CHAPTER 6-Concurrent Computing

Thread Programming

- Throughput computing is realized by means of multiprocessing and multithreading.
- Multiprocessing- execution of multiple programs in a single machine
- Multithreading - 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.
- The Aneka Thread Programming Model will be taken as a reference model to review a practical implementation of a multithreaded model for computing clouds.

6.1 Introducing parallelism for single-machine computation

- Parallelism has been a technique for improving the performance of computers.
- Asymmetric multiprocessing involves the concurrent use of different processing units that are specialized to perform different functions. (ex-GPU)
- Symmetric multiprocessing features the use of similar or identical processing units to share the computation load. (ex- multicore technology)
- Other examples are

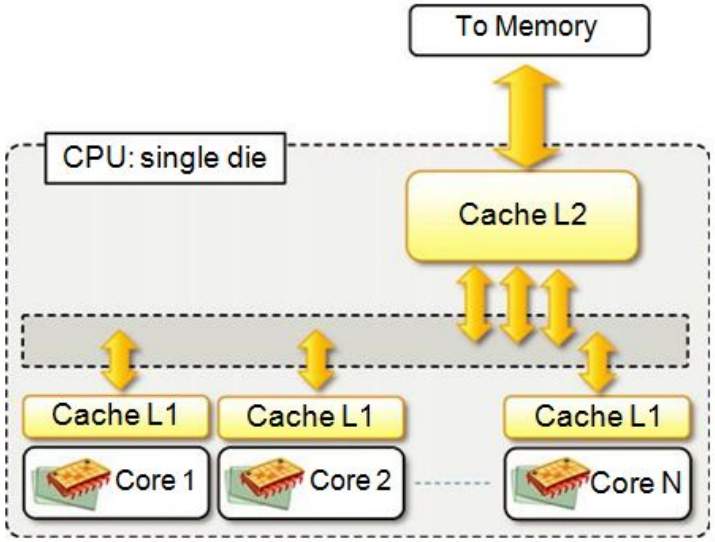
  * Nonuniform memory access (NUMA) - define a specific architecture for accessing a shared memory between processors

  * Clustered multiprocessing, the use of multiple computers joined together as a single virtual

6.1 Introducing parallelism for single-machine computation

- 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. Multicore processor.
- Multicore technology has been used as a support for processor design and also in GPUs and network devices

•Multiprocessing is just one technique that can be used to achieve parallelism



6.1 Introducing parallelism for single-machine computation

- A process is the runtime image of an application, or 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.
- Almost all the commonly used operating systems support multitasking and multithreading.

6.2 Programming applications with threads

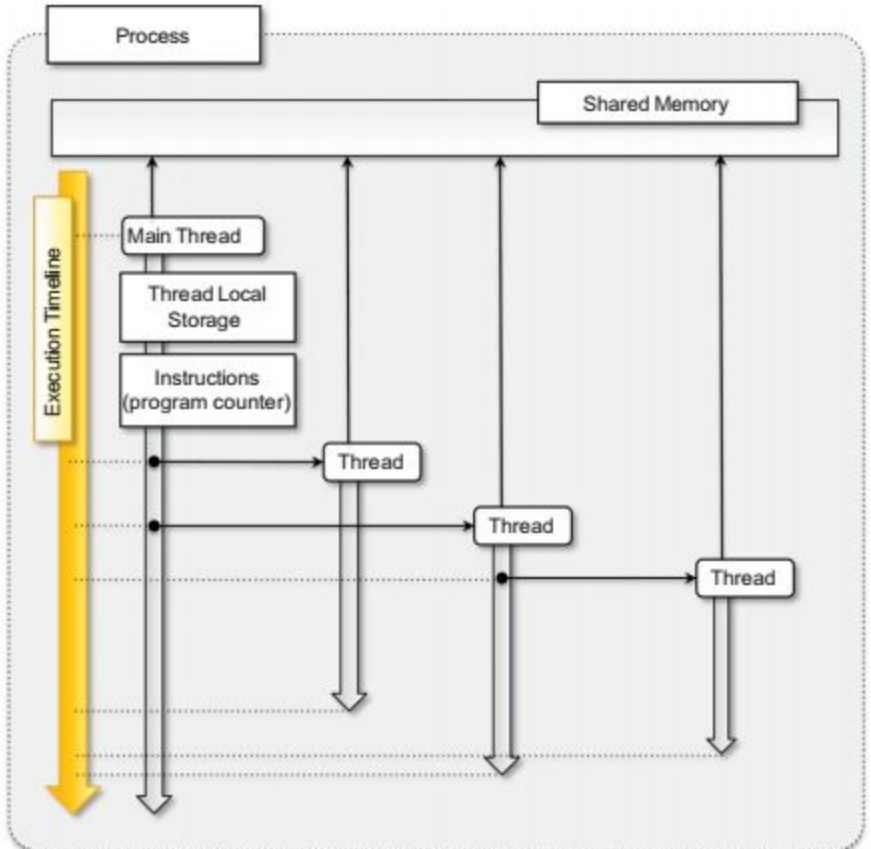
- Developers organize programs in terms of threads in order to express intra process concurrency. The use of threads might be implicit or explicit.
- *Implicit threading* happens when the underlying APIs use internal threads to perform specific tasks supporting the execution of applications such as graphical user interface (GUI) rendering.
- *Explicit threading* is characterized by the use of threads within a program by application developers, who use this abstraction to introduce parallelism.
- **6.2.1 What is a thread?**
- A thread identifies a single control flow, which is a logical sequence of instructions(designed to be executed one after the other one.) within a process.
- 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, OS assigns different time slices to each process and interleaves their execution.
- Context switch - 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

