Task Parallel Library (TPL)

Purpose

- The Task Parallel Library (TPL) is a set of public types and APIs in the System.Threading and System.Threading.Tasks namespaces.
- TPL is to make developers more productive by simplifying the process of adding parallelism and concurrency to applications.
- TPL scales the degree of concurrency dynamically to most efficiently use all the processors that are available.
- TPL handles the partitioning of the work, the scheduling of threads on the ThreadPool, cancellation support, state management, and other low-level details.
- By using TPL, you can maximize the performance of your code while focusing on the work that your program is designed to accomplish.

Class Parallel

- Provides support for parallel loops and regions

- Parallel.For
- Parallel.ForEach
- Parallel.Invoke

Parallel.For signatures

- `For(Int32, Int32, Action<Int32>)`
- `For(Int32, Int32, ParallelOptions, Action<Int32>)`
- `For(Int32, Int32, ParallelOptions, Action<Int32,ParallelLoopState>)`
- `For<TLocal>(Int32, Int32, ParallelOptions, Func<TLocal>, Func<Int32,ParallelLoopState,TLocal,TLocal>, Action<TLocal>)`

ParallelOptions Class

- CancellationToken - Gets or sets the CancellationToken associated with this ParallelOptions instance.
- MaxDegreeOfParallelism - Gets or sets the maximum number of concurrent tasks enabled by this ParallelOptions instance.
- TaskScheduler - Gets or sets the TaskScheduler associated with this ParallelOptions instance. Setting this property to null indicates that the current scheduler should be used.

ParallelLoopState Class

- IsExceptional
- IsStopped
- LowestBreakIteration
- ShouldExitCurrentIteration

- Break()
- Stop()

Simple usage of Parallel.For

```
Parallel.For(fromInclusive: 0, toExclusive: 10, body: i =>
{
    int threadId = Thread.CurrentThread.ManagedThreadId;
    Console.WriteLine($"I: {i} (Thread #{threadId})");
});
Console.ReadKey();

// Result:
//      I: 2(Thread #3)
//      I: 4(Thread #5)
//      I: 5(Thread #5)
//      I: 7(Thread #5)
//      I: 9(Thread #5)
//      I: 1(Thread #5)
//      I: 3(Thread #3)
//      I: 6(Thread #6)
//      I: 0(Thread #1)
//      I: 8(Thread #4)
```

Parallel.ForEach signatures

- `ForEach<TSource>(IEnumerable<TSource>, ParallelOptions, Action<TSource>)`

- `ForEach<TSource>(IEnumerable<TSource>, Action<TSource,ParallelLoopState,Int64>)`

- `ForEach<TSource>(Partitioner<TSource>, …)`

- `ForEach<TSource,TLocal>(IEnumerable<TSource>, ParallelOptions, Func<TLocal>, Func<TSource,ParallelLoopState,TLocal,TLocal>, Action<TLocal>)`

Simple usage of Parallel.ForEach

```
string[] fileList = Directory.GetFiles("./");

Parallel.ForEach(fileList, body: file =>
{
    int threadId = Thread.CurrentThread.ManagedThreadId;
    int fileLineCount = File.ReadAllLines(file).Length;
    Console.WriteLine($"File '{file}' has {fileLineCount} lines (Thread #{threadId})");
});
Console.ReadKey();

// Result:
//      File './Program.cs' has 34 lines (Thread #1)
//      File './SimpleUsageOfParallelForEach.csproj' has 8 lines(Thread #3)
```

Simple usage of Parallel.Invoke

```
Action createAction(int j)
{
    return () => Console.WriteLine($"Action {j} invoked in {Thread.CurrentThread.ManagedThreadId}");
}

Action[] actionsArray = Enumerable.Range(0, 5).Select(createAction).ToArray();

Parallel.Invoke(actionsArray);
Console.ReadKey();

// Result:
//      Action 1 invoked in 5
//      Action 3 invoked in 6
//      Action 2 invoked in 4
//      Action 0 invoked in 1
//      Action 4 invoked in 3
```

Parallel.For Loop with Thread-Local Variables

```
long total = 0;
int[] nums = Enumerable.Range(0, 1000000).ToArray();

Parallel.For<long>(0, nums.Length, () => 0, (j, loop, subtotal) =>
{
    subtotal += nums[j];
    return subtotal;
}, x => Interlocked.Add(ref total, x));

Console.WriteLine($"The total is {total:N0}");
Console.ReadKey();

// Result:
//       The total is 499,999,500,000
```

Parallel.ForEach Loop with Thread-Local Variables

```
long total = 0;
int[] nums = Enumerable.Range(0, 1000000).ToArray();

Parallel.ForEach<int, long>(nums, () => 0, (j, loop, subtotal) =>
{
    subtotal += nums[j];
    return subtotal;
}, x => Interlocked.Add(ref total, x));

Console.WriteLine($"The total is {total:N0}");
Console.ReadKey();

// Result:
//      The total is 499,999,500,000
```

