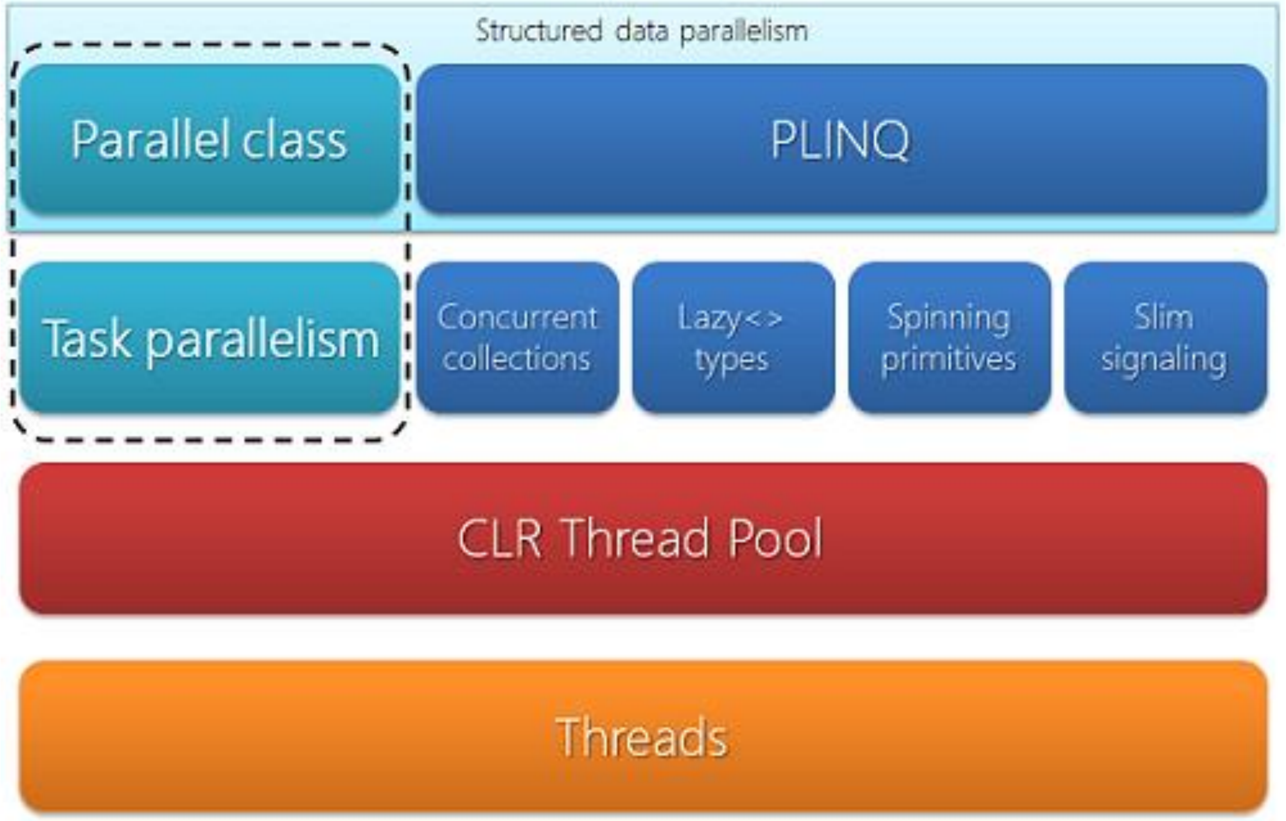
Task Parallel Library (TPL)

TPL



- a set of useful classes and methods for tasks and powerful (parallel) extensions to LINQ in form of PLINQ
- PLINQ part can be triggered by calling the AsParallel extension method before calling other LINQ extension methods
- PLINQ queries usually tend to run slower than their sequential counterparts, since most queries do not have enough computational time required to justify the overhead of creating a thread

Parallel.For()(1)

The sequential version

```
int N = 10000000;
double sum = 0.0;
double step = 1.0 / N;

for (var i = 0; i < N; i++)
{
    double x = (i + 0.5) * step;
    sum += 4.0 / (1.0 + x * x);
}

return sum * step;
```

Parallel.For()(2)

The parallel version 1:

```
object _ = new object();
int N = 10000000;
double sum = 0.0;
double step = 1.0 / N;

Parallel.For(0, N, i =>
{
    double x = (i + 0.5) * step;
    double y = 4.0 / (1.0 + x * x);
    lock(_)
    {
        sum += y;
    }
});

return sum * step;
```

- a lock-block is the required for synchronization
- the synchronization overhead is much more than we gain by using multiple processors (the workload besides the synchronization is just too small)
- a better version uses another overload of the For method, which allows the creation of a thread-local variable