- A running program is identified by a process, which contains at least one thread, also called the mainthread. Such a thread is implicitly created by the compiler or the runtime environment executing the program.

- This thread last for the entire lifetime of the process and be the origin of other threads. As main threads, these threads can spawn other threads.

- 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.

- **Fig- Relationship b/w processes & thread**



• 6.2.2 Thread APIs

•*6.2.2.1 POSIX Threads*

•Portable Operating System Interface for Unix (POSIX) is a set of standards related to the application programming interfaces for a portable development of applications over the Unix operating system flavors.

•Standard POSIX 1.c addresses the implementation of threads and the functionalities that should be available for application programmers to develop portable multithreaded applications.

•The POSIX standard defines the following operations:

1.creation of threads with attributes, 2.termination of a thread, and 3.waiting for thread completion (join operation).

- 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,
- Different synchronization abstractions are provided to solve different synchronization problems.

6.2.2.2 Threading support in java and .NET

- Languages such as Java and C# provide a rich set of functionalities for multithreaded programming by using an object-oriented approach. Since both Java and .NET execute code on top of a virtual machine, the APIs exposed by the libraries refer to managed or logical threads.
- These are mapped to physical threads by the runtime environment in which programs developed with these languages execute.
- Both Java and .NET express the thread abstraction with the class Thread exposing the common operations performed on threads: start, stop, suspend, resume, abort, sleep, join, and interrupt.
- *Start and stop/abort* are used to control the lifetime of the thread instance.
- *Suspend and resume* are used to programmatically pause and then continue the execution of a thread.
- *Sleep* – allows pausing the execution of a thread for a predefined period of time.
- Join- makes one thread wait until another thread is completed.
- These waiting states can be interrupted by using the interrupt operation, which resumes the execution of the thread.
- Implementing mutexes, critical regions, and reader-writer locks are completely covered by means of the basic class libraries or additional libraries in

->java (java.util.concurrent package,) and ->.net(.NET Parallel Extension framework)

6.2 Programming applications with threads

6.2.3 Techniques for parallel computation with threads

- Decomposition is a useful technique that aids in understanding whether a problem is divided into components (or tasks) that can be executed concurrently.
- It also provides a starting point for a parallel implementation, since it allows the breaking down into independent units of work that can be executed concurrently with the support provided by threads.
- The two main decomposition partitioning techniques are *domain and functional decompositions.*
- *6.2.3.1 Domain decomposition*
- Domain decomposition is the process of identifying patterns of functionally repetitive, but independent computation on data.
- This is the most common type of decomposition in the case of throughput computing, and it relates to the identification of repetitive calculations required for solving a problem.
- When these calculations are identical, only differ from the data they operate on, and can be executed in any order, the problem is said to be embarrassingly parallel(do not share any data).

