, 202.56, 165.76, 170.45, 292.08
203.49, 186.40, 153.71, 69.45,  150.81, 103.40, 164.84, 136.66, 93.65,  212.21
335.58, 267.99, 192.95, 126.12, 238.13, 118.63, 145.27, 231.51, 183.32, 285.00
215.20, 274.21, 225.08, 128.53, 258.47, 169.15, 206.73, 210.35, 214.25, 285.23
273.68, 214.88, 264.08, 168.33, 299.64, 172.47, 196.59, 165.18, 175.61, 319.92
```

# 8 Summary

This tutorial introduced you to Aneka's Thread Programming Model. We looked at some of the fundamental concepts in both local and distributed thread programming. AnekaThreads provide a similar interface to the local Thread class in .Net and allows you to program in much the same manner. Distributed threads are however fundamentally different, and it is important to understand the tradeoffs when writing parallel applications.

# 9 Exercises

### *Objective Questions*

1. A ______ is an instance of a program in execution. A ______ is a dispatchable unit of work that can be schedule by the CPU.
2. An AnekaThread is a distributed thread that executes on a ______ machine.
3. The ______ method in the AnekaThread class starts the thread.
4. The ______ method in the AnekaThread class joins the thread to the main application thread.
5. A process can contains more than one local thread at the same time: True or False.
6. A process can contains more than one distributed AnekaThread at the same time: True or False.
7. AnekaThreads support thread priorities: True or False.
8. The only synchronization mechanism supported by AnekaThreads is through the ______ method.
9. An AnekaThread requires a _____ delegate as one of its parameters.
10. Unlike distributed threads, local threads can coordinate their work with each other using synchronization mechanisms such as locking and _____.

### *Review Questions*

11. What are threads? Briefly explain the differences between a thread and a process.
12. Briefly explain the differences between local and distributed threads.
13. What are the challenges in developing parallel applications?
14. Explain how you would go about designing and developing a parallel application.
15. What are the limitations of distributed threads compared to local threads?

### *Programming Problems*

16. Modify the example for calculating Pi so that the number of darts and repetitions are passed as command line arguments to the program.
17. Modify the example for matrix multiplication so that the size of the square matrix is passed as a command line argument to the program.
18. Remodel and implement the example for trigonometric calculations, such that MathExample functions as a multi-threaded server that listens to computation requests from clients via sockets, and forwards them to the Aneka runtime for execution.
19. Improve the example for matrix multiplication to multiply matrices with different, but compatible dimensions.

### *Group Project*

20. Mandelbrot sets are known as iterative fractals, which when graphically rendered produces and image as shown below. Study the literature on Mandelbrot sets and implement a program to render its graphical representation using AnekaThreads.

.

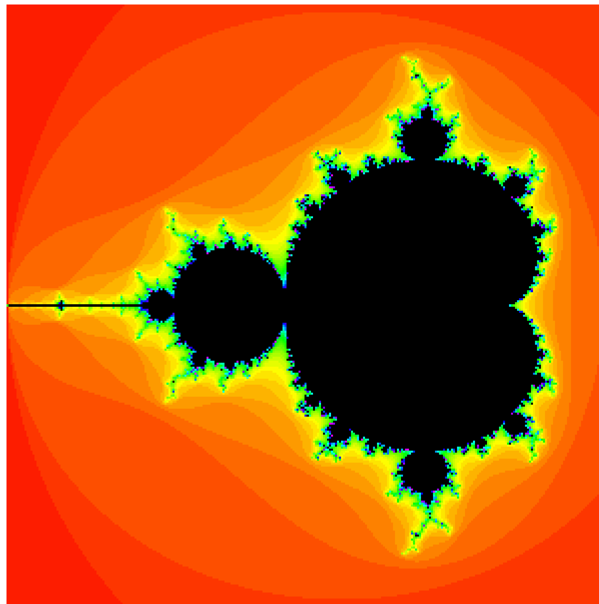

Figure 15: Graphical rendering of the Mandelbrot set.