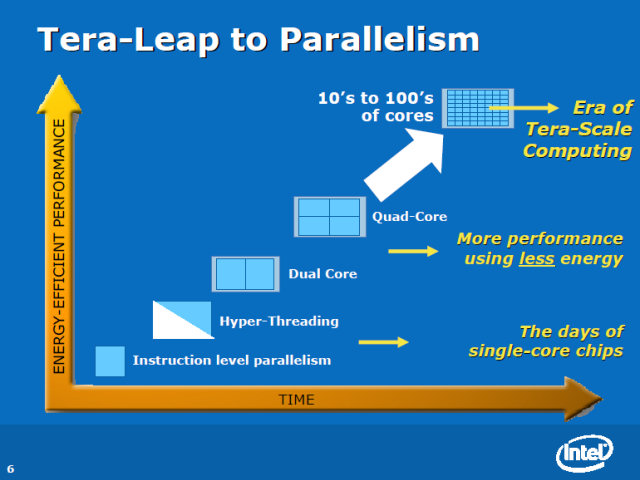
|  |  |
| --- | --- |
| PFX  **PARALLEL PROGRAMMING** Data parallelism and task parallelism Programming to leverage multicores or multiple processors called *parallel programming*. This is a subset of the broader concept of multithreading. | Simplifying the process of adding parallelism and concurrency to applications────More efficient and more scalable use of system resources.────More programmatic control than is possible with a thread or work item────Delegate ….. |

The new age is of asynchronous programming. Why is it so important?

Computers and other technology originally began with single-core processors; in the early 2000s, Intel, AMD and several other manufacturers altered the history of computing forever by pushing multi core processors on the market.

Over time, the multi-core processor evolved from dual core to tri, quad, hex and octa core designs (or processor chips with 2, 3, 4, 6, or 8 processors). Some processors now hold dozens or hundreds of cores. As individual CPU gates grow, question is the code utilizing platform capabilities or not?

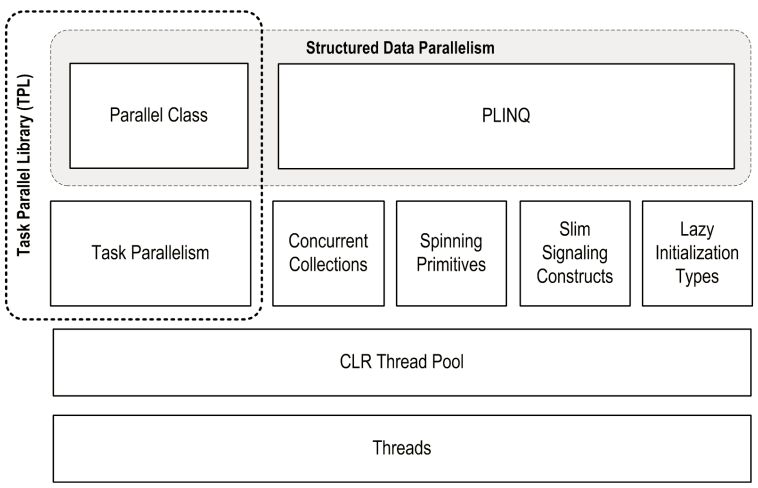
PFX Concepts

In the .NET, a slew of new support has been added to handle common needs in parallel programming, to help developers tackle the difficult problem that is programming for multi-core. The primary goal for PFX is *parallel programming* i.e. leveraging multicore processors to speed up computationally intensive code. There are two strategies for work partitioning, data parallelism and task parallelism.

When a set of tasks must be performed on many data values, we can parallelize by having each thread perform the (same) set of tasks on a subset of values. This is called data parallelism because we are partitioning the *data* between threads. In contrast, with task parallelism we partition the *tasks*; in other words, we have each thread perform a different task.

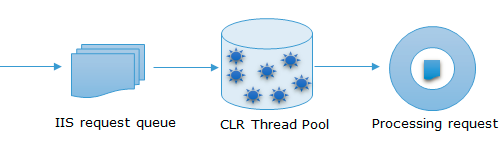
In general, data parallelism is easier and scales better to highly parallel hardware, because it reduces or eliminates shared data (thereby reducing contention and thread-safety issues). Also, data parallelism leverages the fact that there are often more data values than discrete tasks, increasing the parallelism potential.

PFX Components

PLINQ offers the richest functionality: it automates all the steps of parallelization — including partitioning the work into tasks, executing those tasks on threads, and collating the results into a single output sequence.

It is called *declarative* — because you simply declare that you want to parallelize your work (which you structure as a LINQ query), and let the Framework take care of the implementation details.

[PLINQ](http://www.albahari.com/threading/part5.aspx#_PLINQ) and the [Parallel](http://www.albahari.com/threading/part5.aspx#_The_Parallel_Class) class are useful whenever you want to execute operations in parallel and then wait for them to complete (structured parallelism). This includes non-CPU-intensive tasks such as calling a web service.