6.2 Programming applications with threads

•*6.2.3.1 Domain decomposition(cond)*

The master-slave model is a quite common organization for these scenarios:

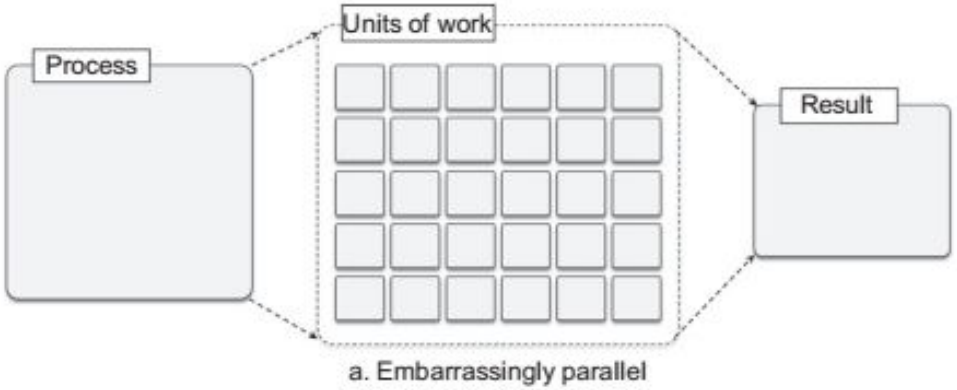
•The system is divided into two major code segments.

- 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.

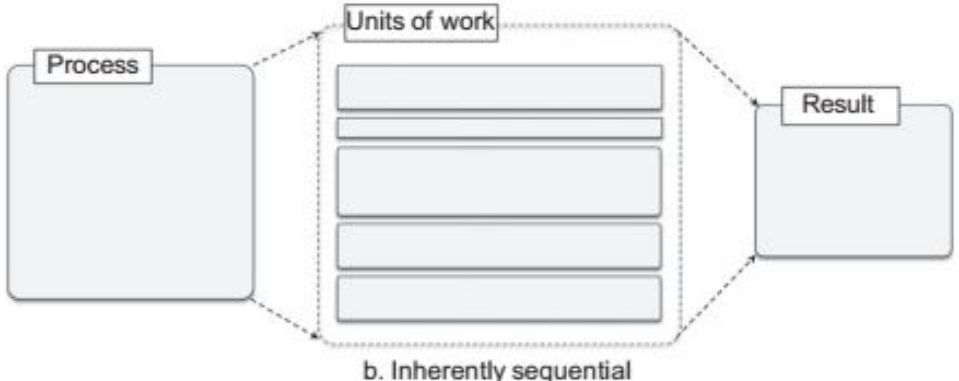
•In general, a while or a for loop is used to express the decomposition logic, and each iteration generates a new unit of work to be assigned to a slave thread.

•Figure 6.3 provides a schematic representation of the decomposition of embarrassingly parallel and inherently sequential problems.

- *6.2.3.1 Domain decomposition(cond)*
- *Embarrassingly parallel problems- isolate an independent unit of work.*
- *Inherently sequential-if the values of all the iterations are dependent on some of the values obtained in the previous iterations.(grouping into single computation-dependent iterations)*
- *Fig-Domain decomposition techniques*



a. Embarrassingly parallel



b. Inherently sequential

•6.2.3.1 Domain decomposition(cond)

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

•Matrix multiplication is a binary operation that takes two matrices and produces another matrix as a result.

•*There are several techniques for performing matrix multiplication; among them, the matrix product is the most popular. Figure 6.4 provides an overview of how a matrix product can be performed.*

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

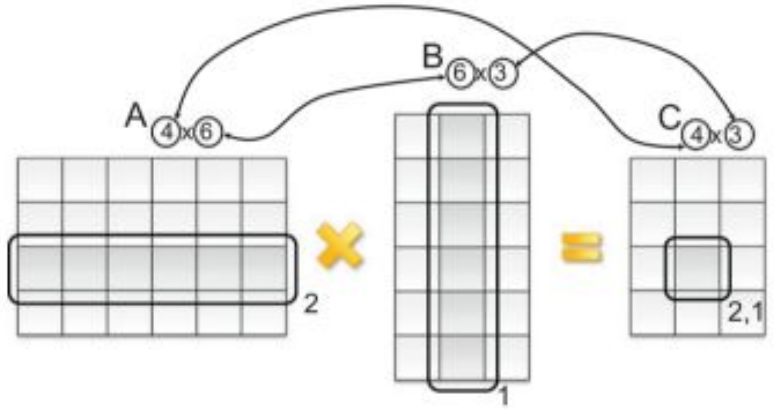
$$C_{ij} = \sum_{k=0}^{n-1} A_{ik} B_{kj}$$

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

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

•6.2.3.1 Domain decomposition(cond)

•Given these conditions, the resulting matrix will have the number of rows of the first matrix and the number of columns of the second matrix,



•Repetitive operation is the computation of each of the elements of the resulting matrix. These are subject to the same formula, and the computation does not depend on values that have been obtained by the computation of other elements of the resulting matrix.

