Parallelism

Many personal computers and workstations have two or four cores (that is, CPUs) that enable multiple threads to be executed simultaneously. Computers in the near future are expected to have significantly more cores. To take advantage of the hardware of today and tomorrow, you can parallelize your code to distribute work across multiple processors. In the past, parallelization required low-level manipulation of threads and locks. Visual Studio 2010 and the .NET Framework 4 enhance support for parallel programming by providing a new runtime, new class library types, and new diagnostic tools. These features simplify parallel development so that you can write efficient, fine-grained, and scalable parallel code in a natural idiom without having to work directly with threads or the thread pool. The following illustration provides a high-level overview of the parallel programming architecture in the .NET Framework 4.

Parallelism – Data

Data parallelism refers to scenarios in which the same operation is performed concurrently (that is, in parallel) on elements in a source collection or array.

```csharp
// Sequential version
foreach (var item in sourceCollection)
{
    Process(item);
}

// Parallel equivalent
Parallel.ForEach(sourceCollection, item => Process(item));
```
When a parallel loop runs, the TPL partitions the data source so that the loop can operate on multiple parts concurrently. Behind the scenes, the Task Scheduler partitions the task based on system resources and workload. When possible, the scheduler redistributes work among multiple threads and processors if the workload becomes unbalanced.

**Parallelism - Task**

The Task Parallel Library (TPL), as its name implies, is based on the concept of the task. The term task parallelism refers to one or more independent tasks running concurrently. A task represents an asynchronous operation, and in some ways it resembles the creation of a new thread or ThreadPool work item, but at a higher level of abstraction. Tasks provide two primary benefits:

- More efficient and more scalable use of system resources.
  
  Behind the scenes, tasks are queued to the ThreadPool, which has been enhanced with algorithms (like hill-climbing) that determine and adjust to the number of threads that maximizes throughput. This makes tasks relatively lightweight, and you can create many of them to enable fine-grained parallelism. To complement this, widely-known work-stealing algorithms are employed to provide load-balancing.

- More programmatic control than is possible with a thread or work item.
  
  Tasks and the framework built around them provide a rich set of APIs that support waiting, cancellation, continuations, robust exception handling, detailed status, custom scheduling, and more.

```csharp
Parallel.Invoke(() => DoSomeWork(), () => DoSomeOtherWork());
```

**Parallelism – Coordination**

Typically, you would wait for a task for one of these reasons:

- The main thread depends on the final result computed by a task.
- You have to handle exceptions that might be thrown from the task.

```csharp
Task[] tasks = new Task[3]
{
    Task.Factory.StartNew(() => MethodA()),
    Task.Factory.StartNew(() => MethodB()),
    Task.Factory.StartNew(() => MethodC())
};

//Block until all tasks complete.
Task.WaitAll(tasks);
// Continue on this thread...
```