Introducing Visual Studio 2010/2012 .NET 4.X Task Parallel Library

CodeValue

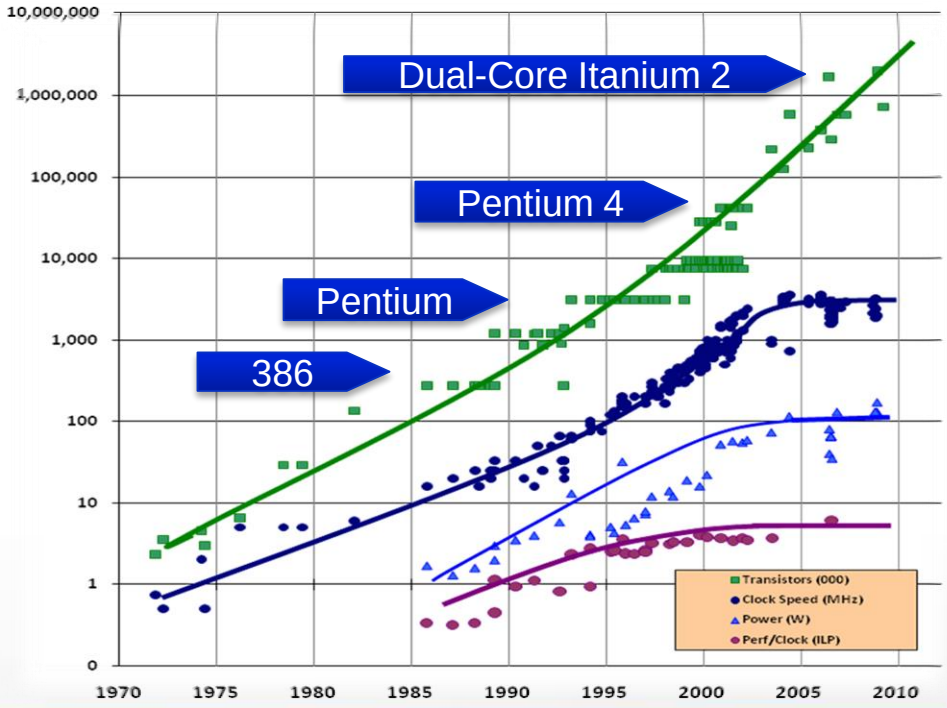
THE TASK PARALLEL LIBRARY

Agenda

- Overview of:
  - **The need for new concurrency abstraction**
  - **.NET 4.X Concurrency**
    - PLINQ
    - PPL
    - Coordinated Data Structure
    - C# 5.0 async/await keywords

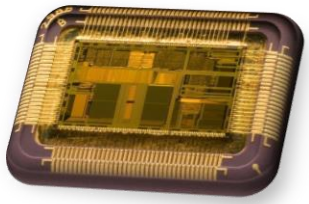
The Free Lunch is Over

- At the beginning of 2005 Herb Sutter had an article stating that the developer's free lunch is over
- We had an assumption that more transistors in the CPU implies better application execution speed

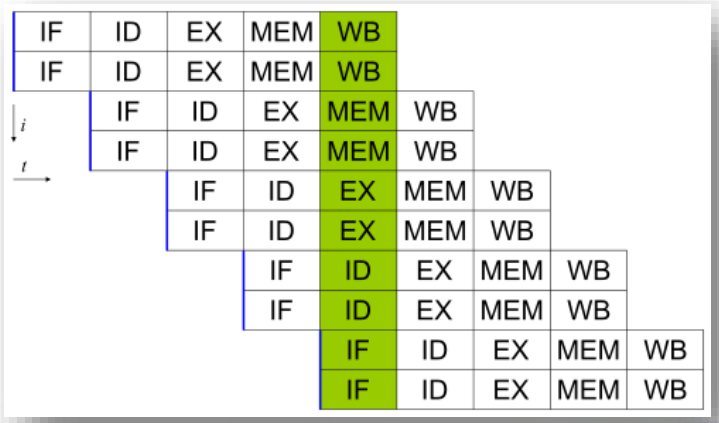


Intel CPU Trends
(sources: Intel, Wikipedia, K. Olukotun)

The Single-Core CPU



- The modern CPU is a very complicated piece of hardware
  - **It is the engine behind the scenes**
  - **Fetch instructions, translate, execute and store**
- Single core CPU parallelism
  - **Super-Scalar pipe line**
    - runs several commands in a sequential pipe-line
  - **Hyper-Threading**
    - Duplication of state storage and arbitration



The CPU Abstraction

- From the software point-of-view:
  - **The CPU executes the code in a sequential manner**
  - **The performance of the CPU-bound code is directly related to CPU frequency**
- This used to be our “Free Lunch”
  - **An old program runs faster on a new CPU**
- Using this assumption with modern low power consumption multi-core CPUs
  - **An old program runs slower on a new CPU**

Physical Limits & The Moore Low

- Why don't we have a 10GHZ CPU?
  - **There are physical limits (heat, power consumption, switching time, etc.)**
- Moore's low
  - **"The number of transistors that can be placed inexpensively on an integrated circuit has doubled approximately every two years"**
- What can be done with these transistors to improve application performance?

