.Net 4 Tasks, Async-Await, Parallel and PLINQ

Introduction

This document is meant to give a deep dive into running Async Await, compared to normally using tasks. On top of that, we want to know if there are differences when we use Async Await, Parallel library or PLINQ when processing loops of that can run in parallel.

How do tasks actually work

First, the new task must be created some examples include:
Task.Factory.StartNew(ACTION);
New Task(ACTION); Note you will need to call the Start method in order to start the task.
Task.Run(ACTION); Note: By default, you cannot attach child tasks here more on this later

After the start is called the task will go through some logic to determine what variables from the parent function should be captured inside the expression in case of a lambda. When capturing is done the task will be scheduled, by default on the default task scheduler.
In the Scheduler, the incoming task is placed on a queue and will be picked up by one of the worker threads of default ThreadPool. Once picked up the task will enter its processing state.

While the task is running you can attach events that will execute right after the main event has been completed. You can add them by calling the GetAwaiter and then the OnComplete function which takes an Action.
Now depending on your Options or way of creating your new task the state will be set to complete before or after the OnComplete is complete. The OnComplete becomes a new task attached to the executing task and Actions in the OnComplete sequence will run Synchronously after another. After the parent task and the OnComplete task are completed the parent task will enter its RanToCompletion state.

An important note to this is if there is not sufficient memory available the OnComplete task might run on another thread.

Why use Tasks instead of running purely synchronous code

Tasks will not block threads while they are waiting for something else to finish.

For example, when doing an I/O operation, this could be a network request or a read operation from a storage device.

If you are running your code synchronously you will block the thread you are running at while you are waiting for an I/O operation on the network device or storage device to finish.
While in contrast to that a Task will not block the thread because it will be able to yield it's executing while it is waiting for the result from the network device or storage device. Thereby enabling the CPU to do work in the background.

This example also applies to your code needing to wait for other tasks to complete.

Findings by browsing through the source

When creating a new task with Task.Factory.StartNew()
The factory will create a new task on the default task scheduler with a parent if called from inside another task.

If you are using task.GetAwaiter().GetResult() is actually just a wrapper to access the internal result of the Task and does the exact same as task<>.Result.

When accessing the tasks result while the task is not complete. The runtime thread will be put into 10 cycles of 'spinning' with an increasing duration compared to the previous cycle before occasional yielding. After this, the thread will be put into sleep. This mechanism helps by not blocking other threads for execution.

If you want to attach an action to a task that will be executed right before the task ends then use the OnComplete method, this method will be executed in all situations before the task enters the "RanToCompletion" state

Note: creating a new task with Task.Run() no child tasks can be attached this means all tasks created in that method OnCompleted will not wait for children to finish and thus runs Asynchronously.

Task scheduler decides if a task should run Synchronously or ASynchronously but the user can ultimately make the call by giving a task modified TaskCreationOptions

What does the Parallel Library do?

The parallel library is an implementation of the tasks system. As the name implies it will run iterations inside of a range in parallel thereby speeding up the time it will take in a regular loop, normally this will run on a single thread but can be altered by modifying the "parallelOptions'.

What you might have already expected is that Parallel 'For' or 'ForEach' only do the following:

1. Create an instance for keeping track at which is the current index that is being executed

2. for each index that is being executed a new Task will be created and configured to run on the same thread as the loop does, and it then waits for the tasks to complete their processing (you can modify this behavior by specifying options in parallel.for method).

Though a parallel loop compared to a task array with an awaited Task.WhenAll() will use fewer resources because the parallel loop only creates a struct to evaluate where it is and will wait only for one task at once while in comparison to the Task.WhenAll will create a new task that will subscribe itself to the complete action of each task it is monitoring.

Parallel versus PLINQ expressions

We all probably really like PLINQ with all the provided extension functions but you might be wondering, they also have 'AsParallel' included in their library, does that also use Tasks. The answer to that is PLINQ parallel does use the task library but only uses as many tasks as the user-defined by using the WithDegreeOfParallelism setting and will try to split the work into those tasks. This makes PLINQ way more efficient rather than waiting for relatively simple functions done in a query with high task startup time respectively.

Note in Linq: AsParallel + WithDegreeOfParallelism you can increase the number of tasks and thus you can run on multiple threads

Parallel pros & cons

- **PRO** Efficient for expensive functions because it is creating a new task for each iteration
- **CON** Overhead cost might be higher than executing inexpensive functions

PLINQ pros & cons

- **PRO** Very efficient when executing inexpensive functions
- **CON** Will become less efficient when the query becomes expensive to execute thereby it might make more sense running the normal parallel library

Async-await

The Await keyword functions as an alias for GetAwaiter().GetResult() on the task and adds a small difference in functionality but gives the user a cleaner looking code by reducing the amount that needs to be written by the user.
But looking into the (IL)Intermediate language bytecode there is a drawback.