Cancel a Parallel.For or ForEach Loop

```
void MathFunction(int num) => Console.WriteLine($"Sqrt of {num} is {Math.Sqrt(num)}");
int[] nums = Enumerable.Range(0, 1000000).ToArray();

using (var cancellationTokenSource = new CancellationTokenSource())
{
    var options = new ParallelOptions();
    options.CancellationToken = cancellationTokenSource.Token;

    try
    {
        Task.Run(() =>
        {
            Thread.Sleep(50);
            cancellationTokenSource.Cancel();
        });
        Parallel.ForEach(nums, options, MathFunction);
    }
    catch (OperationCanceledException e)
    {
        Console.WriteLine(e.Message);
    }
}
Console.ReadKey();

// Result:
//      Sqrt of 0 is 0
//      Sqrt of 500000 is 707.106781186548
//      Sqrt of 250000 is 500
//      The operation was canceled.
```

Questions

- What is the difference between Parallel.For and Parallel.ForEach
- Why are Parallel loops need?

Class Task

Represents an asynchronous operation.

- Task - does not return a value
- Task<TResult> - returns a value with given type

Difference between Task and Thread

- Task represents an asynchronous operation (in .NET)
- Thread is one of the many possible workers which performs that task

Task.Delay vs Thread.Sleep

- Thread.Sleep – blocks current thread
- Taks.Delay – logical delay without blocking the current thread

```
void DoSomething()
{
    Thread.Sleep(100);
    // Do something after 100 ms
}
```

```
void DoSomething()
{
    Task.Delay(100);
    // Do something immediately
}
```

Usage of Tasks

```
void simpleFunc()
{
    Thread.Sleep(100);
    Console.WriteLine($"Invoked in {Thread.CurrentThread.ManagedThreadId}");
}

Task tR1 = Task.Run(new Action(simpleFunc));
tR1.Wait();
// Invoked in 3


Task tN1 = new Task(simpleFunc);

tN1.Start();
tN1.Wait();
// Invoked in 4

Console.ReadKey();
```

Task constructor

- Task(Action)
- Task(Action, CancellationToken)
- Task(Action, CancellationToken, TaskCreationOptions)
- Task(Action, TaskCreationOptions)
- Task(Action<Object>, Object)

TaskCreationOptions Enumeration

- AttachedToParent
- DenyChildAttach
- HideScheduler
- LongRunning
- None
- PreferFairness
- RunContinuationsAsynchronously

Sequence of tasks

TaskCreationOptions AttachedToParent

```
var parent = new Task(() =>
{
    Console.WriteLine("Outer task executing. #" + Thread.CurrentThread.ManagedThreadId);

    Task child = new Task(() =>
    {
        Console.WriteLine("Nested task starting. #" + Thread.CurrentThread.ManagedThreadId);
        Thread.SpinWait(500000);
        Console.WriteLine("Nested task completing. #" + Thread.CurrentThread.ManagedThreadId);
    }, TaskCreationOptions.AttachedToParent);
    child.Start();
});
parent.Start();
parent.Wait();
Console.WriteLine("Outer has completed. #" + Thread.CurrentThread.ManagedThreadId);

//      Outer task executing. #3
//      Nested task starting. #4
//      Nested task completing. #4
//      Outer has completed. #1
```

Awaiter

```
Task<int> primeNumberTask = Task.Run (() =>
    Enumerable.Range (2, 3000000).Count (n =>
        Enumerable.Range (2, (int) Math.Sqrt(n)-1).All (i => n % i > 0)));

var awaiter = primeNumberTask.GetAwaiter();

awaiter.OnCompleted (() =>
{
    int result = awaiter.GetResult();
    Console.WriteLine (result); // Writes result
});
```

Task.ContinueWith

```
void WriteWithThreadId(string message)
{
    Console.WriteLine($"{message} #" + Thread.CurrentThread.ManagedThreadId);
}
```

```
Task first = new Task(() => WriteWithThreadId("Execute first "));
Task second = first.ContinueWith(previousTask => WriteWithThreadId("Execute second "));

first.Start();
second.Wait();

// Result:
//      Execute first  #3
//      Execute second  #4
Console.ReadKey();
```

TaskContinuationOptions Enumeration

Extends TaskCreationOptions

- ExecuteSynchronously
- LazyCancellation
- NotOnCanceled
- NotOnFaulted
- NotOnRanToCompletion
- OnlyOnCanceled
- OnlyOnFaulted
- OnlyOnRanToCompletion