Consolidation Trends

- Having the ability to have more transistors in a chip made it possible to consolidate different functionalities into the CPU
  - **Early CPUs consolidated the Floating Point Unit**
  - **Bigger and Bigger Cache**
  - **Memory Controllers**
  - **GPU**
- Hyper-Threading is the result of adding CPU state storage to gain two physical execution paths

The Multi-Core CPU

- The next level of consolidation was to move the Multi-Process motherboard into the CPU
  - **Since cores are closer to each other, the cooperation between cores is better, resulting in faster execution**
  - **Cores share memory controllers and the last level cache**
- Modern CPUs (like the Core i* family) have several cores in one chip and a NUMA architecture

The Need for a New Abstraction

- Since performance is no longer tied to CPU frequency we need to leverage parallelism
- To some extent this is not a new concept
  - **Multiprocessing systems exist for years**
    - Windows NT was designed back in 1989 as a Symmetric Multiprocessing Operating System
  - **Hyper threading CPUs exist at least half a decade**
- But do we get better performance by moving from one-way machine to 16-ways?
- We need a better abstraction

Getting Back Our Free Lunch

- Our design can be based on multi-threading
  - **We can align the number of threads to the number of cores**
    - Does this mean that if we double the cores, we double the performance?
- Parallelism is hard, instead of inventing the underlying abstraction, we want to use existing well-known ones
- .NET 4.0 introduces new parallelism mechanisms and abstractions