1.Define a function that performs the computation of the single element of the resulting matrix by implementing the previous equation.

2.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.

3.Join all the threads for completion, and compose the resulting matrix.

•6.2.3.1 Domain decomposition(cond)

- 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.
- Such a delegate must reference a defined method in some class. Hence, we can define a simple class that exposes as properties the row and the column to multiply and the result value. This class will also define the method for performing the actual computation.
- Listing 6.1 shows the class ScalarProduct - read the matrices from the standard input or from a file and concentrate our attention on the main control logic that decomposes the computation, creates threads, and waits for their completion in order to compose the resulting matrix.

- *6.2.3.1 Domain decomposition(cond)*

- ***Scalar Product***

```
public class ScalarProduct
{
        private double result;
        private double[] row, column;
        public ScalarProduct(double[] row, double[] column)
        {
        this.row = row;
        this.column = column;
        }
        public void Multiply()
        {
        this.result = 0;
              for(int i=0; i<this.row.Length; i++)
              {
              this.result += this.row[i] * this.column[i];
              }
         }}
```

•*6.2.3.1 Domain decomposition(cond) Matrix **Product***

```
using System; using System.Threading; using System.Collections.Generic;
public class MatrixProduct
{
        private static double[,]a, b;
        private static double[,] c;
/// Dictionary mapping the thread instances to the corresponding ScalarProduct instances that are run
 inside.
        private static IDictionary<Thread, ScalarProduct>workers.
public static void Main(string[] args)
{
        MatrixProduct.ReadMatrices();
        MatrixProduct.ExecuteThreads();
        MatrixProduct.ComposeResult();
}
private static void ExecuteThreads()
{
MatrixProduct.workers = newList<Thread>();
int rows = MatrixProduct.a.Length;
```

- *6.2.3.1 Domain decomposition(cond) Matrix **Product***

```
int columns = MatrixProduct.b.Length;
for(int i=0; i<rows; i++)
for(int j=0; j<columns; j++)
{
double[] row = MatrixProduct.a[i];
double[] column = new double[column];
for(int k=0; k<column; k++)
{
column[j] = MatrixProduct.b[j][i];
}
}
ScalarProduct scalar = newScalarProduct(row, column);
Thread worker = newThread(newThreadStart(scalar.Multiply));
worker.Name =string.Format("{0}.{1}",row,column); worker.Start();
MatrixProduct.workers.Add(worker, scalar);
}
}
```