Task parallelism is the lowest-level approach to parallelization with PFX. A task avoids the overhead of starting a dedicated thread by using the CLR’s [thread pool](http://www.albahari.com/threading/#_Thread_Pooling): this is the same thread pool used by ThreadPool.QueueUserWorkItem, tweaked in CLR 4.0 to work more efficiently with Tasks (and more efficiently in general).

Tasks can be used whenever you want to execute something in parallel. However, they are tuned for leveraging multicores: in fact, the [Parallel](http://www.albahari.com/threading/part5.aspx#_The_Parallel_Class) class and [PLINQ](http://www.albahari.com/threading/part5.aspx#_PLINQ) are internally built on the task parallelism constructs. The Task Parallel Library lets you create hundreds (or even thousands) of tasks with minimal overhead. But, if you want to create millions of tasks, you’ll need to partition those tasks into larger work units to maintain efficiency. The Parallel class and PLINQ do this automatically.

Parallel extensions is to achieve data Parallelism. 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. It provides new constructs to achieve data parallelism by using Parallel.For, Parallel.Invoke (Executes an array of delegates in parallel) and Parallel.Foreach constructs.

|  |  |  |
| --- | --- | --- |
| Task | Parallel | PLINQ |
| Task parallelism we partition the tasks; in other words each thread perform a different task.  For I/O operation Tasks are (still) threads and async is not parallelTask.Delay doesn't need any actual CPU time; it's just like setting a timer to go off in the futureA task returned by Task.Run() really is saying "I want you to execute this code separately"; the exact thread on which that code executes depends on a number of factors.Reduces the context switching time among multiple threads in comparison of Thread. | When a set of tasks must be performed on many data values, we can parallelize by having each thread perform the (same) set of tasks on a subset of values. We partitioning the data between threads. Execution takes place in parallel way for data computation  Parallel.Invoke: Executes an array of delegates in parallel  Parallel.For : Performs the parallel equivalent of a C# for loop  Parallel.ForEach: Performs the parallel equivalent of a C# foreach loop, uses multiple Threads.  Parallel.ForEach is defined in .Net 4.0 and above frameworks.  You can pass a ParallelOptions object, you can use ParallelLoopState and you can use the overload that calls an Init and Finish function once for each thread. ParallelOptions allows you to do three things: *1. You can set its MaxDegreeOfParallelism property to set the maximum number of threads that Invoke will use 2. You can set its TaskScheduler property, if you have written your own scheduler, which will then be used to decide which delegate to run when and on which thread 3. You can set a CancellationToken*  When one of the delegates throws an exception, Invoke will still execute all other delegates in its array. All the exceptions thrown by the delegates are bundled in an AggregateException, which Invoke will throw when it finishes. | PLINQ automatically parallelizes local LINQ queries.  offloads the burden of both work partitioning and result collation  PLINQ may also operate sequentially if it suspects that the overhead of parallelization will actually slow a particular query.  PLINQ is only for local collections: it doesn’t work with LINQ to SQL or Entity Framework because in those cases the LINQ translates into SQL which then executes on a database server. However, you can use PLINQ to perform additional local querying on the result sets obtained from database queries.  PLINQ to be useful there has to be a reasonable amount of computationally intensive work for it to farm out to worker threads. Most LINQ to Objects queries execute very quickly, and not only would parallelization be unnecessary, but the overhead of partitioning, collating, and coordinating the extra threads may actually slow things down.  PLINQ will give unreliable results if the query invokes thread-unsafe methods. |
| * async/await is about asynchrony, whereas Parallel.ForEach is about parallelism. They are related concepts, but not the same. * Parallel.ForEach is used when you want to execute the same operation on all the items in a collection, in parallel, blocking the current thread until all operations have completed. * Async/await is particularly useful in two situations:   + Writing code which basically kicks off various asynchronous operations, one after the other, on the UI thread, accessing the UI briefly between operations. You don't want to block the UI thread, but managing all of those asynchronous operations is a pain otherwise.   + Handling lots of long-running operations at a time - e.g. in a web server. Each individual request may take a long time due to calling into other web service, databases etc - but you don't want to have one thread per request, as threads are a relatively expensive resource. | | |
|  | | |

Why Tasks? Why not threads?

The creation of a thread comes with a huge cost. Creating a huge number of Threads within your application also comes with an overhead of Context Switching. In a single core environment, it might lead to a bad performance as well, since we have a single core which serves various threads. Thread represents an actual OS-level thread, with its own stack and kernel resources.

Task does not create its own OS thread. Instead, tasks are executed by a [TaskScheduler](http://msdn.microsoft.com/en-us/library/dd997402.aspx); the default scheduler simply runs on the ThreadPool. The task on the other hand, dynamically calculates if it needs to create different threads of execution or not. It uses the ThreadPool under the hood, in order to distribute the work, without going through the overhead of Thread creation/or un-necessary context switching if not required.

Summary

PLINQ – Completing the query parallel and collating the results into a single output sequence automatically

TASK: For I/O Operation

Parallel: For Parallel way *f*or data computation, collating the results manually.