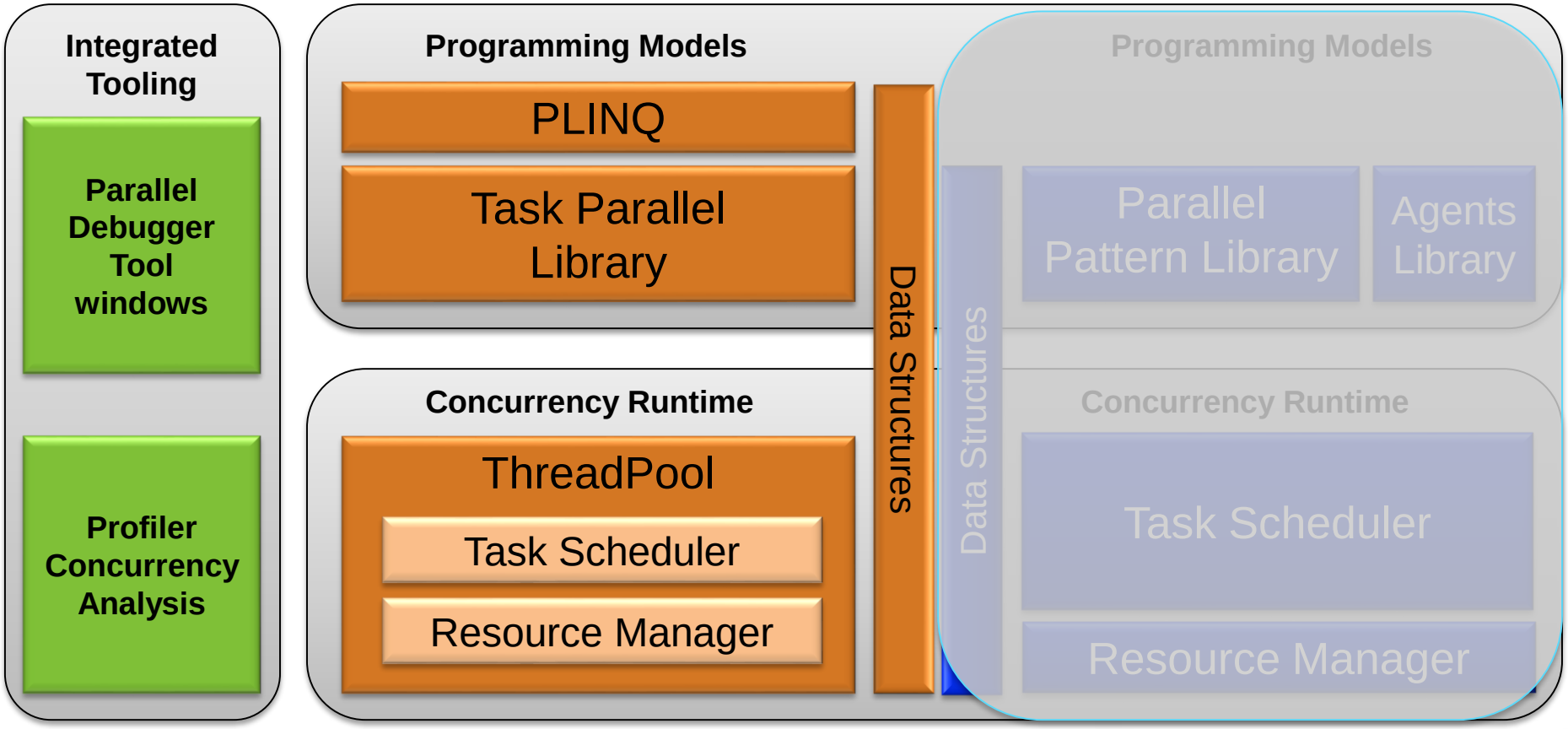
System.Threading | Parallel Extensions

Unified Cancellation Model

DEMO

Getting Back Our Free Lunch

Parallelism in Visual Studio 2010



Parallel Extensions

- A .NET Library that supports:
  - **Declarative data parallelism**
    - Parallel LINQ (PLINQ)
  - **Imperative task parallelism**
    - Task Parallel Library (TPL)
  - **A set of data structures that make coordination easier**
    - Coordination Data Structures (CDS)
  - **Task Dataflow (.NET 4.5)**

Task Parallel Library

- An Abstraction layer for easy parallelism
  - **Handling partitioning of the work, scheduling threads, cancelation and state management**
- Dynamically Scaling of Parallelism to maximize CPU cores utilization
- TPL should be the preferred ways to write parallel code
  - **If it meets your needs**

The Task class

- The Atom of Asynchronous Operations
  - **Create task with the `Task` constructor for deferred use**
    - Use the `Start` method to schedule the Task
  - **Create and schedule new task using the `TaskFactory.StartNew`**
    - `Task.Run` in .NET 4.5
  - **Use `Task<TResult>` to get a future return value**
- A Task can be in several states
  - **Use one of the `IsCanceled`, `IsCompleted`, `IsFaulted` or the `Status` property to get the Task state**
- To synchronize with Task execution end, use one of the `Wait` methods
- Use the AsyncState to have Task local storage