•*6.2.3.1 Domain decomposition(cond) Matrix **Product***

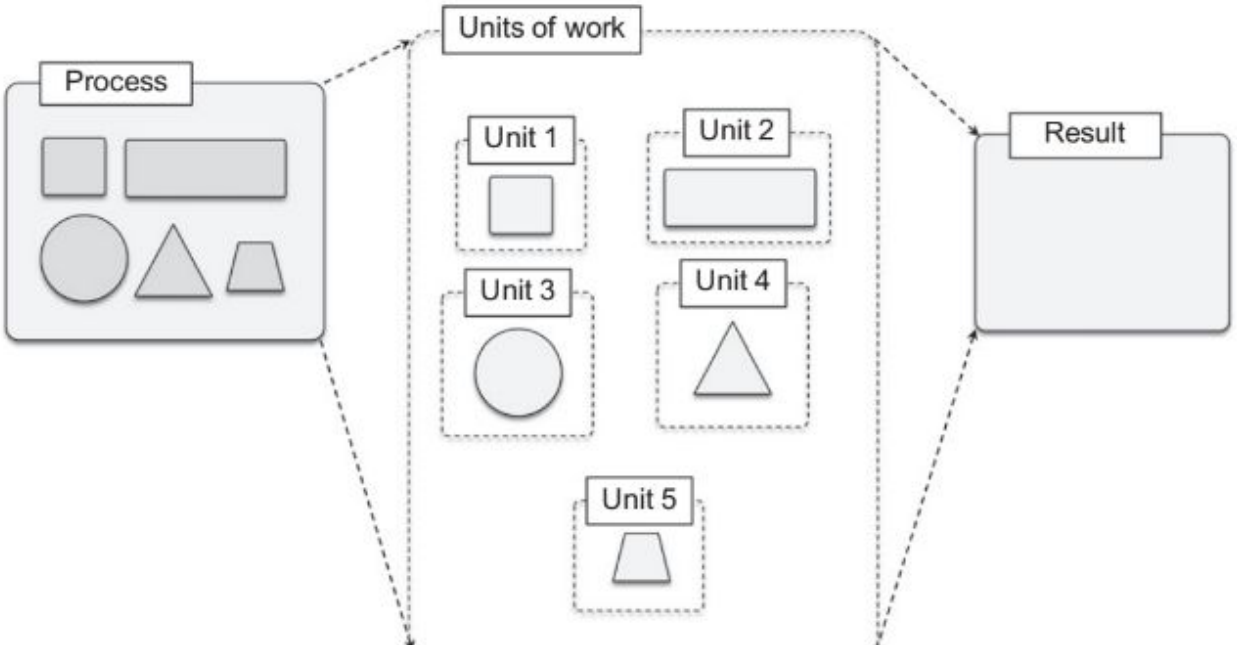
```
 private static void ComposeResult()
{       MatrixProduct.c=new double[rows,columns];
        foreach(KeyValuePair<Thread,ScalarProduct>pair in MatrixProduct.workers)
        {
        Thread worker = pair.Key;
        string[] indices = string.Split(worker.Name, new char[] {'.'});
        int i = int.Parse(indices[0]);
        int j = int.Parse(indices[1]);
        worker.Join();
        MatrixProduct.c[i,j] = pair.Value.Result;
        }
MatrixProduct.PrintMatrix(MatrixProduct.c);
}
private static void ReadMatrices()
{ // code for reading the matrices a and b }
private static void PrintMatrices(double[,] matrix)
{ // code for printing the matrix. }
}
```

•6.2.3.2 Functional decomposition

•Functional decomposition is the process of identifying functionally distinct but independent computations. The focus here is on the type of computation rather than on the data manipulated by the computation.

•This kind of decomposition is less common and does not lead to the creation of a large number of threads, since the different computations that are performed by a single program are limited.

•Functional decomposition leads to a natural decomposition of the problem in separate units of work because it does not involve partitioning the dataset, but the separation among them is clearly defined by distinct logic operations.



•6.2.3.2 Functional decomposition(contd)

•Suppose, for example, that we need to calculate the value of the following function for a given value of x:

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

•The program computes the sine, cosine, and tangent functions in three separate threads and then aggregates the results.

•This technique introduces a synchronization problem that is properly handled with the lock statement in the method referenced by the function pointer. The lock statement creates a critical section that can only be accessed by one thread at time and guarantees that the final result is properly updated.

•6.2.3.2 Functional decomposition(contd) Mathematical function

```
using System; using System.Threading;using System.Collections.Generic;
public delegate void UpdateResult(double x);
public class Sine
{
private double x;
private double 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.
public Sine(double x, UpdateResultupdater)
{
this.x = x;
this.updater = updater;
}
public void Apply()
{
this.y = Math.Sin(this.x);
if (this.updater != null)
{
this.updater(this.y);}}}
```

•6.2.3.2 Functional decomposition(contd)

```
public class Cosine
{
private double x;
private double y;
private UpdateResultupdater;
/// Creates an instance of the Cosine and sets the input to the given angle.
public Cosine(double x, UpdateResultupdater)
{
this.x = x;
this.updater = updater;
}
public void Apply()
{
this.y = Math.Cos(this.x);
if (this.updater != null)
{
this.updater(this.y);
}
}
}
```

•6.2.3.2 Functional decomposition(contd)

```
public class Tangent
{
private double x;
private double y;
private UpdateResultupdater;
/// Creates an instance of the Tangent and sets the input to the given angle.

public Tangent(double x, UpdateResultupdater)
{
this.x = x;
this.updater = updater;
}
public void Apply()
{
this.y = Math.Tan(this.x);
if (this.updater != null)
{
this.updater(this.y);
}
}
}
```

•6.2.3.2 Functional decomposition(contd)

```
public class Program
{
private static double result;
/// Synchronization instance used to avoid keeping track of the threads.
private static object synchRoot = new object();
public static void Main(string[] args)
{
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);
```

•6.2.3.2 Functional decomposition(contd)

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

•6.2.3.3 Computation vs. communication

•The two decomposition methods presented in this section and the corresponding sample applications are based on the assumption that the computations are independent. This means that:

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

•These two assumptions strongly simplify the implementation and allow achieving a high degree of parallelism and a high throughput.