TaskContinuationOptions example

```
var t = new Task(() =>
{
    Console.WriteLine("t1 " + Thread.CurrentThread.ManagedThreadId);
    throw new Exception();
});
t.ContinueWith(t1 =>
{
    Console.WriteLine("Faulted " + Thread.CurrentThread.ManagedThreadId);
}, TaskContinuationOptions.OnlyOnFaulted);
t.ContinueWith(t1 =>
{
    Console.WriteLine("RanToCompletion " + Thread.CurrentThread.ManagedThreadId);
}, TaskContinuationOptions.OnlyOnRanToCompletion);

t.Start();

// t1 3
// Faulted 4
```

Task.WaitAll

```
void writeWithDelay(int delay, string message)
{
    Thread.Sleep(delay);
    Console.WriteLine(message + " #" + Thread.CurrentThread.ManagedThreadId);
}

Task[] tasks =
{
    Task.Run(() => writeWithDelay(300, "Second")),
    Task.Run(() => writeWithDelay(400, "Third")),
    Task.Run(() => writeWithDelay(50, "First"))
};

var allExecuted = Task.WaitAll(tasks, 1000);
Console.WriteLine(allExecuted ? "All executed" : "End with timeout");

// Result
//      First #6
//      Second #4
//      Third #7
//      All executed
```

Task.WaitAll

```
Task[] tasks =
{
    new Task(() => WriteWithDelay(300, "Second")),
    new Task(() => WriteWithDelay(400, "Third")),
    new Task(() => WriteWithDelay(50, "First"))
};

foreach (var task in tasks)
{
    task.Start();
    // No one will executed without start
    //  and WaitAll will return false by timeout
}

var allExecuted = Task.WaitAll(tasks, 1000);
Console.WriteLine(allExecuted ? "All executed" : "End with timeout");

// Result
//      First #6
//      Second #4
//      Third #7
//      All executed
```

Task.WaitAny

```
Task[] tasks =
{
    new Task(() => WriteWithDelay(300, message:"Second")),
    new Task(() => WriteWithDelay(300, message:"Third")),
    new Task(() => WriteWithDelay(100, message:"First"))
};

tasks.ToList().ForEach(x :Task => x.Start());

var taskIndex :int = Task.WaitAny(tasks, millisecondsTimeout:100);

if (taskIndex < 0)
{
    Console.WriteLine("Timeout");
}
else
{
    Console.WriteLine("Executed task index is " + taskIndex);
}

// Result
//      First #6
//      Executed task index is 2
//      Second #4
//      Third #7
```

Task.WhenAll

```
var tasks = new []
{
    Task.Run(() => ReturnWithDelay(500, 1)),
    Task.Run(() => ReturnWithDelay(500, 2)),
    Task.Run(() => ReturnWithDelay(100, 3)),
    Task.Run(() => ReturnWithDelay(500, 1))
};

var superTask = Task.WhenAll(tasks);
var result = superTask.Wait(1000);
if (result)
    Console.WriteLine("Sum is " + superTask.Result.Sum());
else
    Console.WriteLine("End by timeout");

// Sum is 7
```

Task.WhenAny

```
var tasks = new[]
{
    Task.Run(() => ReturnWithDelay(500, 1)),
    Task.Run(() => ReturnWithDelay(200, 2)),
    Task.Run(() => ReturnWithDelay(500, 3)),
    Task.Run(() => ReturnWithDelay(500, 1))
};

var superTask = Task.WhenAny(tasks);
var result = superTask.Wait(1000);
if (result)
    Console.WriteLine("First executed task returned " + superTask.Result.Result);
else
    Console.WriteLine("End by timeout");

// First executed task returned 2
```

Handle exceptions in Tasks

```
Task task = Task.Run(() => throw new Exception("Task1 exception"));

try
{
    task.Wait(); // Throws exception
}
catch (AggregateException e)
{
    Console.WriteLine(task.Status);
    //  Faulted

    Console.WriteLine(e.Message);
    //  One or more errors occurred. (Task1 exception)

    Console.WriteLine(e.InnerExceptions.First().Message);
    //  Task1 exception
}
```

Handle exceptions in Tasks

```
Task task2 = Task.Run(() => throw new Exception("Task2 exception"));

while (!task2.IsCompleted) { }

Console.WriteLine(task2.Status);
//  Faulted

Console.WriteLine(task2.Exception.Message);
//  One or more errors occurred. (Task1 exception)

Console.WriteLine(task2.Exception.InnerExceptions.First().Message);
//  Task2 exception
```

CancellationTokenSource

- CancellationTokenSource()
- CancellationTokenSource(int)
- CancellationTokenSource(TimeSpan)

- Token { get; }
- IsCancellationRequested { get; }
- Cancel() and CancelAfter(int)