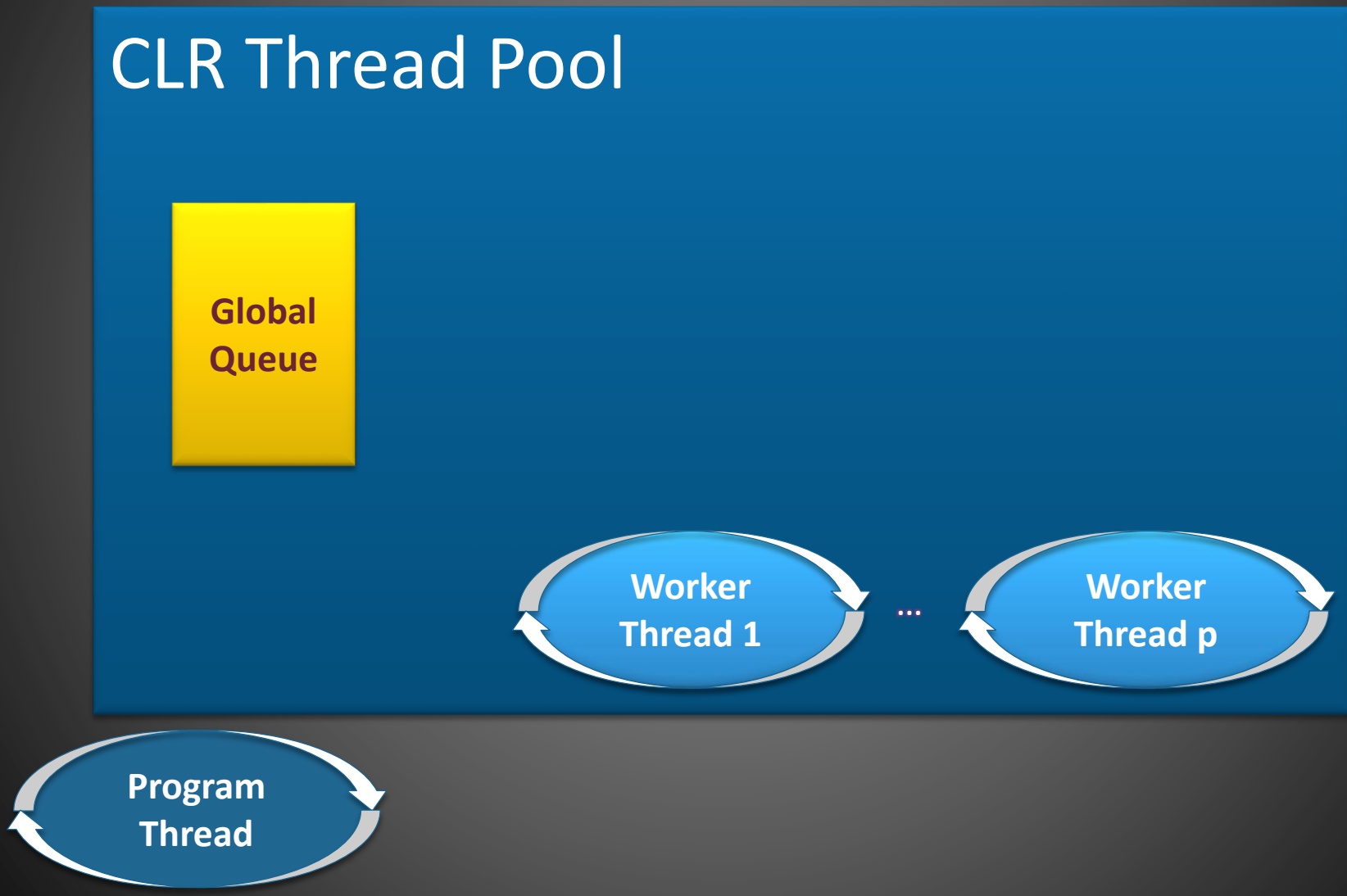
DEMO

Working With Tasks – Parallel QSort

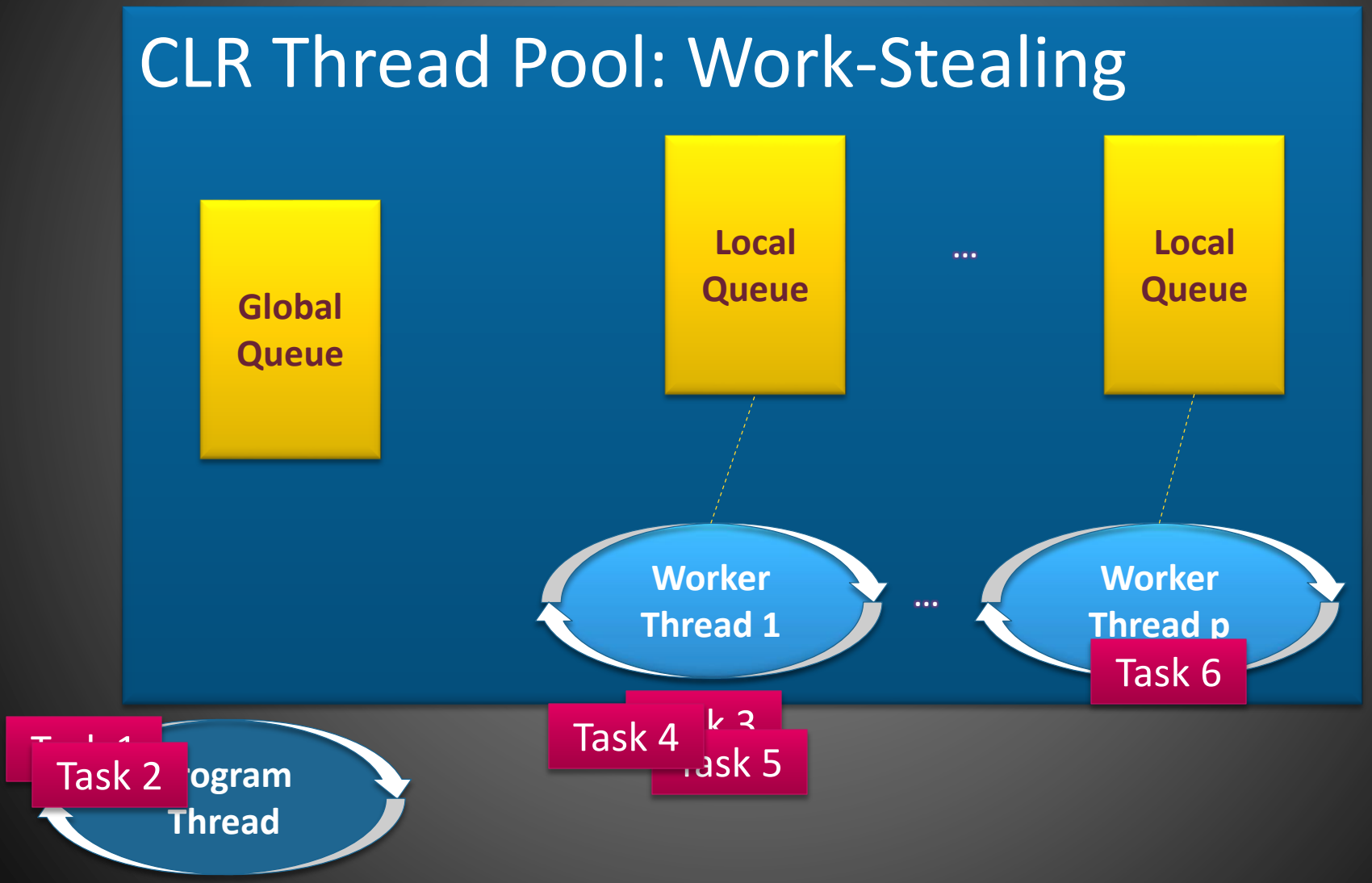
The Task Scheduler

- Task scheduler executes tasks
- The default task scheduler is based on a new ThreadPool and a Work Stealing Queues/Algorithm
- When scheduling a task using Task Factory, the task goes to the global queue
- When scheduling Task using the Start method or as a child task (Task Creation Option) the task is scheduled to a thread local queue
- Look here for a sample of a custom Task Scheduler
- Many Task related methods take an instance of a Task Scheduler

User Mode Scheduler



User Mode Scheduler For Tasks



Task Creation Options

- Specifies flags that control optional behavior for the creation and execution of tasks

Member name	Description
None	Default behavior should be used
PreferFairness	Tasks scheduled sooner will be more likely to run sooner
LongRunning	Specifies that a task will be a long-running and oversubscription may occur
AttachedToParent	Specifies that a task is attached to a parent in the task hierarchy
DenyChildAttach	Prevent attaching a child task
HideScheduler	Use default as the current scheduler

Continuation Tasks

- Sometimes you must preserve execution order
  - **Continuation tasks enable in-order task execution**
  - **The result is a pipeline of task executions**
  - **Data can flow through the pipeline**
  - **Continuation can be conditional**
    - When the previous task has failed, succeeded, canceled, completed, or the opposite of the condition
    - When a group of tasks has completed
  - **Continuation can be a fork to many tasks**
  - **Continuation can handle exception thrown by antecedent**
  - **Continuation can have advanced options such as PreferFairness, LongRunning, AttachedToParent, DenyChildAttach, …**

DEMO

Continuation Task

Nested and Child Tasks