•Having all the worker threads independent from each other gives the maximum freedom to the operating system (or the virtual runtime environment) scheduler in scheduling all the threads.

•The need to exchange data among different threads introduces dependencies among them and ultimately can result in introducing performance bottlenecks.

•For example, we did not introduce any queuing technique for threads; but queuing threads might potentially constitute a problem for the execution of the application if data need to be exchanged with some threads that are still in the queue.

•6.2.3.3 Computation vs. communication

- A more common disadvantage is the fact that while a thread exchanges data with another one, it uses some kind of synchronization strategy that might lead to blocking the execution of other threads.
- The more data that need to be exchanged, the more they block threads for synchronization, thus ultimately impacting the overall throughput.
- As a general rule of thumb, it is important to minimize the amount of data that needs to be exchanged while implementing parallel and distributed applications.
- The lack of communication among different threads constitutes the condition leading to the highest throughput.

6.4 Programming applications with Aneka threads

6.4.1 Aneka threads application model

- The Thread Programming Model is a programming model in which the programmer creates the units of work as Aneka threads.
- Therefore, it is necessary to utilize the AnekaApplication <W,M>class, which is the application reference class for all the programming models falling into this category.
- To develop distributed applications with Aneka threads, it is necessary to specialize the template type as follows:

  AnekaApplication <AnekaThread, ThreadManager >

- This will be the class type for all the distributed applications that use the Thread Programming Model. These two types are defined in the Aneka.Threading namespace noted in the Aneka.Threading.dll library of the Aneka SDK.
- Another important component of the application model is the Configuration class, which is defined in the Aneka.Entity namespace (Aneka.dll).
- This class contains a set of properties that allow the application class to configure its interaction with the middleware.

6.4 Programming applications with Aneka threads

6.4.1 Aneka threads application model(contd)

- Configuration class properties

  1. the address of the Aneka index service, which constitutes the main entry point of Aneka Clouds;
  2. the user credentials required to authenticate the application with the middleware;
  3. some additional tuning parameters and
  4. an extended set of properties that might be used to convey additional information to the middleware.

- The code presented in next slide(listing 6.4) demonstrates how to create a simple application instance and configure it to connect to an Aneka Cloud whose index service is local.

- Once the application has been created, it is possible to create threads by specifying the reference to the application and the method to execute in each thread, and the management of the application execution is mostly concerned with controlling the execution of each thread instance.

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

6.4 Programming applications with Aneka threads

Coding for Application creation and Configuration:

```
using System;
using System.Collections.Generic;
using Aneka;
using Aneka.Util;
using Aneka.Entity;
using Aneka.Threading;
private AnekaApplication<AnekaThread,ThreadManager> CreateApplication();
{
Configuration conf =new Configuration();
conf.SchedulerUrl = newUri("tcp://localhost:9090/Aneka");
conf.Credentials =newUserCredentials("Administrator", string.Empty);
// we will not need support for file transfer,
conf.UseFileTransfer = false;
// we create the application instance and configure it.
AnekaApplication<AnekaThread,ThreadManager> app =
                        new AnekaApplication<AnekaThread,ThreadManager>(conf);
return app;
```

6.4 Programming applications with Aneka threads

Coding for Thread creation and Execution:

```
private void WorkerMethod()
{
// .....
}
///Creates a collection of threads
private void CreateThreads(AnekaApplication<AnekaThread,ThreadManager> app);
{
// creates a delegate to the method to execute inside the threads.
ThreadStart worker = newThreadStart(this.WorkerMethod);
// iterates over a loop and creates ten threads.
for(int i=0; i<10; i++)
{
AnekaThread thread = new AnekaThread(worker, app);
thread.Start();
}
}
```

END OF CHAPTER 6