Cancel task

```
using (var cts = new CancellationTokenSource())
{
    Action simpleFunc = () => { while (true) cts.Token.ThrowIfCancellationRequested(); };

    Task task = Task.Run(simpleFunc, cts.Token);
    cts.CancelAfter(100);

    try
    {
        task.Wait();
    }
    catch (AggregateException e)
    {
        Console.WriteLine($"{e.Message} {e.InnerExceptions.First().Message}");
    }
}
Console.ReadKey();

// Result:
//      One or more errors occurred. (A task was canceled.) A task was canceled.
```

Don’t cancel task

```
using (var cancelTokenSource = new CancellationTokenSource())
{
    var task = Task.Run(() =>
    {
        for (int i = 0; i < 1000; i++)
        {
            Thread.Sleep(100);
            Console.WriteLine(i);
        }
    }, cancelTokenSource.Token);

    cancelTokenSource.CancelAfter(TimeSpan.FromSeconds(3));
    task.Wait();
}
```

Task.Factory

```
var t1 = Task.Factory.StartNew(() => WriteWithDelay(500, "StartNew"));
var t2 = Task.Run(() => WriteWithDelay(1000, "Task.Run"));

Task.Factory.ContinueWhenAny(new[] {t1, t2}, task => Console.WriteLine("ContinueWhenAny")).Wait();
// Result:
//      StartNew 3
//      ContinueWhenAny
//      Task.Run 4


Task.Factory.ContinueWhenAll(new[] {t1, t2}, tasks => Console.WriteLine("ContinueWhenAny"));

// Result:
//      StartNew 3
//      Task.Run 4
//      ContinueWhenAny
```

Task Scheduler

- `protected internal abstract void QueueTask(Task task)`
- `protected abstract bool TryExecuteTaskInline(Task task, bool taskWasPreviouslyQueued)`
- `protected abstract IEnumerable<Task> GetScheduledTasks()`

How to use custom task scheduler

- ```
  Task.Factory.StartNew(() => SomeMethod(),
  CancellationToken.None, TaskCreationOptions.None,
  myTaskScheduler);
  ```
  ```

Question

- Difference between Thread and Task?
- When we use Thread.Sleep? When Task.Delay?
- How to handle exception in Task?
- Ways to create Task
- What is Task scheduler?

Parallel LINQ

ParallelQuery

- Extends:
  - IEnumerable
- Methods:
  - IEnumerator<TSource> GetEnumerator()

Methods

- AsParallel
- AsSequential
- AsOrdered
- AsUnordered
- WithCancellation
- WithDegreeOfParallelism (1 - 512)
- WithMergeOptions
- WithExecutionMode
- ForAll
- Aggregate

WithMergeOptions

- WithMergeOptions(ParallelMergeOptions)
- ParallelMergeOptions Enumeration
  - Default
  - NotBuffered
  - AutoBuffered
  - FullyBuffered

WithExecutionMode

- WithExecutionMode(ParallelExecutionMode)
- ParallelExecutionMode Enumeration
  - Default
  - ForceParallelism

ParallelQuery.ForAll vs ParallelForEach

- `myList.AsParallel().ForAll(i => { });`
- `Parallel.ForEach(mylist, i => { });`

Aggregate

- ```
  Aggregate<TSource,TAccumulate,TResult>(
      ParallelQuery<TSource>, TAccumulate,
      Func<TAccumulate,TSource,TAccumulate>,
      Func<TAccumulate,TResult>)
  ```

- ```
  Aggregate<TSource,TAccumulate,TResult>(
      ParallelQuery<TSource>, Func<TAccumulate>,
      Func<TAccumulate,TSource,TAccumulate>,
      Func<TAccumulate,TAccumulate,TAccumulate>,
      Func<TAccumulate,TResult>)
  ```

- ```
  Aggregate<TSource,TAccumulate>(
              ParallelQuery<TSource>, TAccumulate,
      Func<TAccumulate,TSource,TAccumulate>)
  ```
  ```

Recommendation for avoid potential problems with parallelism

- Avoid Writing to Shared Memory Locations
- Avoid Over-Parallelization
- Avoid Calls to Non-Thread-Safe Methods
- Be Aware of Thread Affinity Issues (i.e. access to UI controls)
- Do Not Assume that Iterations of ForEach, For and ForAll Always Execute in Parallel

Recommendation for avoid potential problems with parallelism

- Avoid Executing Parallel Loops on the UI Thread

```
private void button1_Click(object sender, EventArgs e)
{
    Parallel.For(0, N, i =>
    {
        // do work for i
        button1.Invoke((Action)delegate { DisplayProgress(i); });
    });
}
```

```
private void button1_Click(object sender, EventArgs e)
{
    Task.Factory.StartNew(() =>
        Parallel.For(0, N, i =>
        {
            // do work for i
            button1.Invoke((Action)delegate { DisplayProgress(i); });
        })
         );
}
```

Questions

- Difference PLINQ and LINQ? When we should use PLINQ?
- Difference between Parallel.ForEach and PLINQ ForAll?
- When we should use AsOrdered and AsUnordered?