- Nested ➔ Created in the context of another task
- Child ➔ Nested Task that is created with the AttachedToParrent option

Category	Nested Tasks	Attached Child Tasks
Outer task (parent) waits for inner tasks to complete.	No (Yes by explicitly waiting)	Yes
Parent propagates exceptions thrown by children (inner tasks).	No	Yes
Status of parent (outer task) dependent on status of child (inner task).	No	Yes

Task Cancellation

- Based on the .NET 4.X Cancellation model
- `CancellationTokenSource`
  - **Cancellation token creator**
  - **Cancellation request issuer to all cancellation token copies**
- `CancallationToken`
  - **A Value Typed instance that is usually passed to the asynchronous operation**
  - **Communicate the cancellation request**
    - IsCancellationRequested
- Although the cancellation model does not enforce cancellation, many of the TPL mechanisms do it!

DEMO

Cancellation

Exception Handling

- Unhandled Exception are propagated from the task to the joining thread
  - **On `Task.Wait` or `Task<TResult>.Wait`**
- When waiting on many tasks, many exceptions can be thrown
  - **The TPL collect them into `AggregateException` instance**
  - **This instance is also used to propagate exception from child task to its parent**
  - **Use the `InnerExceptions` property to see them all**
- Cancellation can also be made by throwing an `OperationCanceledException`

The Static Parallel Class

- Also called Data Parallelism
- The `Parallel` class provides three groups of methods
  - `Parallel.Invoke`, `Parallel.For`, `Parall.ForEach`
- Invoke is an easy way to start several tasks and wait for them to finish
- For and ForEach:
  - **the source collection is partitioned so each segment runs in parallel to others**
  - **You can stop or break loop execution, monitor the state of the loop, use thread-local state, control the degree of concurrency, and so on**