Advantages and disadvantages of using Async Await as opposed to using Tasks

Advantages:

1. Clearer looking code
2. Less code

Disadvantages:

1. Losing the ability to use the ‘In’ keyword in the function declaration
2. Losing the ability to lock while awaiting a task
3. Losing the ability to do any operation in unsafe context(extends to the fixed keyword as well)
4. Creation of a nested class for each Async method implemented when compiled

Deep Dive

After digging inside the compiled IL code and I found that the async await generates a new class that contains all the will have all local variables as classifiers variables and implements the IAsyncStateMachine interface it also adds an AsyncTaskMethodBuilder and awaiters for each awaited task inside the function.

Then in a defined function called MoveNext, the actual code you expected is written, and for each task that is running the TaskAwaiter will be stored in its classifier. When the await statement is encountered the TaskAwaiter.GetResult() is called. All logic inside the MoveNext function is wrapped in a try and catch to catch any errors that might occur

On this note, I have to also mention that when you use the Async keyword but not Await it will still generate a nested class in your compiled code but will only house not make any notable modifications to the code

Illustration

When using tasks:

```
public Task<IEnumerable<File>> FindAllFiles(string query = "")
{
    return Task<IEnumerable<File>>.Factory.StartNew(() =>
    {
        Task<FileList> requestTask = Task<FileList>.Factory.StartNew(() => new
FileList());
        double result = 152.4382;
        Task.Delay((int)result).GetAwaiter().GetResult();
        return requestTask.GetAwaiter().GetResult().Files;
    });
}
```

When using Async Await:

```
public async Task<IEnumerable<File>> FindAllFiles(string query = "")
{
    Task<FileList> requesttask = Task<FileList>.Factory.StartNew(() => new
FileList());
```

```
    double result = 152.4382;
    await Task.Delay((int)result);
    return (await requesttask).Files;
}
```

Compiled Async Await code Becomes:

```
sealed class FindFiles : IAsyncStateMachine
{
    public int state;
    public AsyncTaskMethodBuilder<IEnumerable<File>> t_builder;
    public string query;
    public ParentClass __this;
    private Task<FileList> requesttask;
    private double result;
    private FileList s__3;
    private TaskAwaiter u__1;
    private TaskAwaiter<FileList> u__2;

    public override void MoveNext(){
        IEnumerable<File> files;
        try{
            requesttask = Task<FileList>.Factory.StartNew(() => new FileList());
            result = 152.4382;
            s__3 = u__2.GetResult();
            u__1 = Task.Delay((int)result).GetAwaiter();
            u__2 = requesttask.GetAwaiter();
            u__1.GetResult();
        }
        catch(System.Runtime){
            state= -2;
            t__builder.SetException(exception);
            return;
        }

        files= u__2.GetResult();
        t__builder.SetResult(files);
    }

     void IAsyncStateMachine.SetStateMachine(IAsyncStateMachine stateMachine)
    {
        this.SetStateMachine(stateMachine);
    }
```

```
}


public Task<IEnumerable<File>> FindAllFiles(string query = "")
{
    FindFiles d__9 stateMachine = new FindFiles();
    stateMachine.__this = this;
    stateMachine.query = query;
    stateMachinet.__builder = AsyncTaskMethodBuilder<IEnumerable<File>>.Create();
    stateMachine.__state = -1;
    AsyncTaskMethodBuilder<IEnumerable<File>> <>t__builder = stateMachine.t__builder;
    t__builder.Start(ref stateMachine);
    return stateMachine.t__builder.Task;
}
```

A good thing to note, The task of an Async Await function is if everything goes well not Actually running but will run the code inside the function Synchronously but instead after the function completes will have it create a new instance of task with the result set inside the constructor.

This is also the reason why if you make an async function without Await it won't actually run anything asynchronously.