Parallel.For()(3)

The parallel version 2:

```
object _ = new object();
int N = 10000000;
double sum = 0.0;
double step = 1.0 / N;
Parallel.For(0, N, () => 0.0, (i, state, local) =>
{
    double x = (i + 0.5) * step;
    return local + 4.0 / (1.0 + x * x);
}, local =>
{
    lock (_)
    {
        sum += local;
    }
});
return sum * step;
```

- It is more efficient because the lock section exists only once per thread instead of once per iteration we actually drop a lot of the synchronization overhead.
- The third parameter is the delegate for creating the thread-local variable. In this case we are creating one double variable *x*.
- If the variable would have already been created it would not be thread-local but global.
- The state parameter gives us access to actions like breaking or stopping the loop execution (or realizing in what state we are), while the local variable is our access point to the thread-local variable (in this case just a double).

Task vs. thread

- Thread is something from the OS (a kind of resource)
- Task is just some class. It is not true that all running Task instances are based on a Thread.
- In fact all IO bound asynchronous methods in the .NET-Framework, which return a Task<T> are not using a thread. They can use callback based, i.e. system notifications or already running threads from drivers or other processes.
- In .NET applications Task based solutions are preferable in general.
- Actual used resource (callback handler or a thread) does not matter anymore.

Task(1)

```csharp
Task<double> SimulationAsync()
{
    //Create a new task with a lambda expression
    var task = new Task<double>(() =>
    {
        Random r = new Random();
        double sum = 0.0;
        for (int i = 0; i < 10000000; i++)
        {
            if (r.NextDouble() < 0.33)
                sum += Math.Exp(-sum) + r.NextDouble();
            else
                sum -= Math.Exp(-sum) + r.NextDouble();
        }
        return sum;
    });
    task.Start(); //Start it and return it
    return task;
}
```

Task(2)

Example 1

```
var sim = SimulationAsync();
var res = sim.Result;//Blocks the current execution until the result is available
sim.ContinueWith(task =>
{
    //use task.Result here!
}, TaskScheduler.FromCurrentSynchronizationContext());
//Continues with the given lambda from the current context
```

Example 2

```
var sim1 = SimulationAsync();
var sim2 = SimulationAsync();
var sim3 = SimulationAsync();
var firstTask = Task.WhenAny(sim1, sim2, sim3);//This creates another task! (callback)
firstTask.ContinueWith(task =>
{
    //use task.Result here, which will the the result of the first task that finished
    //task is the first task that reached the end
}, TaskScheduler.FromCurrentSynchronizationContext());
```

Keywords await, async(1)

- C# 5, two new keywords have been introduced: await and async
- By using *async* we mark methods as being asynchronous, i.e. the result of the method will be packaged in a Task (if nothing is returned) or Task<T> if the return type of the method would be T

Convention

Classical methods

```
void DoSomething()
{
}

int GiveMeSomething()
{
    return 0;
}
```

Asynchronous methods

```
Task async DoSomethingAsync()
{
}

Task<int> async GiveMeSomethingAsync()
{
    return 0;
}
```

Keywords await, async(2)

- Await means asynchronous wait
- It can only be used inside async marked methods, since only those methods will be changed by the compiler
- It tells the runtime to wait for a task result before assigning it to a local variable, in the case of tasks which return values; or simply wait for the task to finish, in the case of those with no return value

Keywords await, async(3)

```
// 1. Awaiting For Tasks With Result:
async void LoadAndPrintUserNameAsync()
{
    // Create, start and wait for the task to finish; then assign the result to a local variable.
    var user = await Task.Factory.StartNew<User>(() => DataContext.Users.FindByName("luis.aguilar"));
    // At this point we can use the loaded user.
    Console.WriteLine("User loaded. Name is " + user.Name);
}
// 2. Awaiting For Task With No Result:
async void PrintRandomMessage()
{
    // Create, start and wait for the task to finish.
    await Task.Factory.StartNew(() => Console.WriteLine("Not doing anything really."));
}
// 3. Usage:
void RunTasks()
{
    // Load user and print its name.
    LoadAndPrintUserNameAsync();
    // Do something else.
    PrintRandomMessage();
}
```

Zadania do samodzielnej realizacji

1. Wykonać implementację równoległą i sekwencyjną funkcji obliczającej iloczyn skalarny dwóch wektorów
2. Wykonać implementację równoległą i sekwencyjną funkcji obliczającej iloczyn dwóch macierzy
3. Wykonać implementację równoległą i sekwencyjną funkcji sortującej rosnąco elementy tablicy jednowymiarowej