Parallel Invoke

```
public static void Invoke(
ParallelOptions parallelOptions,
            params Action[] actions)
```

- This is the easiest way to run tasks in parallel

```
Parallel.Invoke(new ParallelOptions
                {MaxDegreeOfParallelism = 2},
                ()=> { Thread.Sleep(1000);
                       Console.WriteLine(1); },
                ()=> { Thread.Sleep(2000);
                       Console.WriteLine(2); }
                );
Console.WriteLine(3);
```

Parallel.For

- The simplest form:

```
public static ParallelLoopResult For(
        int fromInclusive,
        int toExclusive, Action<int> body)
```

- The most complex form:

```
public static ParallelLoopResult For<TLocal>(
        long fromInclusive, long toExclusive,
        ParallelOptions parallelOptions,
        Func<TLocal> localInit,
        Func<long, ParallelLoopState, TLocal, TLocal>
                body,
        Action<TLocal> localFinally)
```

Parallel.ForEach

- The simplest form:

```
public static ParallelLoopResult ForEach<TSource>
(IEnumerable<TSource> source, Action<TSource> body)
```

- The most complete form:

```
public static ParallelLoopResult ForEach<TSource, TLocal>(
        Partitioner<TSource> source,
        ParallelOptions parallelOptions,
        Func<TLocal> localInit,
        Func<TSource, ParallelLoopState, TLocal, TLocal>
                body,
        Action<TLocal> localFinally)
```

- Implementing custom partitioner

Stop or Break from a Parallel Loop

- The `ParallelLoopState` class enables iteration controlling
  - **The `Stop` method stops all executions**
    - `IsStopped` can tell you if Stop has been called
  - **The `Break` method is used like the original break keyword**
    - It means that no other iteration above the break value will run
      - However lower values iteration continue to run
  - **Use `ShouldExitCurrentIteration` property to stop execution of long running iterations**

```
Parallel.For(0, source.Length, (i, loopState) => {
        var result = Calc(source[i]);
        results.Add(result);
        if (results.Count > 100)
                loopState.Break(); } });
```

Cancel Parallel Loop

- Many of the Parallel methods get a `ParallelOptions` parameter
  - `MaxDegreeOfParallelism`, `TaskScheduler`, `CancallationToken`
- The `CancellationTokenSource` can be provided as a state object or as a capture variable
  - **This is needed if we want to cancel all tasks from within the loop's body**
- `CancellationToken.ThrowIfCancellationRequested` can be used to stop current execution with the `OperationCanceledException` exception
- See MSDN example and another example

TPL Summary

- TPL provides an abstraction over parallel tasks and loop execution
  - **Use the `Task` class for low-level and better control of task execution**
  - **Use Task Continuation to form a pipeline and a parallel network**
  - **Use the `Parallel.Invoke` to have an easy way to execute parallel actions and wait**
  - **Use the `Parallel.For` and `Parallel.ForEach` to execute parallel action on a sub-range of a collection**

Parallel LINQ

- Parallel LINQ-to-Objects (PLINQ)
  - **Built on top of Tasks**
  - **Leverage multiple cores**
  - **Fully supports all .NET standard query operators**
  - **Minimal impact to existing LINQ model**

```
var q = from p in people.AsParallel()
        where p.Name == queryInfo.Name &&
              p.State == queryInfo.State &&
              p.Year >= yearStart &&
              p.Year <= yearEnd
        orderby p.Year ascending
        select p;
```

DEMO

PLINQ

PLINQ

- Controlling
  - `AsParallel`, `AsSequential`, `AsOrdered`, `AsUnordered`
  - `WithCancellation`, `WithDegreeOfParallelism`, `WithExecutionMode`, `WithMergeOptions`
- Works for any IEnumerable<T>
  - **Optimizations for other types**
    - `T[]`, `IList<T>`
  - **Supports custom partitioning**
    - `Partitioner<T>`, `OrderablePartitioner<T>`

Beware of Sequential Mode

- PLINQ will always attempts to execute a query at least as fast as the query would run sequentially
  - **However it is not familiar with the actual computation**
  - **It does look at the query to find known templates that result in poor parallel performance**
    - In such a case it falls back to sequential mode
  - **For example a query that contains Reverse on a non-indexable data source**
  - **Look here for more PLINQ heuristics and speedup rules**
- Measure your PLINQ queries
  - **You can force parallelism with `ParallelExecutionMode. ForceParallelism`**

Merge Options in PLINQ

- `ParallelMergeOptions` Provides a hint to PLINQ about how it should merge the results from the various partitions

```
var result = from n in nums.AsParallel()
.WithMergeOptions(ParallelMergeOptions.NotBuffered)
 where IsPrime(n) select n;
```

Category	Description
`NotBuffered`	As soon as a result element has been computed, make that element available to the consumer of the query.
`AutoBuffered`	Results will be accumulated into many system selected size output buffers. This is the default policy.
`FullyBuffered`	All results will be accumulated before making any of them available to the consumer of the query.