Async await versus Parallel versus PLINQ

Let's compare the implementations while we want to run a loop with multiple tasks and waiting for them to complete

Async Await

```
async Task<IEnumerable<float>> RunTasks()
{
    Task<float>[] tasks = new Task<float>[50];
    for (int i = 0; i < tasks.Length; i++)
    {
        tasks[i] = Task<float>.Factory.StartNew(() => 798234f);
    }

    await Task.WhenAll(tasks);

    return tasks.Select(task => task.Result);
}
```

Will create 50 tasks that will execute asynchronously

```
Parallel
IEnumerable<float> RunTasks()
{
       float[] output = new float[50];
       Parallel.For(0, 50, index =>
           {
               output[index] = 234234f;
           });
       return output;
}
```

Creates 50 tasks that execute asynchronously

Will block parent function from continuing, can be altered by wrapping it in a task

```
PLINQ
IEnumerable<float> RunTasks()
{
    IEnumerable<int> range = Enumerable.Range(0, 50);

    float[] output = new float[50];
    range
        .AsParallel()
        .WithDegreeOfParallelism(4)
        .ForAll(index => { output[index] = 524f; });

    return output;
}
```

Creates 4 tasks that execute asynchronously

Thereby reducing the overhead cost of starting multiple tasks

This will block parent function from continuing, can be altered by wrapping it in a task

And for reference, how to run without any implementations

```
IEnumerable<float> RunTasks()
{
    Task<float>[] tasks = new Task<float>[50];
    for (int i = 0; i < tasks.Length; i++)
        tasks[i] = Task<float>.Factory.StartNew(() => 531f);

    Task.WhenAll(tasks);
```

```
    return tasks.Select(task => task.Result);
}
```

Note: the loop with async-await probably makes use of an await Task.WaitAll() or some equivalent for minimizing the number of sleep cycles from the runtime-thread

Async Await

- Straightforward to write, few worries about concurrency settings
- Minimal performance impact

Parallel:

Pros

- Easy to start many heavy operations

Cons

- Overhead cost increases when the function becomes less expensive

PLINQ

Pros

- Most efficient for executing small expressions
- intuitive to chain multiple operations together

Cons

- Will not automatically scale when not configured

Conclusion

All 3 implementations are built upon the Tasks Library and therefore all implementations have the advantage of being efficient by not dedicating threads to relatively smaller operations. This is even elevated by the functionality from tasks to be able to Yield execution and give room for other operations to be executed.

Async-Await tries to improve the readability of your code,
By automatically implementing TaskAwaiters and yielding executing without much effort from the user.
But this has 2 trade-offs

- it creates nested classes in your compiled code.
- Losing the ability to use the keywords ‘in’ ‘lock’ ‘fixed’ and ‘unsafe’

But, it does have a small impact on performance and Thus comes down personal preference to use async-await or just plain tasks because the functionality is already available to the user.

The Parallel Library creates a new task for every iteration all the iterations, this could lead to large overhead if you are going over small expressions

The PLINQ library is out of the box very efficient for smaller expressions by being conservative with creating tasks, the user has to manually select how much the degree of parallelism he wants per query. This is nice because you don't have to deal with high overhead when running long queries.

I found this to be a really interesting investigation to undertake with some unexpected turns where I had my head wrapped. I hope that whoever read this enjoyed it, thank you for reading, Mark Oostveen.

Sources

https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/parallel-linq-plinq
https://docs.microsoft.com/en-us/dotnet/api/system.linq.parallelenumerable?view=netframework-4.8
https://github.com/microsoft/referencesource/tree/master/System.Core/System/Linq
https://docs.microsoft.com/en-us/dotnet/csharp/async
https://docs.microsoft.com/en-us/dotnet/standard/async-in-depth
https://en.wikipedia.org/wiki/Futures_and_promises
https://docs.microsoft.com/en-us/dotnet/standard/asynchronous-programming-patterns/task-based-asynchronous-pattern-tap
https://docs.microsoft.com/en-us/dotnet/standard/threading/spinwait
https://en.wikipedia.org/wiki/Yield_(multithreading)#targetText=In%20computer%20science%2C%20yield%20is,of%20the%20same%20scheduling%20priority.

Second answer on this post
https://stackoverflow.com/questions/9719003/spinwait-vs-sleep-waiting-which-one-to-use#targetText=In%20.NET%204%20SpinWait%20performs,for%20a%20set%20time%20period.&targetText=With%20Thread.Sleep%20the%20thread%20is%20blocked.

In more detail
http://www.albahari.com/threading/part5.aspx#_SpinWait

browsed the source code of Microsoft to reference and lookup design choices and to see if the implementation comes with some drawbacks
https://referencesource.microsoft.com/download_48.html

to investigate the workings of the Async Await keyword I used ILSpy to find out what the difference in the compilation is. I Recommend taking a look at the AsyncTaskMethodBuilder and all workings associated with it. If you can't figure out what calls your async method is making to the AsyncTaskMethodBuilder you've to look at the IL or the C# version but set to C# 1.0 - 4.0
https://github.com/icsharpcode/ILSpy