Canceling PLINQ Query

```
var timeoutCancellationTokenSource =
                new CancellationTokenSource();
Task.Factory.StartNew(() => {
   Thread.Sleep(TimeSpan.FromSeconds(5));
   timeoutCancellationTokenSource.Cancel();});
try {
 var result = (from number in
       Enumerable.Range(1, 10000000).AsParallel().
       WithCancellation(timeoutCancellationTokenSource.Token)
       where IsPrime(number) select number).Average();
   Console.WriteLine(result);
} catch (OperationCanceledException e)
{       Console.WriteLine(e.Message); }
```

PLINQ Summary

- Parallel LINQ-to-Objects (PLINQ)
  - **Built on top of Tasks**
  - **Leverage multiple cores**
  - **Fully supports all .NET standard query operators**
  - **Minimal impact to existing LINQ model**
  - **Provides control fluent interface methods**
- Beware of sequential fall back
- Measure for getting the best results

Coordination Data Structures

Thread-safe collections

- ConcurrentStack<T>
- ConcurrentQueue<T>
- ConcurrentDictionary<TKey,TValue>
- ConcurrentBag<T>
- BlockingCollection<T>
- IProducerConsumerCollection<T>
- Partitioner, Partitioner<T>, OrderablePartitioner<T>

Initialization

- ThreadLocal<T>
- Lazy<T>, LazyInitializer

Synchronization

- CountdownEvent
- Barrier
- ManualResetEventSlim
- SemaphoreSlim
- SpinLock
- SpinWait

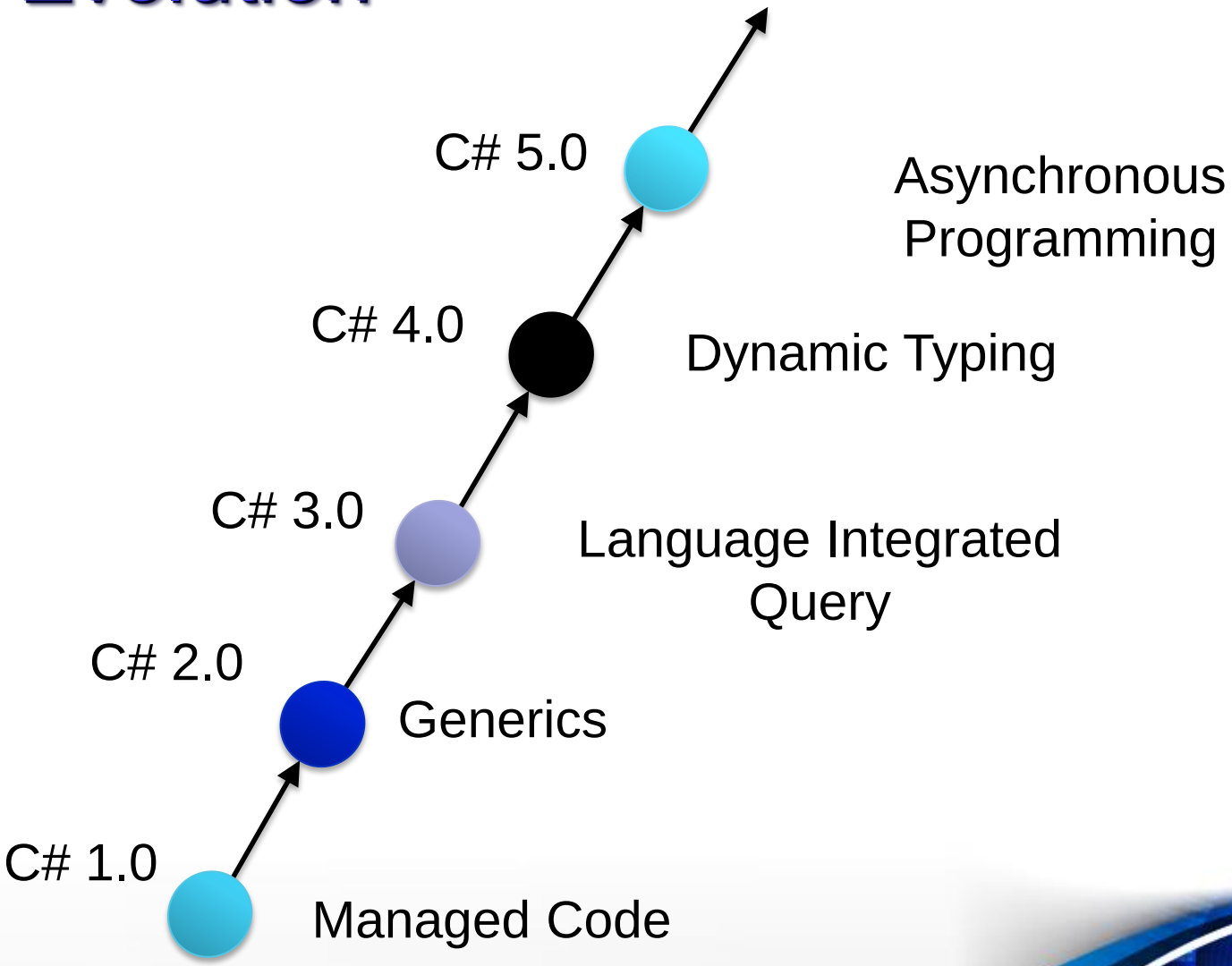
Cancellation

- CancellationToken
- CancellationTokenSource

The .NET Parallel Sample Pack

- The Sample Pack :
  - **Includes Example applications**
  - **It is not production quality**
  - **It is the playground of the Microsoft Parallel team**
- It contains the Parallel Extension Extras
  - LINQ to Tasks, Task<TResult>.ToObservable, Additional Task Extensions Methods, BlockingCollectionExtensions, StaTaskScheduler, ConcurrentExclusiveInterleave, Additional TaskSchedulers, ReductionVariable<T>, ObjectPool<T>, Pipeline, ParallelDynamicInvoke, AsyncCache, AsyncCall, SingleItemPartitioner, Specialized Task Waiting, Async Tasks for WebClient, SmtpClient, and Ping

C# Evolution



Asynchrony in a Nutshell

- Synchronous → Wait for result before returning
  - **string DownloadString(…);**
- Asynchronous → Return now, call back with result
  - **void DownloadStringAsync(…, Action<string> callback);**
- Asynchronous work != Threads

45

Asynchrony with C# 5.0

- Synchronous

```
private void OnGetData(object sender, RoutedEventArgs e) {
    _cmdGet.IsEnabled = false;
    var wc = new WebClient();
    _text.Text = wc.DownloadString(new Uri("http://msdn.microsoft.com"));
    _cmdGet.IsEnabled = true;
}
```

- Asynchronous with C# 5.0

```
private async void OnGetData(object sender, RoutedEventArgs e) {
    _cmdGet.IsEnabled = false;
    var wc = new WebClient();
    _text.Text = await wc.DownloadStringTaskAsync("http://msdn.microsoft.com");
    _cmdGet.IsEnabled = true;
}
```

Asynchronous methods

- Are marked with new **async** modifier
- Must return void or **Task<T>**
- Use **await** operator to cooperatively yield control
- Are resumed when awaited operation completes
- Can await anything that implements the “awaiter pattern”
- Execute in the synchronization context of their caller
- Allow composition using regular programming constructs
- Feel just like good old synchronous code!

47

What About Exceptions?

- Should be as simple as the synchronous case

```
var wc = new WebClient();
try {
  string txt = await wc.DownloadStringTaskAsync(url);
  dataTextBox.Text = txt;
}
catch(WebException x) {
  // Handle as usual
}
```

48

More Async

- Many methods and types are now built with the Task Asynchronous Pattern (TAP) in mind
  - **Stream.ReadAsync / WriteAsync**
  - **HttpClient class (new to .NET 4.5)**

49

Combining Tasks

- Task.WaitAll / WaitAny
  - **Waits for tasks to complete (not awaitable)**
- Task.WhenAll / Task.WhenAny
  - **Return a Task that represents the result of the given Task combination**
  - **Awaitable**
- More awaitables
  - **Task.Delay, Task.Yield**

50

Cancellation and Progress

- Asynchronous methods may support cancellation
  - **With the CancellationToken type**
- Can also report progress
  - **An IProgress<T> interface is defined**
  - **And one stock implementation, Progress<T>**
- Can use both

51

await Limitations

- Cannot use **await** inside a **catch** or **finally** block
  - **May be running due to an exception**
- Cannot use **await** inside a **lock** block
  - **Code may resume in a different thread**
- Cannot use **await** inside the Main method
  - **Main thread will end…**

DEMO

async/await

Concurrent Summary

- The free lunch is over
- The CPU abstraction
- Moore's Low and Multi-Core CPUs
- The Need for new abstraction
- A new free lunch approach

Thank You!

CodeValue